

Universitatea din București
Facultatea de Matematică și Informatică
Specializarea Informatică

LUCRARE DE LICENȚĂ

Coordonator științific:

Conf. univ. dr. Cristina DĂSCĂLESCU

Absolvent:

Rareș-Petru OPĂRIUC

BUCUREȘTI

2019

Universitatea din București
Facultatea de Matematică și Informatică
Specializarea Informatică

LUCRARE DE LICENȚĂ

Immoborcars – Aplicație web pentru gestionarea ofertelor
pentru locuințe și automobile

Coordonator științific:

Conf. univ. dr. Cristina DĂSCĂLESCU

Absolvent:

Rareș-Petru OPĂRIUC

BUCUREȘTI

2019

Abstract

Achiziționarea unei locuințe și a unui automobil a fost întotdeauna o prioritate pentru majoritatea oamenilor, de-a lungul vieții. Această lucrare prezintă scopul și implementarea unei platforme care, prin arhitectura și funcționalitățile sale, își propune să îi ajute pe cei care sunt în căutarea celor mai sus menționate. Pe lângă crearea unei interfețe plăcute, intuitive și ușor de folosit de către persoane aparținând tuturor categoriilor de vârstă, aplicația ImmoBorCars oferă utilizatorilor săi și oportunitatea de a vedea o apreciere a prețului cerut de vânzători, în funcție de specificațiile fiecărui anunț.

Cuprins

Introducere	1
Capitolul I – Arhitectura proiectului	3
1.1. Tehnologii utilizate în implementare	3
1.2. Proiectarea bazei de date	5
1.3. Proiectarea interfețelor de utilizator	8
1.4. Server	8
1.5. Client	9
Capitolul II – Implementarea platformei	10
2.1. Comunicarea prin servicii REST	10
2.1.1. Controllere	10
2.1.2. Metode de tip GET, POST, PUT, DELETE	11
2.1.3. Folosirea DTO (Data Transfer Objects)	13
2.2. Maven.....	15
2.2.1. Dependente	16
2.3. Spring Boot.....	18
2.3.1. Spring Data JPA.....	19
2.3.2. Securitate	22
2.3.3. Injecție pe baza adnotărilor	23
2.3.4. Logging.....	23
2.3.5. Debugging.....	24
2.4. Algoritmul Gradient Boosting Regression	24
2.5. React.....	26
2.5.1. Componente	27
2.5.2. Limbajul JSX	30
2.5.3. Prop vs. State.....	31
2.5.4. Compoziție versus moștenire	32
2.5.5. Rute	33
2.5.6. Încărcarea de fișiere.....	33
2.5.7. Iterații async.....	34
2.5.8. Google Maps API.....	35
2.6. Design Patterns	36
2.6.1. Builder	36

Capitolul III – Prezentarea aplicației.....	38
3.1. Ecranul principal.....	38
3.2. Pagina de login.....	39
3.3. Pagina de înregistrare.....	40
3.4. Formularul de adăugare al unui anunț.....	41
3.5. Pagina de profil a unui utilizator.....	41
Concluzii.....	42
Bibliografie.....	44

Lista figurilor

Fig. 2.1 – Diagrama arhitecturii aplicației	3
Fig. 2.2 – Dependență în pom.xml	5
Fig. 2.3 – Configurare proprietăți baza de date.....	5
Fig. 2.4 – Diagrama entităților	7
Fig. 2.5 – Interfețele utilizatorilor	8
Fig. 3.1 – Annotările unui controller	11
Fig. 3.2 – Metodă din controller.....	12
Fig. 3.3 – Solicitare GET în Postman	12
Fig. 3.4 – Metodă de adăugare din controller pentru case	13
Fig. 3.5 – Metoda de creare a unui HouseResponse.....	14
Fig. 3.6 – Metoda de GET a unei case.....	15
Fig. 3.7 – Fișierul pom.xml fără dependențe	15
Fig. 3.8 – Rularea aplicației cu Maven	16
Fig. 3.9 – Utilizarea Lombok în cadrul aplicației.....	17
Fig. 3.10 – Diagrama Spring Boot [4]	18
Fig. 3.11 – Clasa PagedResponse.....	20
Fig. 3.12 – Returnarea unui răspuns paginat.....	20
Fig. 3.13 – Metode din cadrul HouseRepository.....	21
Fig. 3.14 – Răspunsul unei solicitări fără token	22
Fig. 3.15 – Fereastră de debug	24
Fig. 3.16 – Metoda de antrenare a modelului.....	25
Fig. 3.17 – Metoda de accesare a aprecierii prețului	26
Fig. 3.18 – Limbaj JSX în cadrul componentei House	28
Fig. 3.19 – Secțiune a componentei House	28
Fig. 3.20 – Componenta HouseBox	29
Fig. 3.21 – Notificare în cazul unei erori	30
Fig. 3.22 – Exemplu de utilizare a sintaxei JSX	31
Fig. 3.23 – Exemplu de folosire a compoziției în React	32
Fig. 3.24 – Componenta Filepond	33
Fig. 3.25 – Metodă recursivă de adăugare a unor poze	34
Fig. 3.26 – Câmpul de adresă din formularul de adăugare sau editare	35
Fig. 3.27 – Tabloul de bord pentru cheia Google API folosită	36
Fig. 3.28 – Folosirea unui builder în cadrul aplicației	37
Fig. 4.1 – Ecranul principal al unui utilizator înregistrat	38
Fig. 4.2 – Pagina de login	39
Fig. 4.3 – Validări pentru câmpuri	39
Fig. 4.4 – Formularul de înregistrare.....	40
Fig. 4.5 – Formularul de adăugare al unui anunț.....	41
Fig. 4.6 – Pagina de profil.....	41

Introducere

Conform unei statistici realizate de Eurostat în anul 2011, 96,6% din populația României deținea un apartament sau o casă, țara noastră situându-se, la acel moment, pe primul loc în clasamentul Uniunii Europene. La polul opus se afla Germania, cu numai 53,4%. Această diferență uriașă este explicabilă din motive istorice, întrucât în majoritatea țărilor ce au aparținut fostului bloc sovietic, după căderea regimului comunist, proprietățile private ce fuseseră naționalizate au fost restituite, iar cetățenii au putut cumpăra locuințe la un preț mult mai accesibil.

Situația este diferită, însă, astăzi. Locuințele devin din ce în ce mai scumpe, crizele economice amenință piețele interne cu fluctuații foarte mari de preț, iar băncile nu pot oferi niciun fel de garanție pentru creditele în alte monede pe care le oferă oamenilor pentru aproape jumătate din viața acestora (de exemplu, cazul francului elvețian, a cărui valoare s-a dublat în decursul a doar 10 ani). Totuși, dorința de a deține o locuință se manifestă, în continuare, în rândul majorității persoanelor, indiferent de țara în care locuiesc, iar unul din cei mai importanți factori de luat în calcul este prețul.

Ideea principală a acestui proiect a fost legată de oferirea unei aprecieri corecte a prețului pentru cele mai importante, respectiv de valoare, două bunuri în care oamenii vor investi bani de-a lungul vieții: locuința și automobilul. În ultimul timp, s-au făcut publice tot mai multe înșelătorii în capcana cărora au căzut oameni nevinovați, prin care fie același imobil a fost folosit pentru a lua avans mai multor persoane, fie dezvoltatorii nu au terminat de construit locuințele, fie le-au construit ilegal și multe altele. Toate au avut la bază un aspect important care i-a atras pe cei pătimiți: prețul foarte atrăgător.

Astfel, am dorit să dezvolt o aplicație care, pe baza unui algoritm de machine learning și a unui set de date în continuă creștere, să vină în ajutorul oamenilor prin oferirea unei comparații între prețul cerut pentru un bun și prețul mediu cu care unul asemănător a fost vândut în realitate. Prin urmare, scopul aplicației este de a-i face pe utilizatori să se gândească de două ori când o ofertă are un preț cu mult sub cel real, dar să le și ofere posibilitatea de a alege anunțurile cu preț competitiv.

În timpul explorării soluțiilor deja existente, am remarcat câteva aplicații care, într-o oarecare măsură, încearcă a oferi funcționalități asemănătoare cu cele ale aplicației ImmoBorCars. În cele ce urmează, vom analiza principalele diferențe dintre soluția propusă și cele existente:

- OLX – probabil cea mai populară platformă de vânzări online, însă nu oferă utilizatorilor posibilitatea de a vedea o apreciere a prețului;
- Storia – oferă un preț mediu, calculat pe baza celui mai mic și a celui mai mare preț din zonă;
- imobiliare.ro – oferă un preț minim și un preț maxim al imobilelor cu zonă și număr de camere asemănător, dar de obicei diferența între cele două este prea mare și nu se iau în calcul și alți factori precum suprafața, vechimea etc.

Prin urmare, în comparație cu aplicațiile deja existente la momentul actual, ImmoBorCars își propune folosirea unui algoritm de machine learning pe baza căruia poate fi făcută o regresie după un set de date în continuă schimbare, nu doar prin calcularea unei medii între prețul minim și cel maxim.

Capitolul I – Arhitectura proiectului

În acest capitol vor fi prezentate principalele diagrame de design ale aplicației, precum și descrierea modului în care tehnologiile folosite au fost integrate. Mă voi concentra pe o delimitare clară a fiecărei părți, precum și pe modul de comunicare al acestora.

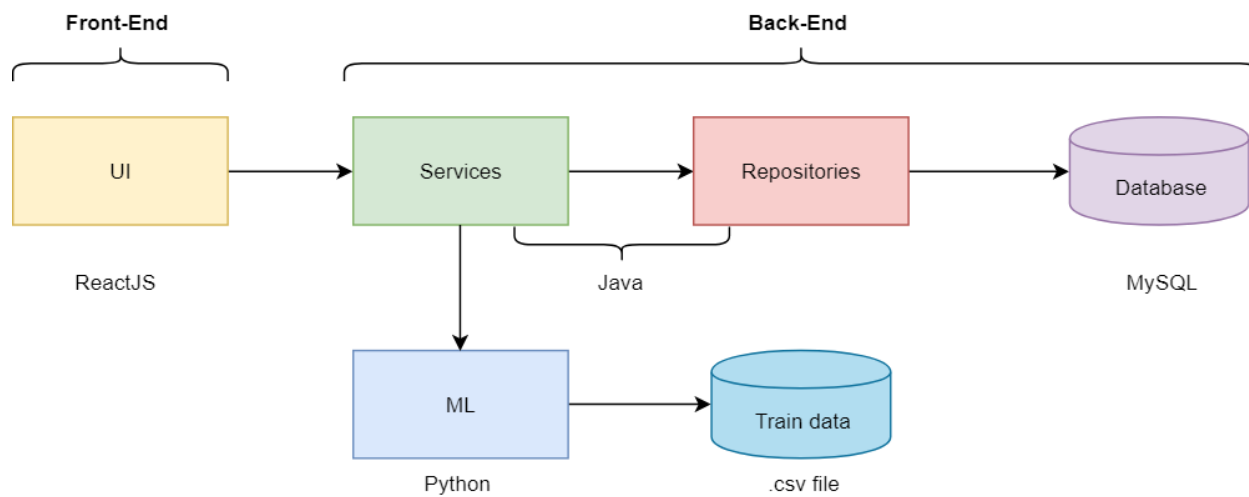


Fig. 2.1 – Diagrama arhitecturii aplicației

1.1. Tehnologii utilizate în implementare

Pentru dezvoltarea acestui proiect, am ales utilizarea următoarelor tehnologii și unelte:

- Java 8, Spring Boot, Maven
- Python, Flask
- ReactJS, Node.js, NPM
- IntelliJ IDE
- MySQL Database
- AntDesign Framework

Java se numără printre cele mai populare limbaje de programare existente, fiind folosit la o scară foarte mare, atât pentru aplicații executabile, cât și pentru aplicații web. Cele mai importante motive luate în calcul în alegerea tehnologiei pentru partea de back-end au fost:

- Compatibilitate foarte bună, aplicațiile Java fiind cross-platform, adică pot rula pe aproape toate sistemele de operare [1];
- Integrare ușoară cu alte proiecte, unde utilitarul Maven pentru Java oferă posibilitatea adăugării de dependențe către alte proiecte, de exemplu pentru metode gata definite de repository, pentru securitate etc.;
- Suportul din mediul online, Java având a doua cea mai mare pondere de utilizatori pe StackOverflow și alte site-uri web asemănătoare, unde se găsesc cu ușurință soluții la majoritatea problemelor întâlnite în dezvoltare.

Fiind o aplicație web, am avut nevoie și de un framework special conceput pentru a ne permite dezvoltarea acesteia. Am ales Spring, mai exact Spring Boot, cel din urmă fiind o versiune construită pe baza framework-ului Spring, dar care aduce foarte multe configurări predefinite.

Tot pentru partea de back-end, însă doar pentru algoritmul de machine learning ce va oferi o aproximare a prețului corect al unui imobil sau autovehicul, am ales Python întrucât este cel mai utilizat limbaj în scrierea algoritmilor de ML, oferind suport foarte bun și în mediul online. La fel ca în cazul anterior, a fost nevoie și de un framework pentru dezvoltare web, iar în acest caz am ales Flask.

Pentru front-end, am ales să folosesc ReactJS, având ca principale avantaje structura bazată pe componente, foarte utilă în astfel de aplicații. Limbajul JSX (specific React) permite injectarea unor obiecte JavaScript direct în codul HTML, reușind astfel o foarte ușoară și permisivă integrare a celor două limbaje din urmă. Mai mult, bibliotecile existente în React permit folosirea unor componente gata create, ce folosesc diferite API-uri, cum ar fi cel pentru Google Maps, despre care vom vedea mai multe în capitolele următoare. Asemănător cu tehnologiile anterioare, am folosit Node.js ca server pentru web și NPM pentru administrarea dependențelor.

Cele trei aplicații menționate anterior sunt complet decuplate una de cealaltă, rulând fiecare pe câte un port diferit și comunicând prin servicii HTTP Rest.

1.2. Proiectarea bazei de date

Aplicația ImmoBorcars folosește o bază de date SQL, ale cărei entități (tabele) sunt automat generate (sau actualizate, dacă este cazul) la fiecare rulare a aplicației. Acest lucru ne este pus la dispoziție de către componenta Spring Data JPA, ce are ca implementare implicită Hibernate. Datorită utilizării Maven, putem folosi implementările menționate anterior prin adăugarea unei singure dependențe în fișierul specific „pom.xml”.

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

Fig. 2.2 – Dependență în pom.xml

Ulterior, vom configura fișierul „application.properties”, adăugând datele de acces pentru baza de date, alături de modul în care framework-ul Hibernate va acționa asupra acesteia la fiecare rulare a aplicației.

```
spring.datasource.url = jdbc:mysql://localhost:3306/licenta?allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username = licenta
spring.datasource.password = Rares123!

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto = update
```

Fig. 2.3 – Configurare proprietăți baza de date

După setarea acestora, nu rămâne decât să creăm modelele în Java, pe care să le adnotăm corespunzător astfel încât framework-ul să le recunoască și să le adauge în baza de date.

Astfel, în urma analizei cerințelor funcționale ale aplicației, am putut identifica următoarele entități: Utilizator, Rol, Casă, Apartament, Mașină, Fișier. Alături de acestea se vor adăuga și tabelele automat generate în urma relaționării dintre entități, specificate tot prin adnotări. Câteva exemple de relaționări între entitățile aplicației sunt:

- Relația mai-mulți-la-mai-mulți între Utilizator și Rol, specificată prin adnotările `@ManyToMany` și `@JoinTable`, ce creează automat o tabelă de legătură, numită `User_Roles`, care transformă relația mai-mulți-la-mai-mulți în două relații unu-la-mai-mulți.
- Relațiile unu-la-mai-mulți între Utilizator și Casă, Apartament, Mașină, întrucât un utilizator poate adăuga mai multe case, apartamente sau autovehicule.
- Relațiile unu-la-mai-mulți între entitățile Casă, Apartament, Mașină și entitatea Fișier, pentru încărcarea de poze.

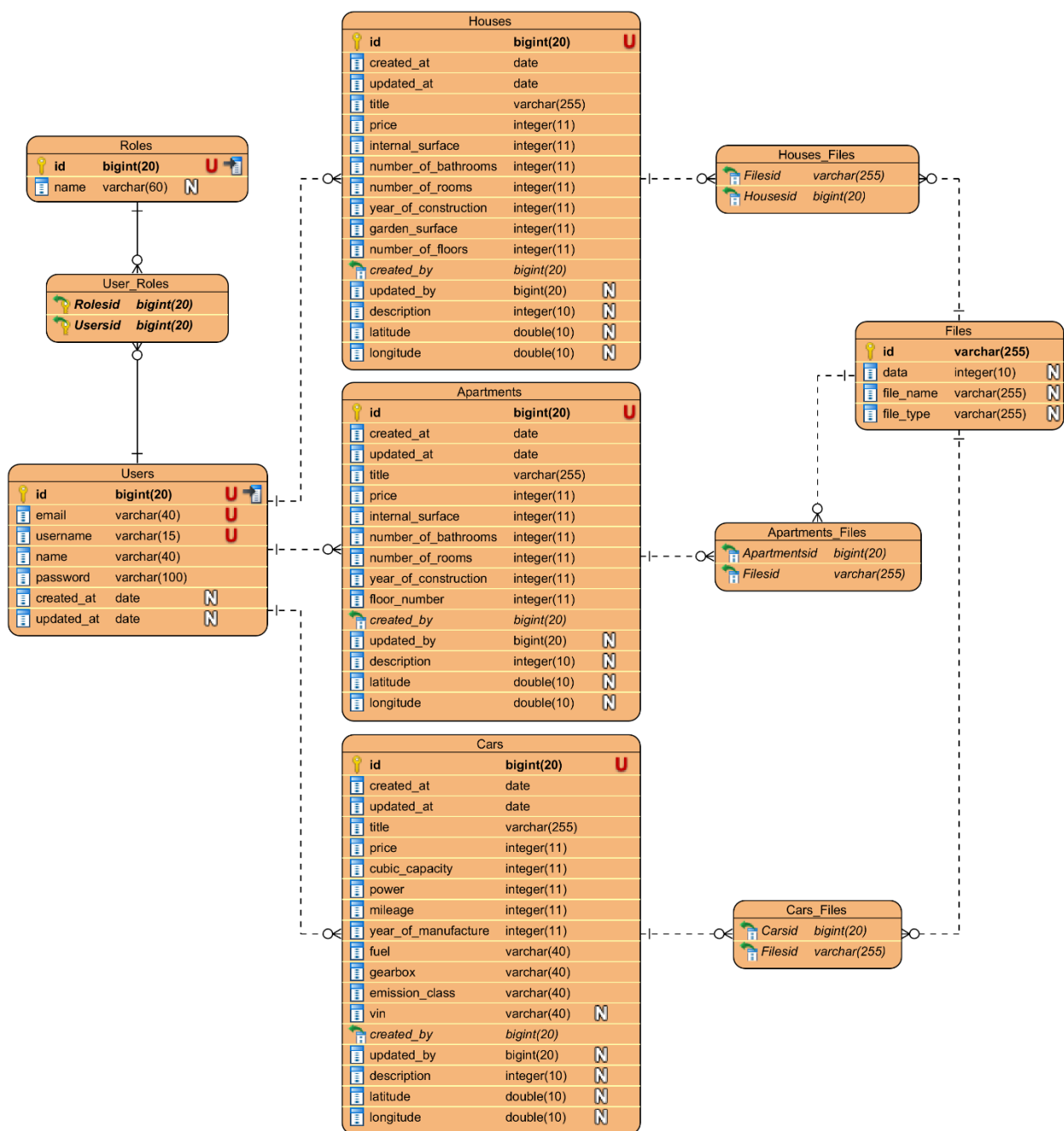


Fig. 2.4 – Diagrama entităților

1.3. Proiectarea interfețelor de utilizator

Interfețele utilizatorilor, reprezentate în Fig. 2.5, au ca scop delimitarea paginilor accesibile din orice punct și stare a aplicației. Crearea lor ajută la formarea unei imagini de ansamblu asupra funcționalităților aplicației și asupra dreptului fiecărui utilizator.

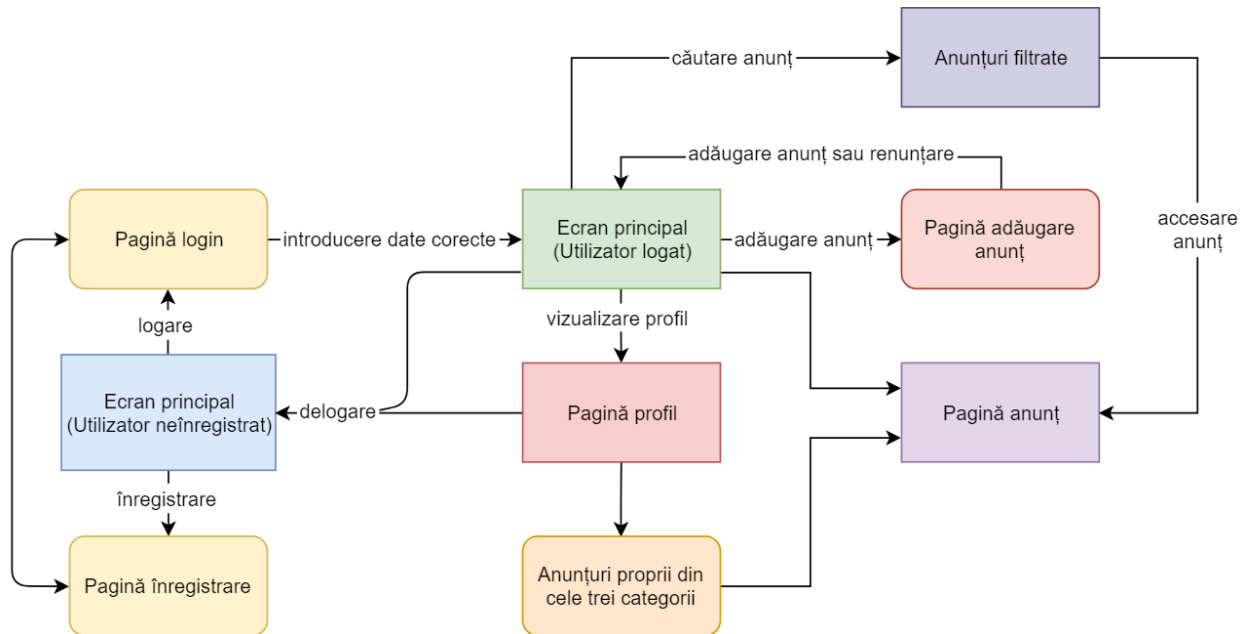


Fig. 2.5 – Interfețele utilizatorilor

1.4. Server

După cum se poate observa în Fig. 2.1, aplicația ImmoBorcars dispune de două servere de back-end, fiecare cu rolul său foarte bine delimitat. Primul dintre ele este serverul de Java cu Spring Boot, disponibil pe portul 8181, care menține comunicarea cu baza de date și prin care se realizează aproape toată logica de business a aplicației. Prin intermediul accesării sale din partea de front-end, entitățile sunt create, actualizate, șterse sau returnate spre a fi afișate utilizatorului.

Am ales să folosesc arhitectura bazată pe servicii și repository-uri, astfel încât solicitările vor fi făcute către un serviciu specific fiecărei entități, care la rândul său se va folosi de repository-ul

specific acesteia pentru a realiza operațiile în baza de date. Aceasta din urmă este disponibilă pe portul 3306.

Al doilea server este cel de Python cu Flask, responsabil de expunerea endpoint-urilor către algoritmul de Machine Learning, care vor oferi posibilitatea de antrenare a modelului și de a returna aprecierile de preț. Acest server este disponibil pe portul 5000. Algoritmul de regresie ales pentru această parte a proiectului este Gradient Boosting Regression, pe care îl vom detalia într-o secțiune următoare.

1.5. Client

Tehnologia aleasă pentru partea de front-end, React, a dictat de la sine modul în care s-a conturat design-ul acestei părți a proiectului. React este o bibliotecă de Javascript ce permite crearea unor componente separate pentru fiecare parte a aplicației web, fiind capabilă să le actualizeze eficient la nevoie, fără necesitatea reîncărcării întregii pagini. Astfel, design-ul se învâрте în jurul componentelor, care sunt asemănătoare claselor din Java. Acestea pot conține variabile locale, metode, dar și parametri primiți de la componenta părinte. Metoda principală a unei componente, `render()`, este cea prin care returnăm ceea ce vrem să afișăm în aplicație. Limbajul folosit în această metodă se numește JSX, fiind o îmbinare între JavaScript și HTML, foarte permisivă din punctul de vedere al transferului de informații între componente și, ulterior, către utilizatorul final.

Datorită celor prezentate mai sus, am reușit o delimitare destul de clară a componentelor în proiect, pornind de la izolarea modelului unei case sau al unui apartament, până la izolarea componentei unei hărți, unde existau, de exemplu, metode specifice pentru apelarea API-ului în vederea afișării hărții într-o pagină.

Toate solicitările către back-end le-am păstrat, de asemenea, în metode declarate într-o clasă special creată pentru asta, pe care am denumit-o `APIUtils`. Astfel, pe principiul includerii asemănător celorlalte limbaje de programare, le-am putut folosi printr-o simplă apelare din orice componentă am avut nevoie.

Capitolul II – Implementarea platformei

În acest capitol va fi prezentată, în detaliu, implementarea proiectului ImmoBorCars, aprofundând cele prezentate în capitolul I, la descrierea design-ului. Vom avea în vedere aprofundarea blocurilor din Fig. 2.1, a entităților din baza de date și a modului prin care ele comunică în cadrul aplicației.

2.1. Comunicarea prin servicii REST

În partea de back-end, cele două framework-uri folosite (Spring și Flask) ne pun la dispoziție expunerea unor endpoint-uri, pe baza cărora vor fi făcute solicitări din partea de front-end. Cu ajutorul adnotărilor specifice, putem defini metodele ce vor fi apelate în momentul în care aplicația va primi o solicitare la adresa specificată. În Spring Boot, adnotările sunt similare cu verbele HTTP, respectiv `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`. Acestea li se poate adăuga, între paranteze, calea către care se va face solicitarea, alături de parametrii necesari între paranteze de tip acoladă.

Aceste metode sunt, de obicei, grupate pe controllere, fiecare entitate din baza de date având propriul său controller. Ele primesc drept parametru un obiect, denumit generic Data Transfer Object, și pot returna, după caz, fie un alt obiect, fie un răspuns cu un cod HTTP. Spring Boot oferă și posibilitatea îmbinării acestora prin genericul `ResponseEntity`, clasă care conține obiectul ce trebuie returnat, alături de un obiect de tip `HttpStatus`, cu un cod specific HTTP.

2.1.1. Controllere

Pentru a defini o clasă de tip controller în Java, am folosit adnotările specifice Spring Boot `@RestController`, alături de `@RequestMapping("/adresa")`. Cea din urmă reprezintă adresa, concatenată la adresa IP a aplicației, la care vor fi disponibile metodele ce urmează a fi declarate în interiorul clasei.

```
@RestController
@RequestMapping("/api/houses")
public class HouseController {
```

Fig. 3.1 – Adnotările unui controller

De precizat este că adnotarea `@RestController` reprezintă o compunere a adnotărilor `@Controller` și `@ResponseBody`, cea din urmă indicând faptul că se dorește trimiterea obiectului returnat de metodă ca body în răspuns.

2.1.2. Metode de tip GET, POST, PUT, DELETE

În secțiunea anterioară, am arătat ce este un controller și cum a fost folosit în aplicație. În această secțiune, urmează prezentarea metodelor ce vor fi definite în controllere, pentru a permite comunicarea între cele 3 părți ale aplicației ImmoBorcars. Vom lua ca exemplu controller-ul specific modelului House.

După cum am spus, metodele sunt adnotate conform operației pe care o execută, însă există și alte adnotări pe care le putem folosi pentru acestea, puse la dispoziție de framework-ul Spring, care oferă diferite facilități, de exemplu pentru controlarea accesului utilizatorilor. Vom vedea în secțiunile următoare, la configurarea securității aplicației, care este principiul prin care funcționează securizarea metodelor din controllere. Eu am ales folosirea adnotării `@PreAuthorize`, care primește ca parametru un rol al utilizatorului, acesta din urmă definit tot manual, regăsindu-se în tabela Roluri, de care am amintit în capitolul II.

Ideal, aceste metode ar trebui să se folosească numai de metodele prezente în servicii, pentru ca straturile de abstractizare să fie foarte bine delimitate. Mai departe, serviciile vor folosi repository-urile pentru comunicarea cu baza de date.

```

@GetMapping("/{houseId}")
public HouseResponse getHouseById(@CurrentUser UserPrincipal currentUser,
                                   @PathVariable Long houseId) {
    return houseService.getHouseById(houseId, currentUser);
}

```

Fig. 3.2 – Metodă din controller

Drept exemplificare pentru o metodă de tip GET, am ales obținerea unei case după ID-ul său, acesta din urmă fiind transmis prin intermediul adresei la care se face solicitarea. Adresa se obține din concatenarea la adresa aplicației a câmpului specific pentru controller, apoi a câmpului specific pentru metodă. În cazul nostru, pentru exemplul dat, o adresă ar putea fi „localhost:8181/api/houses/2”, iar testarea o putem face, pentru început, prin aplicația Postman.

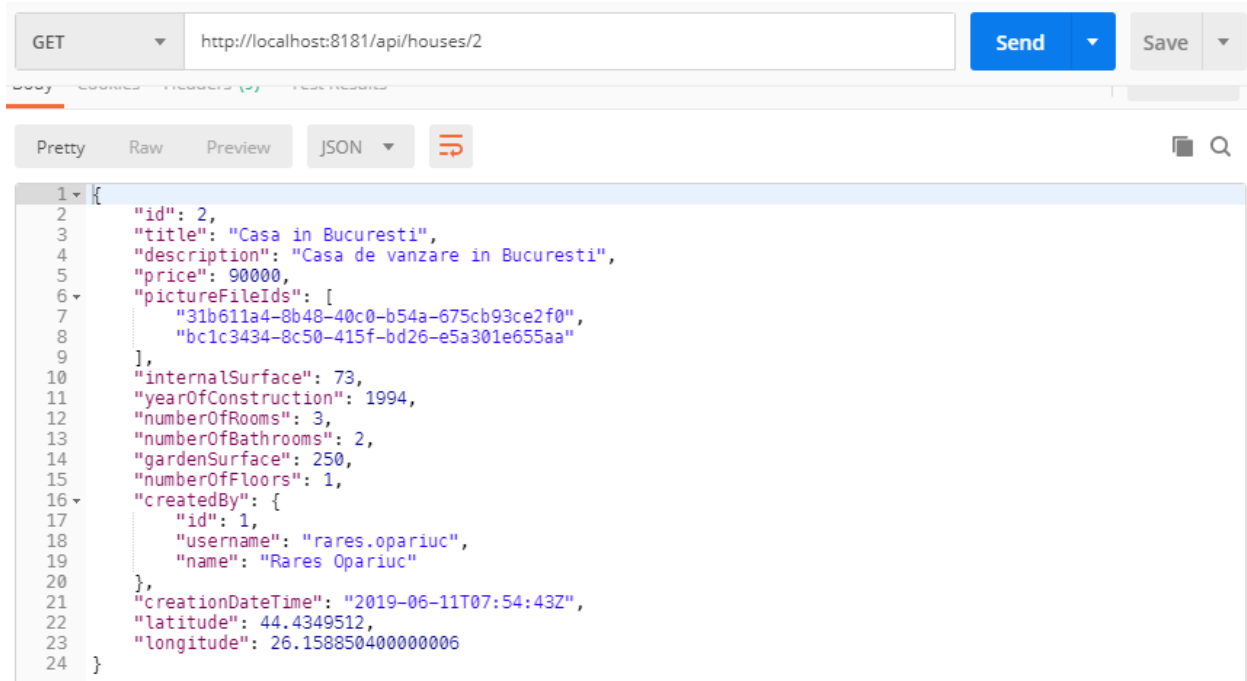


Fig. 3.3 – Solicitare GET în Postman

Un alt exemplu, de data aceasta pentru o metodă de tip POST, care conține și o adnotare pentru controlul accesului utilizatorilor, este pentru crearea unei case. Acesta returnează, în header, locația la care resursa adăugată poate fi accesată, iar în corpul răspunsului am ales să întorc ID-ul obiectului nou creat. Acesta va fi foarte util mai târziu, când la afișarea unei case vom putea crea, doar pentru utilizatorul proprietar al anunțului, butoanele de editare și de ștergere, ale căror adrese sunt dependente de ID-ul obiectului.

```
@PostMapping
@PreAuthorize("hasRole('USER')")
public ResponseEntity<> createHouse(@Valid @RequestBody HouseRequest houseRequest) {
    House house = houseService.createHouse(houseRequest);

    URI location = ServletUriComponentsBuilder
        .fromCurrentRequest().path("/{houseId}")
        .buildAndExpand(house.getId()).toUri();

    return ResponseEntity.created(location)
        .body(new ApiResponse( success: true, message: "House successfully created!", house.getId().toString()));
}
```

Fig. 3.4 – Metodă de adăugare din controller pentru case

2.1.3. Folosirea DTO (Data Transfer Objects)

Am văzut în exemplele anterioare că, în cadrul controller-ului pentru case, am folosit ca tip de returnare clasa numită HouseResponse. Acest fapt este datorat unui design pattern numit Data Transfer Objects (similar cu Data Access Objects), prin care se încearcă a se limita numărul de solicitări trimise către back-end pentru procesarea unei entități, dar și eliminarea câmpurilor care nu sunt relevante (de exemplu, câmpul ID nu este relevant într-o solicitare de tip POST pentru adăugarea unei case).

Să luăm ca exemplu entitatea House din aplicația ImmoBorCars. Aceasta extinde clasa abstractă ImmoBorCars, care la rândul său extinde clasa abstractă UserDateAudit, responsabilă de menținerea utilizatorilor care au creat sau actualizat un obiect. Aceasta din urmă conține adnotări specifice Spring Data JPA, din componenta AuditingEntityListener, care adaugă automat utilizatorul obiectului ce a fost creat. Însă, revenind la entitatea noastră, House, în momentul în care vom face

o solicitare pentru a vizualiza o anumită casă, dorim să vedem și cine a creat-o pentru a putea afișa datele de contact. Pentru a evita două solicitări prin care să obținem separat datele casei și separat datele utilizatorului, am creat clasa `HouseResponse` care le conține pe toate, iar procesarea va fi făcută la nivel de back-end în urma unei singure solicitări de returnare a unei case, după cum se observă în metoda de mai jos.

```
public static HouseResponse mapHouseToHouseResponse(House house, User creator) {  
  
    UserSummary creatorSummary = UserSummary.builder()  
        .id(creator.getId())  
        .username(creator.getUsername())  
        .name(creator.getName())  
        .build();  
  
    return HouseResponse.builder()  
        .id(house.getId())  
        .title(house.getTitle())  
        .description(house.getDescription())  
        .price(house.getPrice())  
        .pictureFileIds(house.getPictureFiles().stream()  
            .map(DBFile::getId).collect(Collectors.toList()))  
        .internalSurface(house.getInternalSurface())  
        .yearOfConstruction(house.getYearOfConstruction())  
        .numberOfRooms(house.getNumberOfRooms())  
        .numberOfBathrooms(house.getNumberOfBathrooms())  
        .gardenSurface(house.getGardenSurface())  
        .numberOfFloors(house.getNumberOfFloors())  
        .createdBy(creatorSummary)  
        .creationDateTime(house.getCreatedAt())  
        .latitude(house.getLatitude())  
        .longitude(house.getLongitude())  
        .build();  
}
```

Fig. 3.5 – Metoda de creare a unui `HouseResponse`

Modul în care această metodă este folosită se poate observa în metoda de returnare a casei după ID din serviciu, respectiv cea pe care am văzut-o apelată în controller, la începutul secțiunii 2.1.2.

```
public HouseResponse getHouseById(Long houseId, UserPrincipal currentUser) {  
    House house = houseRepository.findById(houseId).orElseThrow(  
        () -> new ResourceNotFoundException("House", "id", houseId));  
  
    User creator = userRepository.findById(house.getCreatedBy())  
        .orElseThrow(() -> new ResourceNotFoundException("User", "id", house.getCreatedBy()));  
  
    return ModelMapper.mapHouseToHouseResponse(house, creator);  
}
```

Fig. 3.6 – Metoda de GET a unei case

2.2. Maven

Așa cum am menționat la prezentarea arhitecturii proiectului, am folosit utilitarul Maven pentru a adăuga dependențe către alte proiecte, dar și pentru că este un foarte bun utilitar pentru build [3]. La baza lui stă fișierul „pom.xml”, în care sunt declarate toate dependențele, proprietățile, versiunile, plug-in-urile, fazele build-ului, module în cazul aplicațiilor mari și altele.

```
<?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>  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.1.3.RELEASE</version>  
        <relativePath/>  
    </parent>  
    <groupId>com.raresopariuc</groupId>  
    <artifactId>licenta</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
    <name>licenta</name>  
    <description>Proiect licenta Opariuc Rares-Petru (2019)</description>  
  
    <properties>  
        <java.version>1.8</java.version>  
    </properties>
```

Fig. 3.7 – Fișierul pom.xml fără dependențe

Rularea aplicației în timpul dezvoltării este făcută tot cu ajutorul său, mai exact prin intermediul plug-in-ului „Spring Boot Maven plugin”. Acesta din urmă ne pune la dispoziție mai multe configurații, cum ar fi `spring-boot:run` pentru a rula aplicația, `spring-boot:start` sau `spring-boot:stop` pentru a porni aplicația și a rula teste, precum și alte configurații pentru informații despre build sau pentru împachetare. Se poate folosi comanda „`mvn spring-boot:run`” dintr-un terminal, sau printr-o opțiune în cadrul IDE-ului, după cum se poate observa în figura de mai jos.

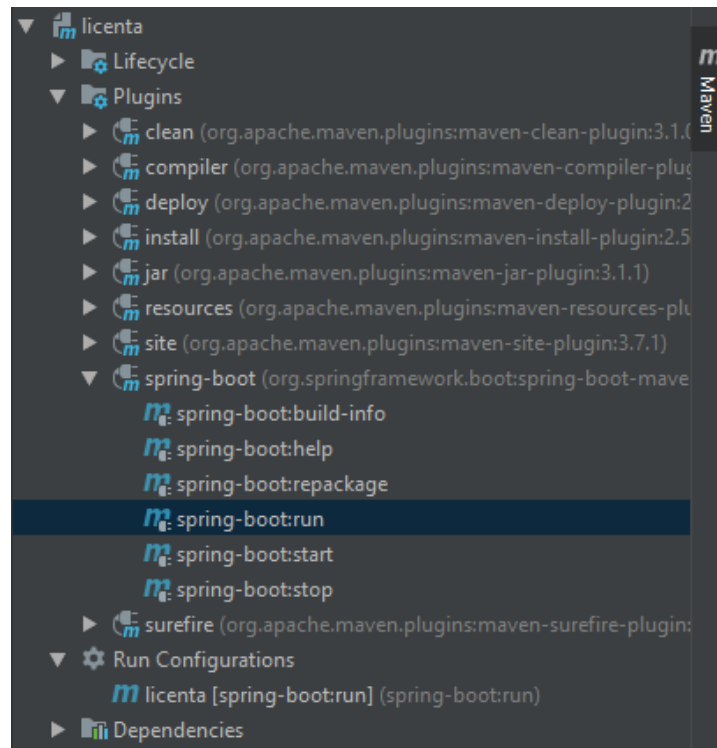


Fig. 3.8 – Rularea aplicației cu Maven

2.2.1. Dependente

O funcționalitate foarte folositoare a dependențelor din Maven este tranzitivitatea. Spre exemplu, dacă se adaugă o dependență directă către un proiect prin intermediul fișierului `pom.xml`, dependențele directe ale acelor proiecte vor fi incluse automat, devenind dependențe indirecte pentru proiectul nostru. Nu există o limită a numărului de nivele pentru care o dependență poate fi

inclusă în mod tranzitiv, însă trebuie să avem grijă să nu se formeze dependențe ciclice atunci când folosim un număr destul de mare.

Pentru fiecare dependență, poate fi specificat și scopul în care aceasta va fi folosită. De exemplu, putem avea nevoie de aceasta la compilare, la runtime sau în timpul testelor.

În momentul generării inițiale a proiectului Spring Boot, sunt adăugate automat câteva dependențe folositoare în funcție de tipul de aplicație pe care dorim să îl dezvoltăm. În cazul ImmoBorcars, dependențele adăugate inițial au fost pentru lucrul cu baza de date MySQL, pentru securitate, pentru teste și pentru JWT (Json Web Token), pe care îl vom detalia în secțiunile următoare.

2.2.1.1. Lombok

O dependență foarte folositoare, pe care am folosit-o în implementare, este Lombok. Acesta este o bibliotecă de Java care oferă un set de adnotări ce reduc foarte mult din duplicitatea codului, când vine vorba de gettere, settere, constructori și builder. Spre exemplu, în clasa House, am folosit adnotările `@Getter`, `@Setter` și `@NoArgsConstructor`.

```
@Entity
@Table(name = "houses")
@Getter
@Setter
@NoArgsConstructor
public class House extends Immobile {
    @NotNull
    @PositiveOrZero
    private Integer gardenSurface;

    @NotNull
    @PositiveOrZero
    private Integer numberOfFloors;

    @Builder
    public House(String title, String description, Integer price, Integer internalSurface, Integer ye
        Integer numberOfBathrooms, Integer gardenSurface, Integer numberOfFloors, Double lat
        this.setTitle(title);
        this.setDescription(description);
    }
```

Fig. 3.9 – Utilizarea Lombok în cadrul aplicației

Întrucât nu suportă, pentru moment, moșteniri, a fost necesară declararea în mod explicit a lui `@Builder` pentru a putea seta câmpurile moștenite din clasa abstractă `Immobile`. În cazul unei clase ale cărei câmpuri nu sunt moștenite din altă clasă, simpla utilizare a adnotării în același loc cu celelalte ar fi funcționat fără probleme.

2.3. Spring Boot

Spring Boot este un proiect construit pe baza framework-ului Spring pentru Java, folosit pentru a simplifica modul în care putem crea aplicațiile. Folosind Spring Initializr, am avut de ales între a crea un proiect Maven sau Gradle, am ales versiunea dorită, am adăugat câteva cuvinte cheie pentru ca Spring Boot să adauge automat dependențele (câteva exemple sunt Web, Security, JPA), după care am generat proiectul. Acesta a fost livrat sub forma unei arhive ce conținea structura predefinită a unui proiect de tip Spring, fișierul `pom.xml` cu toate dependențele adăugate și o clasă `Main`. Din acest punct, proiectul a fost deschis și dezvoltat cu ajutorul IntelliJ IDEA.

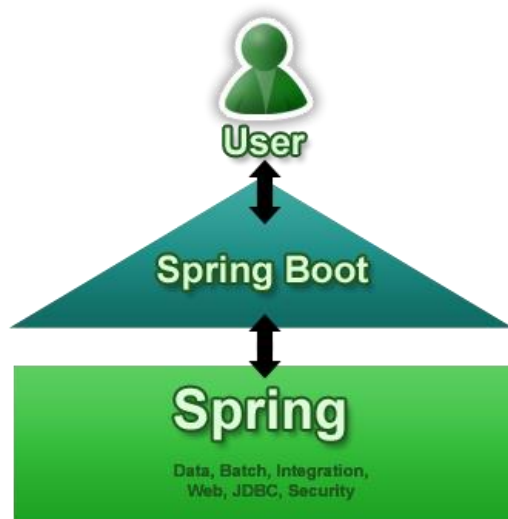


Fig. 3.10 – Diagrama Spring Boot [4]

De aici, putem folosi dependențele și plug-in-urile adăugate de Spring Boot pentru a dezvolta, porni, opri sau împacheta aplicația.

2.3.1. Spring Data JPA

Spring Data JPA este folosit pentru a facilita lucrul cu o bază de date relațională, prin asigurarea suportului în crearea unui repository de tip Java Persistence API (JPA). Astfel, ne sunt puse la dispoziție interfețe cu metode gata implementate, pentru a reduce codul duplicat în rândul aplicațiilor dezvoltate folosind acest framework. În aplicația ImmoBorcars, am extins fiecare repository din interfața `JpaRepository<ObjectType, ObjectIdType>`, interfață care a pus la dispoziție metode implementate pentru operațiile CRUD (create, read, update, delete). [5]

Rezultatele întoarse de aceste metode sunt încapsulate într-un obiect de tip `Optional`, pentru a evita primirea valorilor de tip `null`, însă pot fi și paginate, așa cum am făcut în cazul obiectelor returnate într-o listă, de exemplu pentru pagina principală a aplicației.

2.3.1.1. Răspunsuri paginate

Pentru a putea oferi utilizatorilor o experiență de utilizare cât mai plăcută, am ales oferirea rezultatelor unor liste de componente sub formă paginată. Astfel, am creat o clasă numită `PagedResponse<T>`, cu ajutorul căreia vom oferi, pe lângă o listă de obiecte ce fac parte din întreaga listă, și date despre pagina pe care se află elementele din listă, câte elemente există în total, câte pagini există în total și câte elemente au fost returnate. Toate aceste câmpuri vor fi populate cu răspunsurile oferite de metodele oferite de Spring Data JPA.

```

@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class PagedResponse<T> {
    private List<T> content;
    private int page;
    private int size;
    private long totalElements;
    private int totalPages;
    private boolean last;
}

```

Fig. 3.11 – Clasa PagedResponse

Astfel, în serviciul aferent fiecărei entități, vom putea returna un obiect de tipul clasei descrise mai sus, pentru a facilita afișarea paginată a răspunsurilor. Un exemplu concret din aplicație este cel al metodei de returnare a tuturor caselor, ilustrată în figura de mai jos.

```

public PagedResponse<HouseResponse> getAllHouses(UserPrincipal currentUser, int page, int size) {

    Pageable pageable = PageRequest.of(page, size, Sort.Direction.DESC, ...properties: "createdAt");
    Page<House> houses = houseRepository.findAll(pageable);

    if(houses.getNumberOfElements() == 0) {
        return new PagedResponse<>(Collections.emptyList(), houses.getNumber(), houses.getSize(),
            houses.getTotalElements(), houses.getTotalPages(), houses.isLast());
    }

    Map<Long, User> creatorMap = getHouseCreatorMap(houses.getContent());

    List<HouseResponse> houseResponses = houses.map(house -> {
        return ModelMapper.mapHouseToHouseResponse(house, creatorMap.get(house.getCreatedBy()));
    }).getContent();

    return new PagedResponse<>(houseResponses, houses.getNumber(), houses.getSize(),
        houses.getTotalElements(), houses.getTotalPages(), houses.isLast());
}

```

Fig. 3.12 – Returnarea unui răspuns paginat

2.3.1.2. Metode pentru repository

Un alt avantaj al Spring Data JPA este declararea metodelor folosind cuvinte cheie, fără a fi nevoie de implementarea lor explicită. Mai exact, analizând modul în care metoda este numită, Spring creează o interogare către baza de date. Desigur, dacă dorim să facem o interogare mai complicată, avem posibilitatea de a adnota o metodă cu `@Query`(“instrucțiune SQL”), fără alte implementări la nivel de back-end. Lista de cuvinte cheie care pot fi folosite în denumirea metodelor se regăsește în documentația oficială, iar mai jos sunt câteva exemple din aplicația ImmoBorars, în cadrul `HouseRepository`.

```
@Repository
public interface HouseRepository extends JpaRepository<House, Long> {

    Optional<House> findById(Long houseId);

    Page<House> findByCreatedBy(Long userId, Pageable pageable);

    long countByCreatedBy(Long userId);

    Optional<House> findHouseByPictureFiles_IdEquals(String pictureFileId);
}
```

Fig. 3.13 – Metode din cadrul HouseRepository

Operațiile CRUD, totuși, ar fi funcționat chiar și cu o interfață fără nicio metodă declarată, cum am lăsat în cazul lui `DBFileRepository`, unde nu am avut nevoie de altceva în afară de operațiile simple de adăugare, citire, respectiv ștergere.

2.3.2. Securitate

Spuneam anterior de existența unor configurări de securitate, prin care aveam capacitatea de a restricționa accesul anumitor metode doar către unele categorii de utilizatori. Astfel, am împărțit funcționalitățile aplicației în două categorii: cele disponibile public, adică și utilizatorilor neînregistrați, și cele disponibile doar utilizatorilor înregistrați. Pentru a face posibil acest lucru, am implementat un modul de securitate pentru crearea unui cont, folosind JWT (Json Web Token). Cu ajutorul acestuia, endpoint-urile care nu sunt expuse public vor putea fi accesate de utilizatorii înregistrați.

În momentul în care un utilizator se loghează cu succes, serverul de back-end returnează un token format din trei părți: header-ul, care identifică algoritmul de criptare folosit, payload-ul, care conține câteva informații specifice utilizatorului proaspăt logat și semnătura, care validează token-ul. Acesta este salvat de front-end într-un cookie, urmând să adauge o proprietate în header-ul fiecărei solicitări trimise din partea utilizatorului. Această proprietate este de forma Authorization: Bearer, urmată de token.

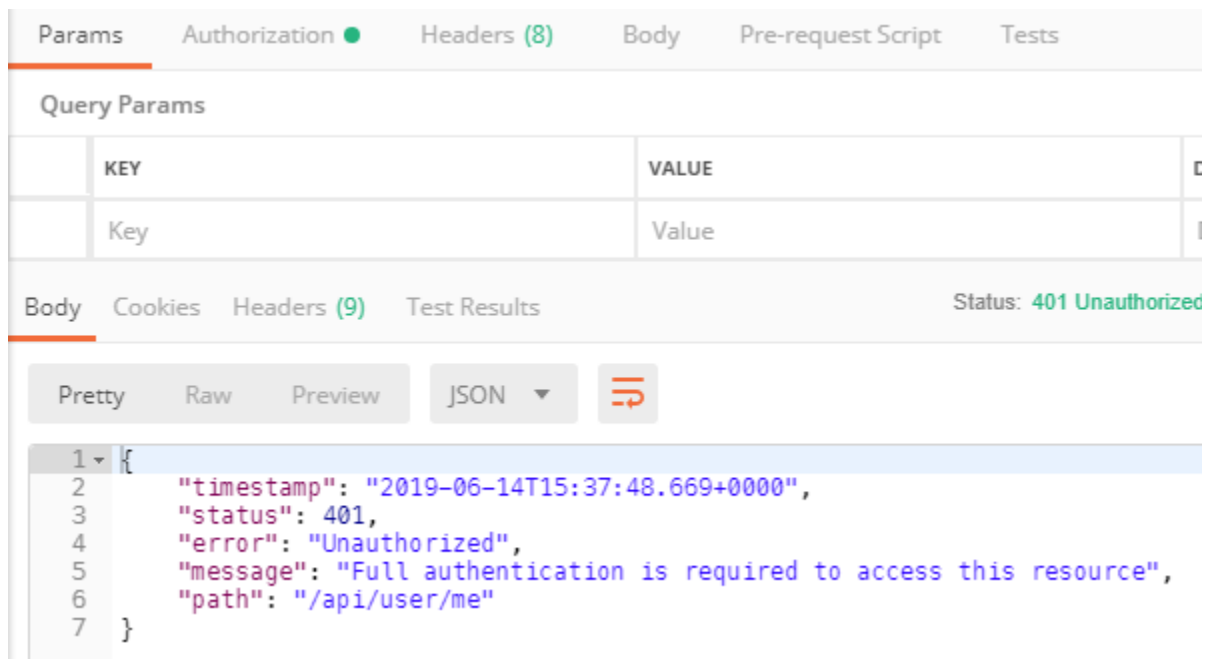


Fig. 3.14 – Răspunsul unei solicitări fără token

2.3.3. Injecție pe baza adnotărilor

O adnotare interesantă din Spring, folosită în proiect, este `@Autowired`. Aceasta permite injectarea unor câmpuri (dependențe) fără a fi nevoie de niciun fel de configurare în prealabil. Poate fi folosită fie în momentul declarării obiectului de injectat, fie înaintea unui setter în care va fi folosit obiectul de injectat.

Întrucât în Spring toate obiectele instanțiate se regăsesc într-un container, numit contextul aplicației, este ușor pentru framework să ofere o instanță a unui obiect pentru un altul care depinde de primul. Astfel, în momentul utilizării adnotării `@Autowired`, aceasta din urmă transmite către contextul aplicației faptul că în acel loc este nevoie de o instanță a obiectului respectiv. Însă, această instanțiere nu este făcută manual, cum am procedat când am creat un obiect de tip casă, de exemplu, ci este făcută de către contextul aplicației.

În cele mai multe cazuri, la fel ca în aplicația *Immoborcars*, vom folosi această metodă pentru a ne folosi de un repository într-un serviciu, sau de un serviciu într-un controller. De precizat este și faptul că instanțele `@Autowired` trebuie să fie adnotate specific în locul lor de definire, de exemplu cu `@Service` sau `@Repository` pentru cazurile anterior menționate.

2.3.4. Logging

Pentru a putea ține o evidență a erorilor din aplicație sau a excepțiilor aruncate în timpul rulării, am ales folosirea unui logger, declarat explicit în fiecare clasă în care dorim să îl folosim. Este importat din pachetul `org.slf4j.Logger` și l-am ales datorită ușurinței în a fi utilizat. Putem loga mesaje de tipul `info`, `debug`, `error` sau `trace`, iar în funcție de cât de în amănunt dorim să urmărim aplicația, putem configura care din cele 4 să fie afișate. De asemenea, când aplicația va ajunge într-un stadiu mai avansat, se recomandă crearea unui fișier special în care să logăm activitatea aplicației, lucru disponibil tot prin configurare.

2.3.5. Debugging

Pe parcursul dezvoltării aplicației, au existat momente în care a fost nevoie de debug, lucru disponibil prin intermediul Spring Boot și IntelliJ. Prin pornirea în modul de debug și atașarea de breakpoint-uri, solicitările trimise fie prin Postman, fie din front-end au putut fi oprite în anumite puncte, urmărind pas cu pas evoluția parametrilor și a metodelor din aplicație.

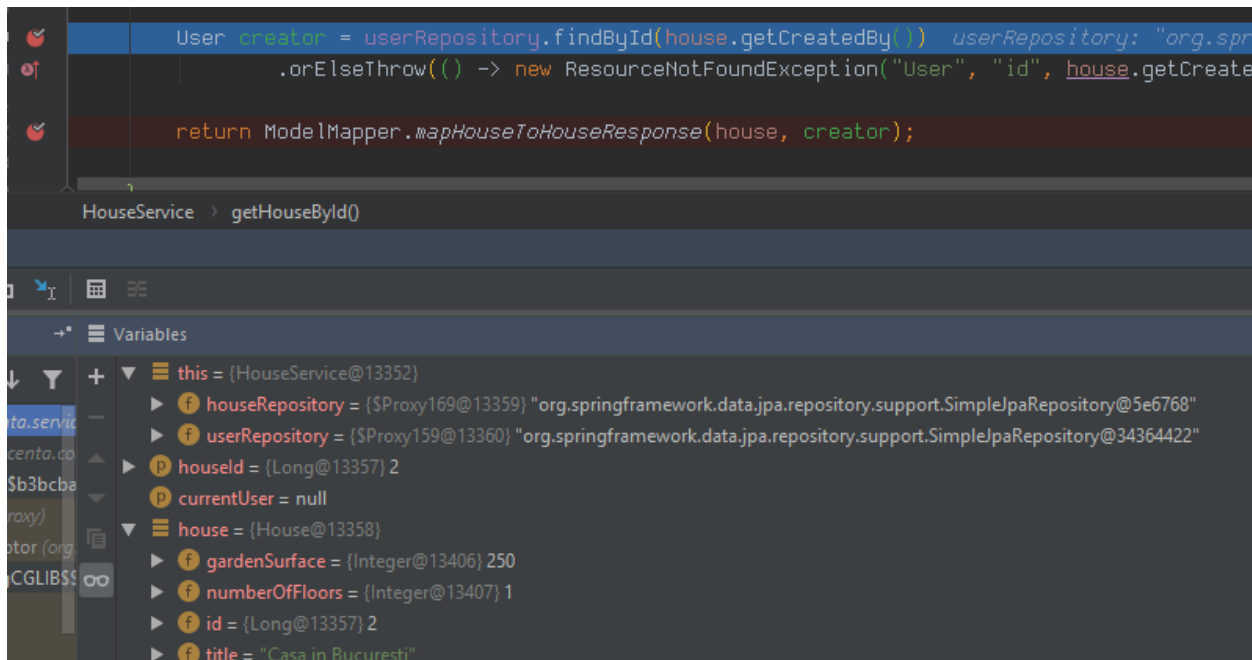


Fig. 3.15 – Fereastră de debug

2.4. Algoritmul Gradient Boosting Regression

Gradient Boosting este un algoritm de machine learning, folosit pentru probleme de regresie și clasificare, ce produce un model de predicție sub forma unor arbori de decizie. Am ales acest algoritm întrucât regresiiile bazate pe support-vector machines (SVM) și k-nearest neighbors (KNN) aveau pierderi foarte mari, din cauza datelor ale căror feature-uri nu erau corelate explicit cu prețurile. Astfel, funcția care definește modelul nu este, neapărat, convexă. Soluțiile menționate

anterior sunt și ele eficiente, însă cel mai mult în cazul în care funcția obiectiv de minimizat este convexă.

Setul de date folosit este preluat de pe Kaggle, o platformă online, deținută de Google, unde se pot găsi sau publica seturi de date pentru domeniile Data science și Machine learning. Cu acest set de date, antrenăm modelul pentru a putea evalua un anunț pe baza unui set de câmpuri. Întrucât setul de date conține un număr prea mare de coloane, vom folosi o metodă pe care am denumit-o `getCols()`, ce va returna numele coloanelor pe care le vom folosi pentru antrenarea modelului. Am făcut acest lucru pentru a obține o estimare corectă folosind doar câmpurile disponibile în entitățile noastre.

```
@app.route("/train-gradient-boosting-regressor")
def trainGradientBoostingRegressor():
    _, X, y = utils.gettingData(path=globalPath,
                                colsX=utils.getCols(), colsLabely=['Label'])

    X_train, _, y_train, _ = train_test_split(X, y, test_size=0.1)

    gbr = GradientBoostingRegressor(n_estimators=10000, learning_rate=0.001,
                                    max_features="sqrt", max_depth=10, loss='lad', verbose=1)
    gbr.fit(X_train, y_train)

    filename = 'gbr_model.sav'
    pickle.dump(gbr, open(filename, 'wb'))
    return jsonify(result="model trained")
```

Fig. 3.16 – Metoda de antrenare a modelului

Hiperparametrii utilizați pentru antrenarea modelului sunt:

- `learning_rate = 0.001`: a fost ales foarte mic pentru a permite modelului să învețe corect. Timpul de rulare a fost mare din această cauză, dar rezultatele semnificativ îmbunătățite.
- `max_features = sqrt`: am aplicat funcția „sqrt” pe vectorul de feature-uri primit pentru optimizarea căutării. Inițial, am folosit \log_2 , dar rezultatele au fost mai bune prin varianta actuală.
- `max_depth = 10`: algoritmul este bazat, după cum am menționat anterior, pe structura arborilor de decizie. Astfel, am ales o adâncime cât mai mare disponibilă în arbore, pentru a fi luate decizii cât mai corecte. Cu cât adâncimea crește, cu atât decizia luată va fi mai bună.

- `loss = lad`: reprezintă funcția `loss` de optimizat și se traduce prin „least absolute deviations”, această metodă practic căutând să minimizeze suma dintre diferențele absolute dintre valoarea `y` de preț și valorile estimate $f(x)$ [7]:

$$S = \sum_{i=1}^n |y_i - f(x_i)|$$

```
@app.route("/use-gradient-boosting-regressor", methods=['POST'])
def useGradientBoostingRegressor():
    to_predict = []
    for col in utils.getCols():
        to_predict += [request.json.get(col)]
    to_predict = np.array(to_predict).reshape(1, -1)

    filename = 'gbr_model.sav'
    gbr = pickle.load(open(filename, 'rb'))
    prediction = gbr.predict(to_predict)[0]

    return jsonify(result=str(prediction))
```

Fig. 3.17 – Metoda de accesare a aprecierii prețului

2.5. React

Unul din principalele motive pentru care am ales React pentru partea de front-end este asemănarea cu limbajele orientate pe obiect. Existența componentelor și actualizarea lor eficientă în paginile web sunt punctele cheie la care React s-a dovedit a fi foarte util în dezvoltarea proiectului. La fel ca în cazul Java, fiind o bibliotecă de JavaScript creată și intens folosită la Facebook, comunitatea online este foarte numeroasă, soluții la eventualele probleme găsindu-se cu ușurință.

Mai mult, componentele create de alte persoane pot fi folosite ca dependențe în proiect, singurul dezavantaj fiind acela că, de fiecare dată când dorim folosirea unei noi componente, trebuie să o instalăm manual prin intermediul unui terminal Node.js, prin comanda „`npm install`”.

2.5.1. Componente

Componentele sunt elemente specifice React, similare funcțiilor din JavaScript, care acceptă ca parametri din partea componentei părinte valori prin intermediul props, însă își pot defini și parametri privați prin intermediul state. Componentele returnează elemente care vor fi, ulterior, transpuse în paginile utilizatorilor.

În cele ce urmează, vor fi date câteva exemple de componente pe care le-am creat și folosit în partea de front-end a aplicației, pentru a ușura procesul de lucru cu datele primite de la back-end.

2.5.1.1. House

Componenta House este folosită în afișarea unei pagini pentru o casă specifică. Aceasta cuprinde cele mai multe informații legate de obiectul unei case, fiind transpusă în momentul în care serverul trimite răspuns în urma unei solicitări pe baza unui ID. Pe lângă afișarea tuturor proprietăților existente în răspuns, componenta se folosește de câmpurile latitudine și longitudine, pe care le transmite ca parametri unei alte componente importate ce va afișa o hartă centrată în locul indicat de coordonatele geografice. În partea de sus a paginii, sunt afișate și pozele ale căror ID-uri sunt stocate pentru fiecare casă, prin construirea în cadrul componentei a link-urilor de acces către fișiere. Acestea sunt date de o adresă comună, la care se concatenează ID-ul fiecărui fișier, după care se trimite câte o solicitare către back-end pentru fiecare poză pentru a le obține din baza de date.

```

render() {
  let images = [];
  for (let i = 0; i < this.state.pictureFileIds.length; i++) {
    images = images.concat({
      original: 'http://localhost:8181/api/files/downloadFile/' + this.state.pictureFileIds[i],
      thumbnail: 'http://localhost:8181/api/files/downloadFile/' + this.state.pictureFileIds[i]
    })
  }

  return (
    <div className="house-container">
      <div className="house-content">
        <div className="house-header">
          <div className="house-title">
            {this.state.title}
          </div>
          <div className="house-price">Price: <strong>{this.state.price}</strong> €</div>
        </div>
        <ImageGallery
          autoplay={true}
          slideInterval={7000}
          slideDuration={450}
          showPlayButton={true}
        />
      </div>
    </div>
  )
}

```

Fig. 3.18 – Limbaj JSX în cadrul componentei House

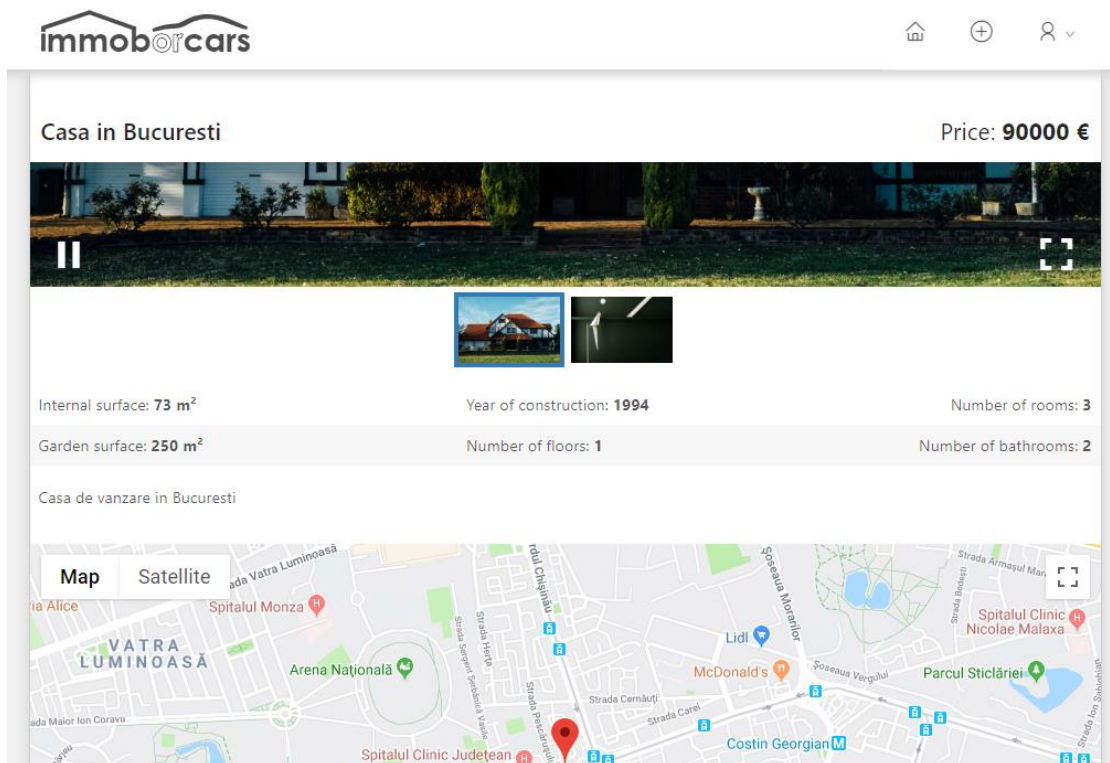


Fig. 3.19 – Secțiune a componentei House

2.5.1.2. HouseBox

Componenta HouseBox este folosită pentru afișarea într-o listă a unei case, afișând doar informațiile importante, lăsând la o parte descrierea, locația pe hartă, respectiv întregul număr de poze încărcate. În colțul din dreapta sus al fiecărei componente va fi afișat un mediu de tip dropdown, doar în cazul în care cel care vizualizează lista este chiar proprietarul anunțului, oferind varianta de a edita sau de a șterge respectiva casă, prin solicitări către server.

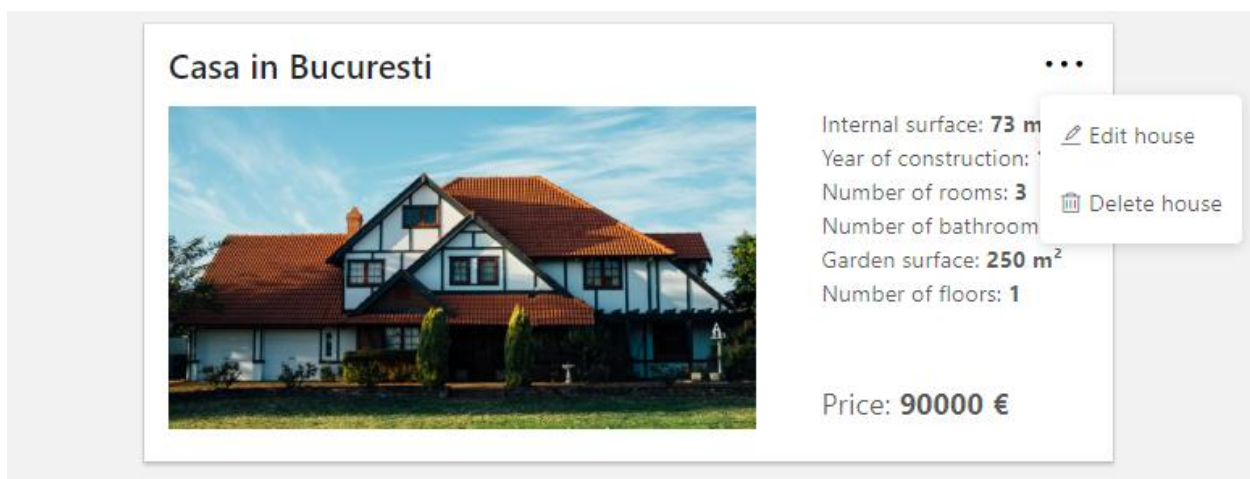


Fig. 3.20 – Componenta HouseBox

2.5.1.3. HouseList

Componenta HouseList este formată dintr-o listă de componente HouseBox, rezultată în urma unei solicitări către server pentru a returna mai multe rezultate, ca de exemplu `getAllHouses(page, size)` sau `getUserCreatedHouses(username, page, size)`. În cazul în care această componentă primește ca parametru, prin intermediul props, un nume de utilizator, este apelată automat a doua metodă exemplificată. În caz contrar, este apelată prima.

2.5.1.4. Notificări

Componenta „notification” din Ant Design ne permite afișarea unor mesaje sub formă de mici ferestre pop-up, pentru care am afișat, pentru moment, dacă logarea s-a efectuat sau nu cu succes, sau dacă au existat alte erori pe parcursul rulării aplicației. De exemplu, la adăugarea unui nou anunț, când serverul răspunde cu un mesaj de eroare, apelăm metoda `notification.error()` cu mesajul erorii, iar dacă este cu succes, apelăm `notification.success()` cu un mesaj specific. Pe viitor, aceste notificări vor putea fi folosite și cu alte scopuri, de exemplu pentru a anunța utilizatorii când unul din anunțurile lor favorite are un preț mai mic.

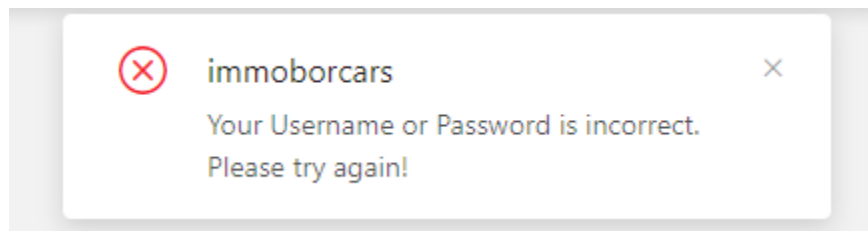


Fig. 3.21 – Notificare în cazul unei erori

2.5.2. Limbajul JSX

Sintaxa prin care React reușește să îmbine componentele, elementele de JavaScript și HTML-ul se numește JSX. Mai exact, JSX reprezintă o extensie la JavaScript, folosită de React pentru a crea elemente, care ulterior vor fi transpuse în pagini.

Prin intermediul său, componentele definite în React pot fi apelate direct în structuri HTML, pentru a fi afișate utilizatorului, având posibilitatea de a le transmite diverși parametri prin „props”, despre care vom vorbi în secțiunea următoare. Apelarea lor se face fie prin delimitatorii „<” și „>” pentru componente, la fel ca în cazul tagurilor HTML, fie prin utilizarea parantezelor de tip acoladă, pentru referențierea variabilelor.

Pentru exemplu, în partea de front-end a aplicației Immoborcars, sintaxa JSX este nelipsită din aproape orice componentă definită. În componenta `HouseList`, după încărcarea caselor în state-ul

componentei, am folosit JSX pentru a apela câte o instanță a componentei HouseBox pentru fiecare casă găsită în state, respectiv pentru a afișa mesajul „No houses found!” în cazul în care nu au fost găsite.

```
render() {
  const houseViews = [];
  this.state.houses.forEach( callbackfn: (house, houseIndex) => {
    houseViews.push(<HouseBox
      currentUser={this.props.currentUser}
      key={house.id}
      house={house} />)
  });

  return (
    <div className="houses-and-search-container">
      {(!this.props.username && <HouseFilterBox/>)}
      <div className="houses-container">
        {houseViews}
        {
          !this.state.isLoading && this.state.houses.length === 0
            ? <div className="no-houses-found">
                <span>No houses found!</span>
              </div>
            : null
        }
      </div>
    </div>
  );
}
```

Fig. 3.22 – Exemplu de utilizare a sintaxei JSX

2.5.3. Prop vs. State

Componentele, la fel ca funcțiile din JavaScript, acceptă ca parametri de inițializare orice câmp trimis de componenta părinte. Toți acești parametri sunt accesibili în componenta de bază prin intermediul referențierii `this.props.numeParametru`. Folosirea acestora se poate observa în Fig. 3.22, unde pentru fiecare componentă HouseBox am trimis utilizatorul curent, ID-ul și obiectul casei, informații folosite pentru a afișa sau prelucra detaliile despre fiecare anunț în parte.

Un alt exemplu al folosirii proprietății de transfer „props”, cu scopul prelucrării și nu al afișării, este în apelarea componentei hărții Google Maps, care primește din baza de date, pentru fiecare casă, două câmpuri de tip double reprezentând latitudinea și longitudinea, componentele folosindu-le mai departe pentru a centra harta și pentru a afișa un punct roșu în centrul ei.

La polul opus, făcând o analogie cu limbajele C++ sau Java, proprietatea state păstrează câmpurile „private” ale unei clase sau componente. Practic, acestea sunt inițializate și prelucrate doar în cadrul repsectivei componente, putând fi, desigur, trimise către altele prin proprietatea props a acelor. Tot în Fig. 3.22 putem observa acest exemplu, unde în apelarea lui HouseBox trimitem câmpul „house”, care este un element al vectorului „this.state.houses”.

Cel mai bun exemplu al folosirii lui state îl putem găsi în componenta HouseList, unde în metoda componentDidMount() – metodă specifică React, apelată la momentul încărcării unei componente – încărcăm toate casele printr-o solicitare către serverul de back-end, urmând a folosi state pentru crearea componentelor HouseBox despre care am vorbit anterior.

2.5.4. Compoziție versus moștenire

Principiul pe care se bazează React este compoziția, fiind metoda recomandată de cei de la Facebook în orice situație în care am avea de ales între cele două. Practic, această metodă reprezintă folosirea unor componente ca parametri în alte componente, pentru a evita moștenirile, care încarcă ierarhia dintre acestea. În documentația oficială, Facebook oferă flexibilitatea ca principal avantaj al compoziției, însă nu oferă nici alte dezavantaje ale folosirii moștenirilor.

```
<GoogleMap
  defaultZoom={15}
  center={{ lat: props.lat, lng: props.lng }}
>
  <Marker position={{ lat: props.lat, lng: props.lng }}/>
</GoogleMap>
```

Fig. 3.23 – Exemplu de folosire a compoziției în React

2.5.5. Rute

Un aspect foarte important pentru dezvoltarea părții de front-end a unei aplicații web este rutarea. Aceasta determină ce componentă va fi încărcată la accesarea unei pagini, putând fi folosită chiar și pentru redirecționarea către o anumită pagină în cazul în care un utilizator neînregistrat dorește să acceseze o pagină pentru care nu are drept de acces. Important de reținut este că ordinea în care sunt declarate rutările cu părți de adresă comune este importantă, fiind necesar să începem cu cele mai specifice, iar cele mai generale rămân la sfârșit. De exemplu, calea „houses/edit/:id” va fi confundată cu „houses/:id” dacă cea din urmă este declarată înainte, aplicându-se principiul expresiilor regulate. În aplicația noastră, de exemplu, pentru calea „/login” am încărcat componenta cu același nume – Login – iar pentru căile de adăugare și editare ale unui anunț, dacă un utilizator nu este logat, am făcut o redirecționare către pagina de login.

2.5.6. Încărcarea de fișiere

Pentru ca un utilizator să poată încărca poze pentru un anunț, am importat o componentă numită Filepond, pentru care au trebuit definite rutele către endpoint-ul de back-end de încărcare sau ștergere de fișiere. Această componentă a avut ca avantaj design-ul prietenos, precum și abilitatea de a configura ușor proprietăți precum numărul maxim de fișiere, previzualizarea imaginilor de încărcat – sau a celor deja încărcate în cazul editării anunțurilor – și afișarea mesajelor de eroare.

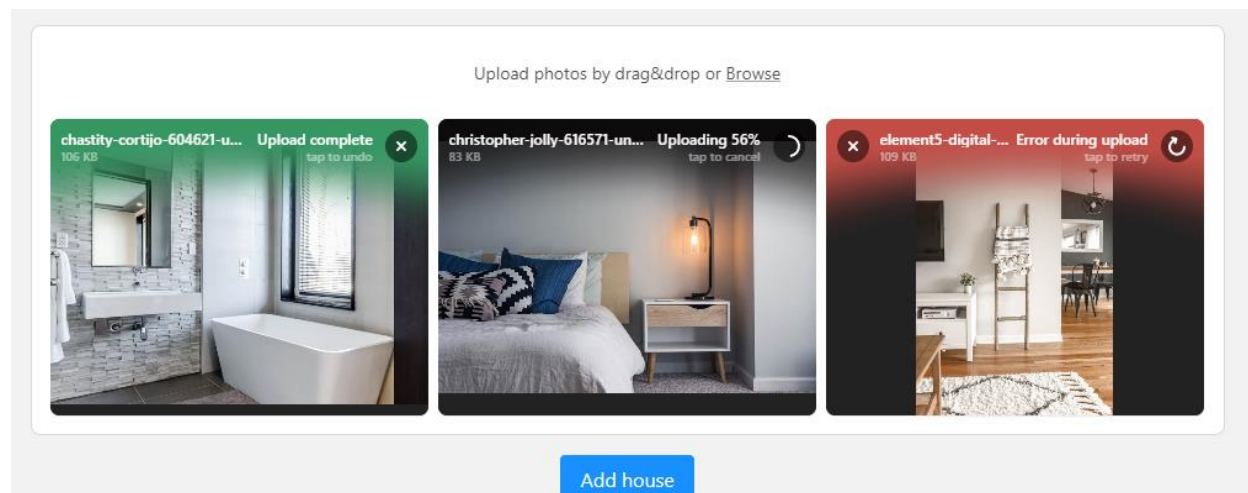


Fig. 3.24 – Componenta Filepond

2.5.7. Iterații async

Unul din lucrurile observate în timpul dezvoltării este faptul că iterațiile folosind „for” sunt asincrone, ceea ce înseamnă că o operație nu este așteptată de celelalte iterații pentru a se finaliza. De cele mai multe ori, acest lucru este benefic pentru timpul de încărcare, însă alteori se pot pierde informații esențiale dintr-o solicitare către back-end, de exemplu.

Pentru a evita o astfel de problemă, am folosit o funcție recursivă, căreia i-am dat ca parametru un index pornind de la 0. Astfel, obținem posibilitatea de a efectua apelul recursiv abia în momentul în care operația apelului curent este finalizată. Tot acest proces se întâmplă în cadrul adăugării pozelor într-un anunț, unde pentru fiecare poză din componenta Filepond, despre care am vorbit anterior, este necesară câte o solicitare către back-end pentru a salva fișierul în baza de date. Cu o iterație de tip „for”, unele solicitări erau trimise simultan, rezultând uneori în erori ce nu erau afișate în componentă.

```
addPicture(index, picturesIds, houseId) {
  if(index < picturesIds.length) {
    addPictureToHouse(houseId, picturesIds[index])
      .then( onfulfilled: response => {
        this.addPicture( index: index+1, picturesIds, houseId);
      }).catch( onrejected: error => {
      });
  }
}
```

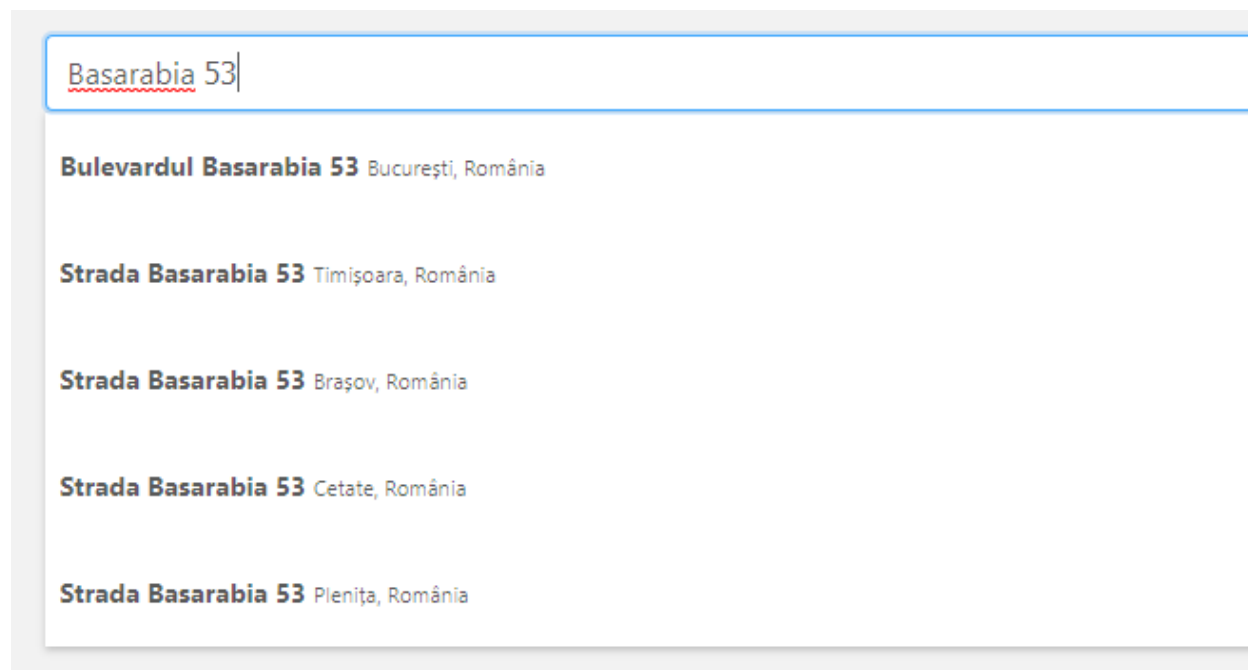
Fig. 3.25 – Metodă recursivă de adăugare a unor poze

După cum putem observa, apelul recursiv este realizat abia în momentul în care răspunsul este primit din partea serverului, componenta Filepond putând seta mesajele de succes sau eroare pentru fiecare poză, ca în Fig. 3.24.

2.5.8. Google Maps API

Pentru a putea oferi utilizatorilor o experiență de utilizare cât mai plăcută, am ales folosirea API-ului de la Google pentru introducerea de adrese pentru fiecare anunț. Astfel, în componenta de adăugare a unui anunț, am adăugat câmpul „Adresă”, care oferă sugestii preluate din câmpul de căutare din Google Maps, folosind API-ul Places. Pe baza acestuia sunt stocate în baza de date, pentru fiecare anunț, câmpurile de tip double latitudine și longitudine, care vor fi folosite ulterior pentru afișarea hărții prin API-ul Maps JavaScript.

Cheile necesare pentru accesarea API-urilor Google se obțin prin platforma console.cloud.google.com, unde se înregistrează proiectul și se pot face limitări în funcție de IP-ul solicitării, pentru a împiedica folosirea cheii de către alte persoane.



The image shows a web form with a text input field containing the text "Basarabia 53". Below the input field, a dropdown menu is open, displaying five suggestions. Each suggestion consists of a street name followed by a city and country. The suggestions are: "Bulevardul Basarabia 53 București, România", "Strada Basarabia 53 Timișoara, România", "Strada Basarabia 53 Brașov, România", "Strada Basarabia 53 Cetate, România", and "Strada Basarabia 53 Plenița, România". The first suggestion, "Bulevardul Basarabia 53 București, România", is highlighted with a blue background.

Suggestion
Bulevardul Basarabia 53 București, România
Strada Basarabia 53 Timișoara, România
Strada Basarabia 53 Brașov, România
Strada Basarabia 53 Cetate, România
Strada Basarabia 53 Plenița, România

Fig. 3.26 – Câmpul de adresă din formularul de adăugare sau editare

În platforma `console.cloud.google.com`, avem acces la diferite informații privind numărul de solicitări disponibile rămase, traficul generat folosind cheile API pentru aplicația noastră, cât și procentajul de erori din timpul solicitărilor.

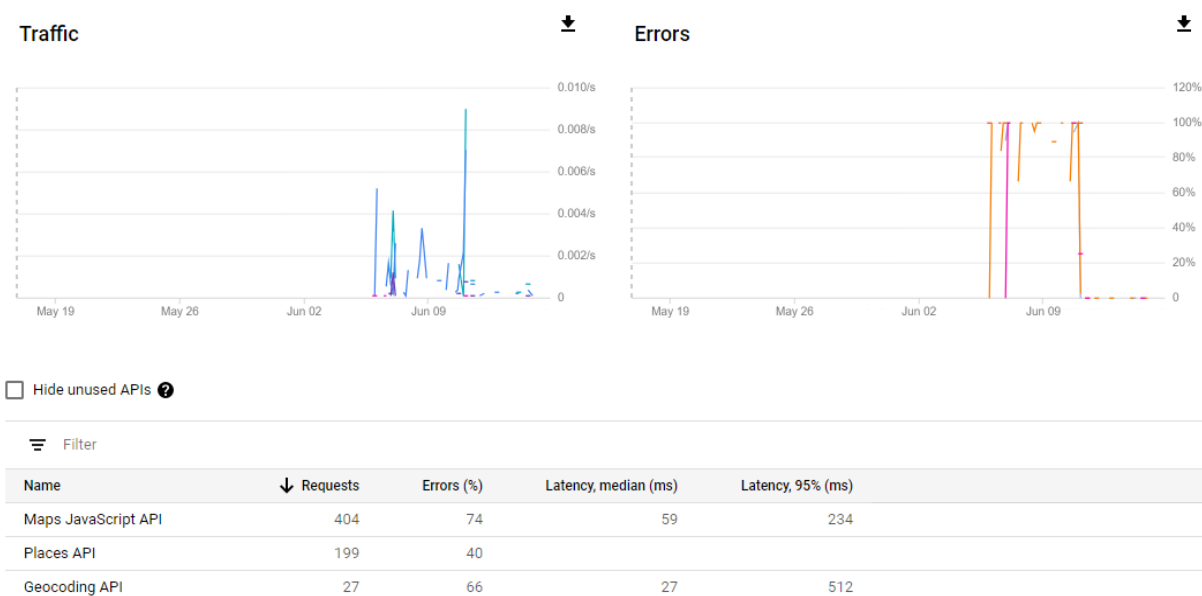


Fig. 3.27 – Tabloul de bord pentru cheia Google API folosită

2.6. Design Patterns

2.6.1. Builder

În implementarea proiectului, am folosit metoda de design pattern Builder, cu ajutorul adnotării Lombok `@Builder`. Scopul acesteia este de a separa construirea unui obiect complex de reprezentarea sa, astfel încât, folosind același proces de construire, un obiect poate fi creat cu diferite reprezentări. Practic, procesul creării unor reprezentări diferite ale unui obiect complex ar trebui să fie același.

Acest concept se bazează pe apelarea metodelor de set pentru fiecare câmp în parte, astfel încât putem construi obiecte cu orice număr de câmpuri dorim prin aceeași metodă, fără a avea nevoie de constructori separați pentru fiecare caz. Numărul de linii de cod se reduce semnificativ, chiar și fără ajutorul lui Lombok.

```
return HouseResponse.builder()
    .id(house.getId())
    .title(house.getTitle())
    .description(house.getDescription())
    .price(house.getPrice())
    .pictureFileIds(house.getPictureFiles().stream().map(DBFile::getId).collect(Collectors.toList()))
    .internalSurface(house.getInternalSurface())
    .yearOfConstruction(house.getYearOfConstruction())
    .numberOfRooms(house.getNumberOfRooms())
    .numberOfBathrooms(house.getNumberOfBathrooms())
    .gardenSurface(house.getGardenSurface())
    .numberOfFloors(house.getNumberOfFloors())
    .createdBy(creatorSummary)
    .creationDateTime(house.getCreatedAt())
    .latitude(house.getLatitude())
    .longitude(house.getLongitude())
    .build();
```

Fig. 3.28 – Folosirea unui builder în cadrul aplicației

În figura de mai sus, am folosit builder-ul pentru a crea un obiect de tip HouseResponse, în vederea returnării sale de către un endpoint. Cu aceeași implementare a acestui design pattern, fie putem folosi toate câmpurile de mai sus în orice ordine dorim, fie putem folosi doar o selecție de câmpuri, în funcție de necesități.

Capitolul III – Prezentarea aplicației

În acest capitol, urmează prezentarea propriu-zisă a părților principale ale aplicației, urmărind fluxul paginilor și al funcționalităților.

3.1. Ecranul principal

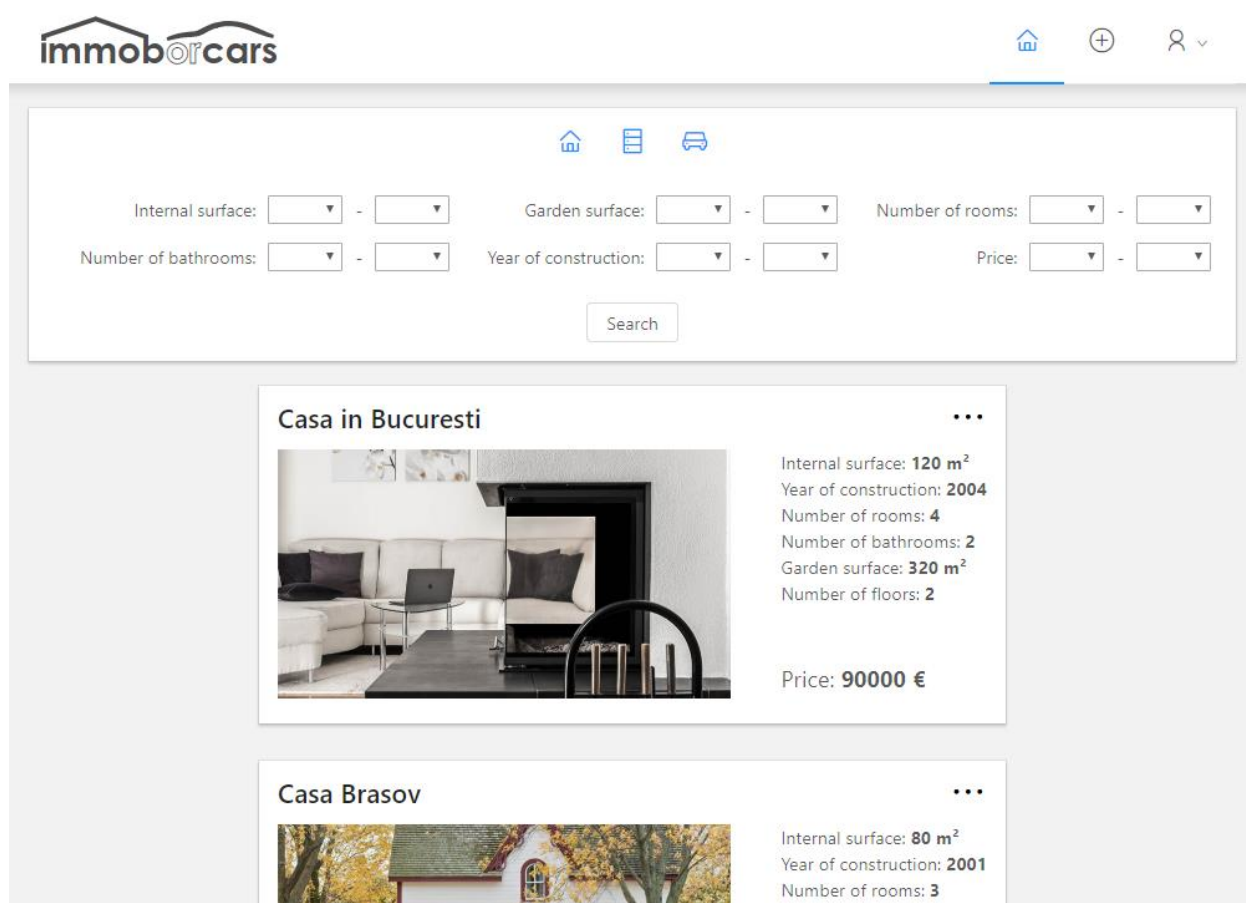
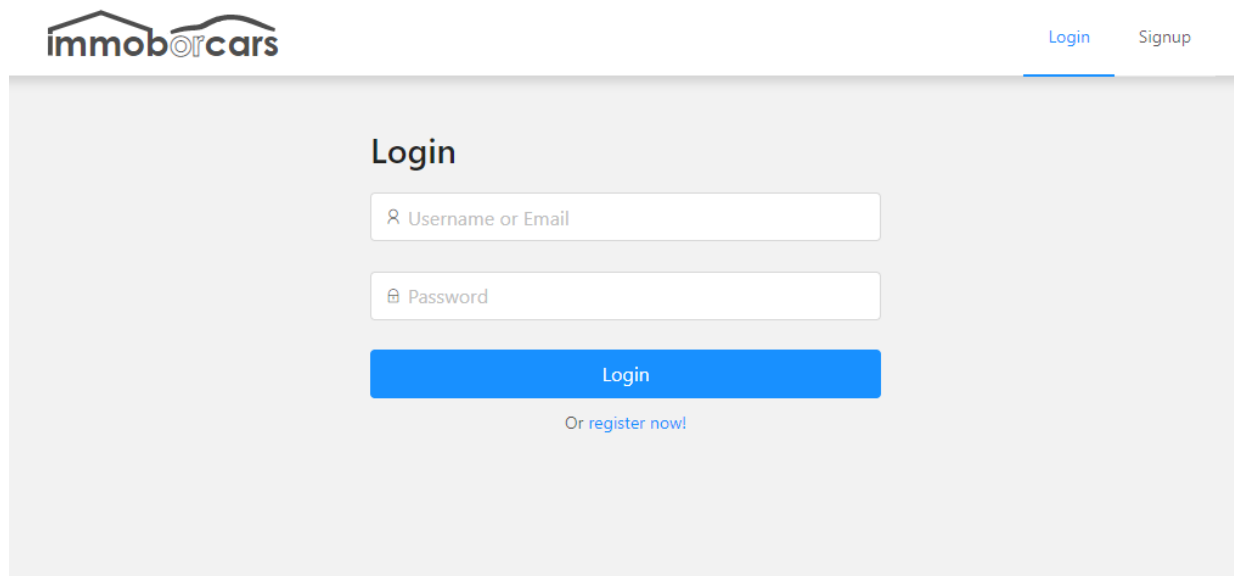


Fig. 4.1 – Ecranul principal al unui utilizator înregistrat

În Fig. 4.1 este ilustrată pagina principală a unui utilizator înregistrat, ce conține câmpurile de filtrare și lista de anunțuri în ordine descrescătoare a datei de adăugare. Fiecare anunț al cărui proprietar este utilizatorul curent conține un buton, în colțul din dreapta-sus, de unde se poate edita

sau șterge anunțul. Din acest punct, utilizatorul are acces la paginile de creare a unor noi anunțuri, de filtrare a anunțurilor, de accesare a profilului și de delogare.

3.2. Pagina de login



immobORcars

Login Signup

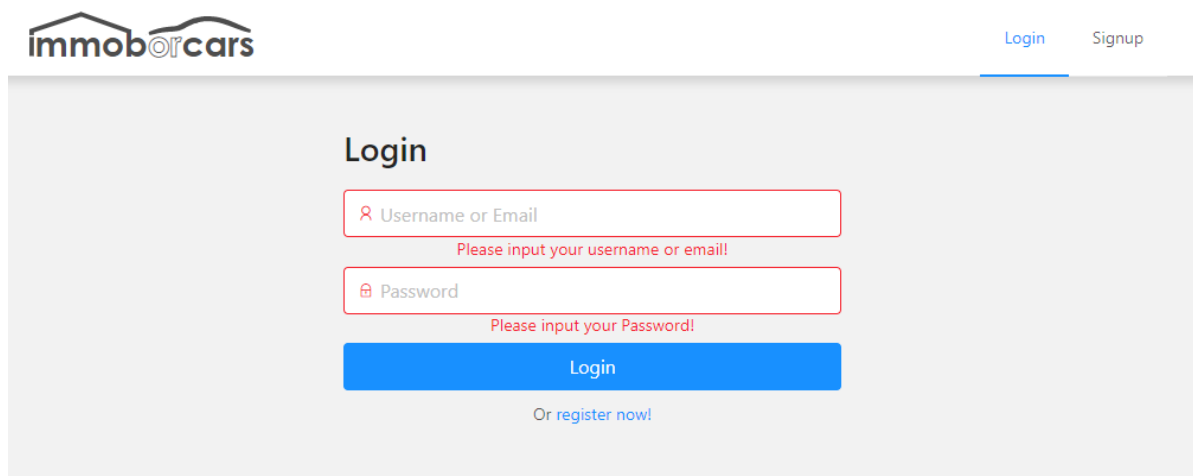
Login

Login

[Or register now!](#)

Fig. 4.2 – Pagina de login

Pagina de login conține cele două câmpuri, pentru numele de utilizator sau email, respectiv parolă. Fiecare dintre acestea conține validări de completare, iar din acest punct putem accesa pagina principală, printr-un click pe logo, sau pagina de înregistrare, prin click pe butonul Signup sau pe „register now”.



immobORcars

Login Signup

Login

Please input your username or email!

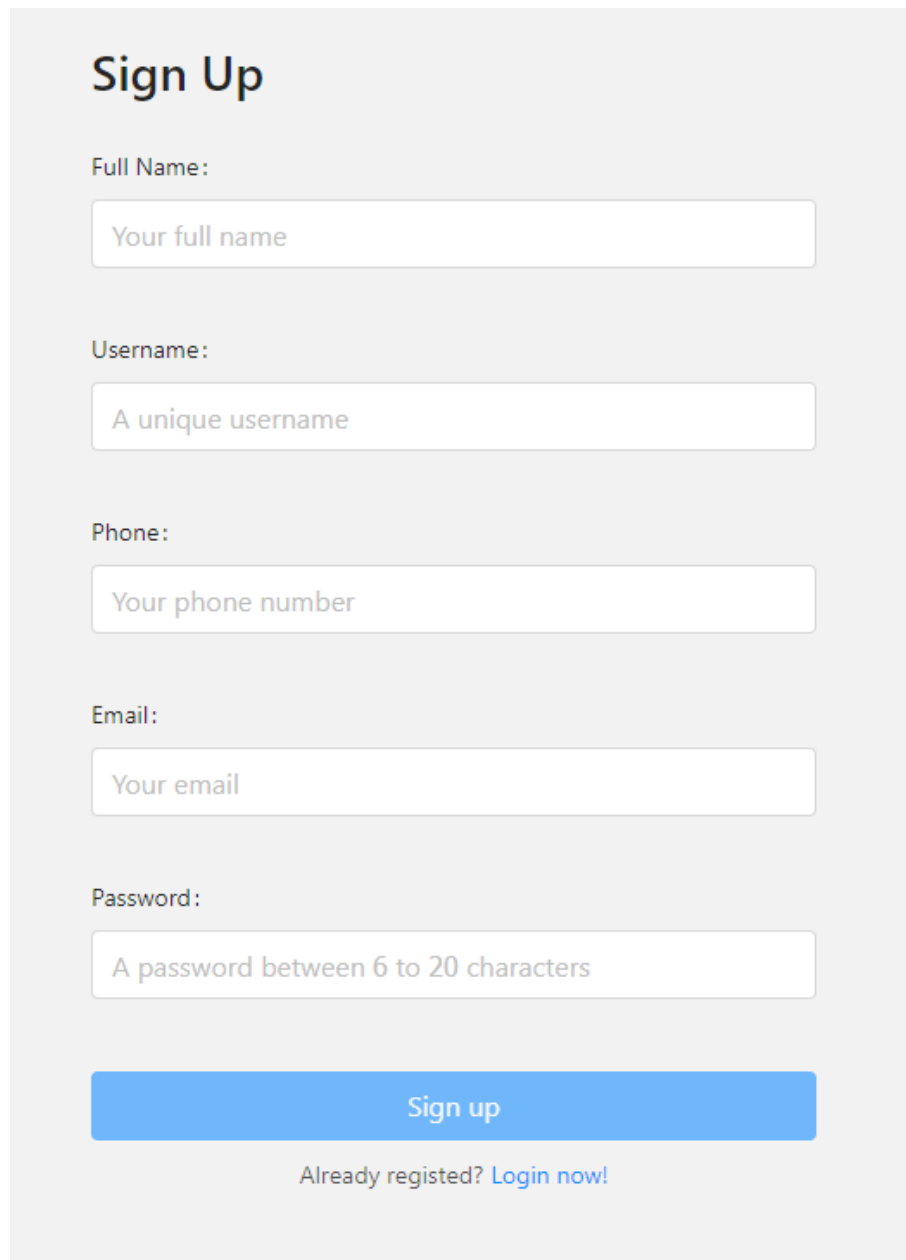
Please input your Password!

Login

[Or register now!](#)

Fig. 4.3 – Validări pentru câmpuri

3.3. Pagina de înregistrare

A registration form titled "Sign Up" with a light gray background. It contains five input fields, each with a label above it: "Full Name:" with placeholder "Your full name", "Username:" with placeholder "A unique username", "Phone:" with placeholder "Your phone number", "Email:" with placeholder "Your email", and "Password:" with placeholder "A password between 6 to 20 characters". Below the fields is a blue "Sign up" button. At the bottom, there is a link that says "Already registered? Login now!".

Sign Up

Full Name:

Your full name

Username:

A unique username

Phone:

Your phone number

Email:

Your email

Password:

A password between 6 to 20 characters

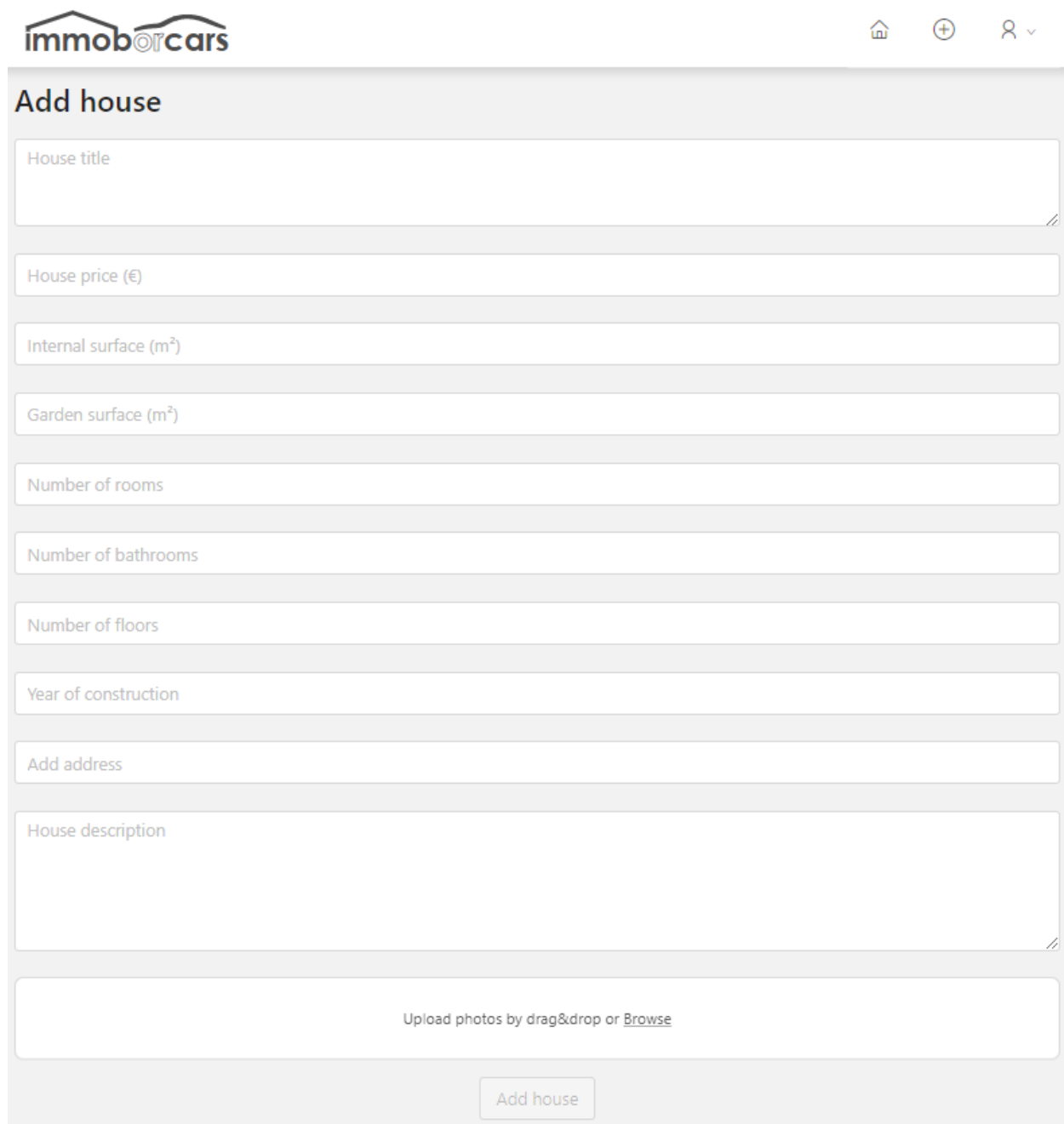
Sign up

Already registered? [Login now!](#)

Fig. 4.4 – Formularul de înregistrare

Formularul de înregistrare (Fig. 4.4) conține cinci câmpuri, toate obligatorii, constând în numele întreg, numele de utilizator, numărul de telefon, email-ul și parola.

3.4. Formularul de adăugare al unui anunț




The screenshot shows the 'Add house' form on the immoborcars website. The form is titled 'Add house' and contains several input fields for property details. At the top right, there are navigation icons: a home icon, a plus icon, and a user profile icon with a dropdown arrow. The form fields are as follows:




- House title
- House price (€)
- Internal surface (m²)
- Garden surface (m²)
- Number of rooms
- Number of bathrooms
- Number of floors
- Year of construction
- Add address
- House description
- Upload photos by drag&drop or [Browse](#)


At the bottom of the form is a button labeled 'Add house'.

Fig. 4.5 – Formularul de adăugare al unui anunț

3.5. Pagina de profil a unui utilizator








Rares Opariuc
@rares.opariuc
Joined March 2019

2 Houses

0 Apartments

0 Cars


Casa in Bucuresti



Internal surface: **120 m²**
Year of construction: **2004**
Number of rooms: **4**
Number of bathrooms: **2**
Garden surface: **320 m²**
Number of floors: **2**

Price: **90000 €**

Casa Brasov



Internal surface: **80 m²**
Year of construction: **2001**
Number of rooms: **3**
Number of bathrooms: **2**
Garden surface: **250 m²**
Number of floors: **1**

Price: **65500 €**

Fig. 4.6 – Pagina de profil

Pagina de profil a utilizatorului oferă informații cu privire la numele întreg, numele de utilizator și data de înregistrare a acestuia, precum și câte o listă de anunțuri adăugate pentru fiecare categorie disponibilă în aplicație.

Concluzii

Fiind în curs de dezvoltare, aplicația la care am lucrat încă are foarte multe lucruri de oferit. Ideea inițială, de a oferi utilizatorilor o platformă în care să poată obține aprecieri ale prețurilor reale ale unui anunț pentru cele mai de valoare lucruri cumpărate de-a lungul vieții, s-a concretizat în ceea ce Immoborcars poate oferi în prezent. Desigur, succesul unei aplicații necesită efortul continuu al unei echipe de oameni bine pregătiți și motivați, însă, în opinia mea, această aplicație are potențialul necesar pentru a deveni un bun competitor al soluțiilor existente în prezent în mediul online. În cele ce urmează, voi prezenta câteva îmbunătățiri pe care le-am putea aduce, pentru început, aplicației.

Adăugarea unui calculator de rate pentru diferite instituții bancare din țara noastră i-ar ajuta pe utilizatori să își facă o idee despre un potențial credit pentru imobilul sau automobilul visat. Acesta nu este greu de implementat, însă dificultatea se transpune în menținerea ofertelor la zi a portofoliilor băncilor, respectiv al metodelor de calcul pentru creditele sau serviciile oferite.

Ulterior, păstrarea în baza de date a căutărilor fiecărui utilizator, mai exact a filtrelor aplicate cel mai des, pentru a îi putea oferi recomandări când noi anunțuri, care se potrivesc cerințelor sale, sunt adăugate.

De asemenea, m-am gândit și la crearea unei pagini dedicate dezvoltatorilor sau dealerilor, care vor putea face oferte personalizate pentru noile imobile pe care le construiesc sau pentru noile automobile pe care le primesc în stoc.

Bibliografie

- [1] OCA/OCP Java SE 8 Programmer Certification Kit, Jeanne Boyarsky, Scott Selikoff, 1152 pagini, 2016
- [2] Java in a Nutshell, 7th Edition, O'Reilly Media, 456 pagini, 2018
- [3] Maven - <https://books.sonatype.com/mvnex-book/reference/index.html>
- [4] Spring Boot - <https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone/>
- [5] Spring Data JPA - <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
- [6] ReactJS - <https://reactjs.org/docs/getting-started.html>
- [7] Algoritmi machine learning - <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html>