



UNIVERSITATEA DIN
BUCUREŞTI



FACULTATEA DE
MATEMATICA ŞI
INFORMATICA

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

DRIVER'S HUB

Absolvent
Gheorghe Radu-Mihai

Coordonator științific
Conf. Univ. Dr. Boriga Radu-Eugen

Bucureşti, iunie 2024

Rezumat

Într-o lume în continuă mișcare, fiecare șofer se confruntă cu provocările gestionării eficiente a tuturor aspectelor legate de vehiculul său. De la întreținerea regulată la situațiile neprevăzute, gestionarea acestor responsabilități poate deveni o sursă semnificativă de stres și neplăceri.

Driver's Hub este o aplicație mobilă cross-platform ce răspunde acestor nevoi prin oferirea unei platforme centralizate care simplifică și eficientizează aceste procese. Prin monitorizarea eficientă a datelor de întreținere a vehiculului în timp real și integrând algoritmi de **inteligentă artificială** pentru predicția prețurilor și diagnosticarea rapidă a problemelor vehiculului, dar și oferind funcții pentru marcarea locurilor de parcare și localizarea rapidă a service-urilor auto din proximitate, Driver's Hub devine un companion de ajutor pentru fiecare șofer.

Abstract

In a world that is constantly in motion, every driver faces the challenges of efficiently managing all aspects related to their vehicle. From regular maintenance to unforeseen situations, managing these responsibilities can become a significant source of stress and inconvenience.

Driver's Hub is a cross-platform mobile application that addresses these needs by offering a centralized platform that simplifies and streamlines these processes. Through effective monitoring of vehicle maintenance data in real time and integrating **artificial intelligence** algorithms for price prediction and rapid diagnosis of vehicle problems, as well as providing functions for marking parking spots and quickly locating nearby auto services, Driver's Hub becomes a helpful companion for every driver.

Cuprins

1 Introducere	6
1.1 Motivație	6
1.2 Preliminarii	6
1.2.1 Gestionarea informațiilor despre utilitățile mașinii și trimiterea notificărilor înainte de expirare	7
1.2.2 Detectarea și recunoașterea indicatorilor de probleme din bord folosind inteligență artificială	7
1.2.3 Estimarea prețului mașinii pe piața second-hand folosind inteligență artificială	7
1.2.4 ChatBot integrat pentru o identificare mai bună a problemelor mașinii folosind API-ul OpenAI	8
1.2.5 Localizarea service-urilor auto din proximitate, alături de marcarea locului de parcare folosind harta	8
1.3 Aplicații similare	8
1.3.1 Movcar	8
1.3.2 My Car	8
1.4 Avantajele „Driver’s Hub”	9
1.5 Structura lucrării	9
2 Tehnologii folosite	11
2.1 Node.js	11
2.2 React Native	12
2.3 PostgreSQL	12
2.4 Framework-uri	12
2.4.1 Express.js	12
2.4.2 Expo	12
2.5 Amazon Web Services	13
2.5.1 Elastic Beanstalk	13
2.5.2 Relational Database Service	13
2.5.3 S3 (Simple Storage Service)	14

2.6	Google Cloud	14
2.6.1	Maps JavaScript API	14
2.6.2	Places API	15
2.6.3	Distance Matrix API	15
2.6.4	Directions API	15
2.7	XGBoost și YOLOv5	15
2.8	Altele	16
2.8.1	OpenAI API	16
3	Arhitectura aplicației	17
3.1	Backend	17
3.1.1	Baza de date	17
3.1.2	Controllers	18
3.1.3	Rutele pentru API-uri	19
3.1.4	Middlewares	20
3.1.5	Mecanismul de verificare și expirare al documentelor	22
3.1.6	Trimiteerea notificărilor	24
3.1.7	Gestionarea erorilor	26
3.2	Frontend	26
3.2.1	Componente	26
3.2.2	Ecrane	32
3.2.3	Funcționalitățile bazate pe hartă	37
3.2.4	Comunicarea cu backend-ul	39
3.3	Machine Learning	40
3.3.1	Estimarea prețului de revânzare al mașinilor	40
3.3.2	Detectarea și recunoașterea indicatorilor din bord	42
3.3.3	Integrarea modelelor în backend	45
3.4	Cloud	47
4	Prezentarea aplicației	49
4.1	Pagina de înregistrare și logare	49
4.2	Panoul de control	50
4.2.1	Bara de navigare	50
4.2.2	Adăugarea unui document	50
4.2.3	Estimarea prețului second-hand al mașinii	51
4.3	Pagina pentru detecția simbolurilor din bord	52
4.4	Adăugarea unei noi mașini	52
4.5	Chatbot-ul pentru depistarea problemelor mașinii	53
4.6	Pagina pentru funcționalitățile ce folosesc locația	53
4.7	Sectiunea de profil	54

4.8 Notificările	55
5 Concluzii	57
Bibliografie	59

Capitolul 1

Introducere

1.1 Motivație

Motivația din spatele dezvoltării aplicației Driver's Hub a constat în faptul că posesorii de vehicule, inclusiv eu, deseori se confruntă cu dificultăți în organizarea și urmărirea aspectelor esențiale legate de întreținerea mașinilor lor. Problemele frecvente precum uitarea termenelor de reînnoire pentru asigurări sau inspecții tehnice pot duce la neplăceri cauzate sau chiar la penalizări legale. În plus, problemele neașteptate ce pot apărea la mașină sunt de multe ori greu de identificat de către șoferul obișnuit, iar găsirea unui service auto de încredere în apropiere sunt alte două aspecte care adesea complică experiența de șofer.

Aceste observații, coroborate cu pasiunea pentru tehnologie și dorința de a îmbunătăți corelația dintre tehnologie și activitățile zilnice, m-au condus la ideea aplicației Driver's Hub. Prin integrarea tehnologiilor de inteligență artificială pentru estimarea prețurilor și diagnosticarea problemelor, aplicația nu doar că simplifică gestionarea vehiculelor, ci transformă și modul în care șoferii interacționează cu mașinile lor, făcând experiența acestora mai plăcută.

1.2 Preliminarii

Obiectivele aplicației sunt realizate prin următoarele funcționalități:

- Gestionarea informațiilor despre utilitățile mașinii și trimiterea notificărilor înainte de expirare;
- Detectarea și recunoașterea indicatorilor de probleme din bord folosind inteligență artificială;
- Estimarea prețului mașinii pe piața second-hand folosind inteligență artificială;

- ChatBot integrat pentru o identificare mai bună a problemelor mașinii folosind API-ul¹ OpenAI;
- Localizarea service-urilor auto din proximitate, alături de marcarea locului de parcare folosind harta.

1.2.1 Gestionarea informațiilor despre utilitățile mașinii și trimiterea notificărilor înainte de expirare

Prin intermediul unor formulare, utilizatorul poate introduce, pentru fiecare mașină, detalii relevante despre utilitățile acesteia, precum asigurarea obligatorie, inspecția tehnică periodică și reviziile. În funcție de tipul ales, câmpurile sunt diferite, spre exemplu pentru asigurare câmpurile cerute sunt firma de asigurări la care este făcută asigurarea, numărul poliței de asigurare, valabilitatea acesteia și o poză cu aceasta, în timp ce pentru inspecția tehnică periodică este importantă valabilitatea acesteia.

Odată introduse datele necesare, utilizatorul poate vedea în panoul de control al aplicației aceste date sub forma unor *widget-uri* unde este evidențiată valabilitatea acestora.

De asemenea, pe măsura ce data de expirare a acestor documente se apropie, utilizatorul va primi câte o notificare pentru fiecare document, cu 7 zile, 3 zile, respectiv o zi înainte de expirarea acestora, dar și în ziua respectivă.

1.2.2 Detectarea și recunoașterea indicatorilor de probleme din bord folosind inteligență artificială

Utilizatorii au posibilitatea să încarce o fotografie sau să facă una direct prin aplicație, care apoi este procesată folosind inteligență artificială. Aplicația identifică indicatorii de bord prezenti în imagine și îi încadrează în chenare. De asemenea, utilizatorii primesc explicații text pentru fiecare indicator detectat, oferindu-le informații despre ce reprezintă fiecare martor de bord și posibilele acțiuni recomandate.

1.2.3 Estimarea prețului mașinii pe piața second-hand folosind inteligență artificială

Utilizatorii pot beneficia de funcționalitatea de estimare a prețului mașinii pe piața second-hand, care utilizează inteligență artificială pentru a oferi o valoare aproximativă a vehiculului, pe baza mărcii, a modelului, a anului fabricației și al kilometrajului mașinii. Rezultatul este o estimare a prețului, care poate ajuta proprietarii de vehicule să înțeleagă mai bine valoarea de piață a mașinilor lor.

¹API = Application Program Interface

1.2.4 ChatBot integrat pentru o identificare mai bună a problemelor mașinii folosind API-ul OpenAI

Aplicația include un ChatBot integrat cu ajutorul API-ului OpenAI pentru a ajuta utilizatorii să identifice problemele mașinii. Utilizatorii pot interacționa direct cu ChatBot-ul, descriind simptomele sau problemele întâmpinate, iar ChatBot-ul procesează aceste informații și oferă diagnoze preliminare și recomandări. Această funcționalitate ajută utilizatorii să își contureze o idee mai clară despre problemele mașinii și despre pașii ce trebuie abordati pentru soluționarea lor.

1.2.5 Localizarea service-urilor auto din proximitate, alături de marcarea locului de parcare folosind harta

Prin intermediul hărții integrate, utilizatorii pot identifica service-uri auto din apropiere. La selectarea unui service de pe hartă, aplicația afișează timpul estimat necesar pentru a ajunge la destinație, folosind API-ul celor de la Google.

Utilizatorul poate, de asemenea, să marcheze pe hartă locul de parcare, pentru ca apoi să fie afișate și direcțiile de navigație către acesta, funcționalitate utilă turistilor ce vizitează cu mașina și nu cunosc împrejurimile.

1.3 Aplicații similare

1.3.1 Movcar

Movcar este o platformă online, care vine și sub forma de aplicație mobile, atât pentru iOS, cât și pentru Android, ale cărei funcționalități sunt concentrate pe menținerea evidenței utilităților mașinii. Pe lângă aceste funcționalități, aplicația oferă și posibilitatea de a obține oferte de asigurare și de a plăti parcare și alte taxe direct din aceasta, dar și posibilitatea verificării istoricului mașinilor folosind CarVertical².

1.3.2 My Car

My Car este o aplicație mobile cross-platform în care utilizatorii pot gestiona aspecte ale mașinilor lor, cu accent pus în special pe partea ce ține de cheltuieli și buget. Spre deosebire de celelalte aplicații, în această aplicație se pot adăuga doar detalii despre vizitele la service, fără a putea introduce detalii despre utilități care urmează să expire. În plus, în aplicație se găsesc diverse grafice, chart-uri și pie-uri pe baza cheltuielilor introduse.

²CarVertical = platformă unde se poate verifica istoricul unei mașini

1.4 Avantajele „Driver’s Hub”

Driver’s Hub își propune să devină asistentul fiecărui șofer, îmbinând cât mai multe funcționalități diverse din spectrul celor de care ar avea nevoie un șofer. Pe lângă gestionarea utilităților mașinii, un avantaj al aplicației este folosirea inteligenței artificiale pentru a diagnostica probleme ale mașinii prin analiza vizuală a indicatorilor din bord.

Un alt avantaj al aplicației este funcționalitatea de estimare a prețului mașinilor pe piața second-hand, care oferă utilizatorilor evaluări precise bazate pe datele pieței actuale, oferindu-le acestora o idee mai clară în cazul dorinței de a-și vinde vehiculul.

Driver’s Hub include și funcții de localizare a service-urilor auto și a locurilor de parcare, oferind direcții și estimări ale timpului de conducere pentru a facilita accesul la aceste locații.

De asemenea, ChatBot-ul integrat cu ajutorul OpenAI este de mare ajutor și servește utilizatorilor pentru situațiile când întâmpină probleme cu mașina, astfel utilizatorii pot purta discuții libere cu asistentul până când consideră că problema a fost înțeleasă.

Interfața aplicației este simplă și intuitivă, feature-urile fiind independente unul de celălalt, separate clar de către opțiunile din bara de navigație a aplicației.

1.5 Structura lucrării

Această lucrare detaliază dezvoltarea și funcționalitatea aplicației „Driver’s Hub”, începând cu o introducere care subliniază utilitatea și scopul aplicației. Următoarele capitole vor aborda în profunzime aspectele tehnice și structurale ale aplicației, împreună cu o demonstrație a funcționalității sale.

Capitolul 2 va explora tehnologiile utilizate în crearea aplicației, acoperind atât frontend-ul, cât și backend-ul. Vor fi aduse detalii despre **Node.js** și **Express** pentru partea de server, iar pentru partea de client se vor introduce **React Native** și **Expo**, prin intermediul căroror a fost creată interfața aplicației. Acest capitol va introduce, de asemenea, tehnologiile specifice folosite pentru stocarea datelor, autentificarea utilizatorilor și integrarea cu alte servicii externe de cloud, **Amazon Web Services** și **Google Cloud**.

Capitolul 3 va oferi o viziune detaliată asupra arhitecturii aplicației, împărțind discuția în patru secțiuni principale: **backend**, **frontend**, **machine learning** și **cloud**. În secțiunea de backend, sunt prezentate structura bazei de date, controllerele, rutele și middleware-urile, precum și mecanismul de gestionare al notificărilor. Frontend-ul va fi explorat prin componente, ecrane și funcționalități specifice, cum ar fi integrarea hărții și a chatbot-ului, plus modul de comunicare cu backend-ul. De asemenea, vor fi descriși și algoritmii de învățare automată și desfășurarea infrastructurii în cloud folosind AWS Elastic Beanstalk, AWS RDS și AWS S3.

Capitolul 4 va prezenta aplicația din perspectiva utilizatorului, demonstrând funcționalitatea prin capturi de ecran și descrieri detaliate ale interacțiunii cu aplicația. Acest capitol va evidenția cum fiecare componentă a designului contribuie la experiența generală a utilizatorului.

În final, **capitolul 5** va oferi concluzii despre întregul proces de dezvoltare, subliniind provocările întâmpinate și experiența dobândită.

Capitolul 2

Tehnologii folosite

2.1 Node.js

Node.js este un mediu de execuție open source pentru JavaScript, creat în 2009, care permite rularea codului JavaScript pe partea de server. Este construit pe motorul V8 JavaScript Engine de la Google, ceea ce îi conferă viteza și performanța necesare pentru a procesa operațiuni de intrare/ieșire asincrone și evenimente non-blocante.

Node.js este folosit pentru a dezvolta aplicații web scalabile, datorită arhitecturii sale bazate pe evenimente și a naturii sale non-blocante, care permite gestionarea simultană a mai multor conexiuni. Acest mediu este compatibil cu majoritatea sistemelor de operare, inclusiv Linux, macOS, și Windows, ceea ce îl face o alegere populară pentru dezvoltarea cross-platform [6].

Împreună cu npm¹, cea mai mare bibliotecă de pachete open source, Node.js permite dezvoltatorilor să folosească și să partajeze unelte pentru a construi și menține diverse tipuri de aplicații [1]. Npm facilitează integrarea de biblioteci și module terțe, accelerând dezvoltarea aplicațiilor și reducând timpul necesar pentru lansarea pe piață.

Node.js se remarcă prin eficiența sa în construirea de API-uri RESTful și a aplicațiilor cu pagină unică (SPA²), dar și în realizarea de soluții de backend pentru aplicații mobile. Desi este excelent pentru aplicații ce necesită un volum mare de procesare a datelor în timp real, Node.js poate avea limitări în cazul sarcinilor CPU-intensive, fiind mai puțin eficient comparativ cu alte limbi specializate pentru aceste tipuri de operațiuni.

Cu toate acestea, popularitatea sa în rândul dezvoltatorilor și suportul comunității largi rămân puncte forte, Node.js fiind adoptat de companii mari și start-up-uri deopotrivă pentru flexibilitatea și eficiența sa în dezvoltarea rapidă a aplicațiilor web moderne.

¹npm = Node Package Manager

²SPA = Single Page Applications

2.2 React Native

React Native este un framework open source dezvoltat de Facebook, lansat în 2015, care permite dezvoltatorilor să creeze aplicații mobile native folosind JavaScript și React. Principala sa atracție este capacitatea de a scrie o singură bază de cod care rulează pe atât Android, cât și iOS, reducând astfel timpul și costurile de dezvoltare comparativ cu abordările native separate [7].

Framework-ul beneficiază de un ecosistem extins și o comunitate activă, cu acces la o multitudine de biblioteci și module terțe. De asemenea, permite dezvoltatorilor să integreze cod nativ atunci când este necesar, pentru a optimiza performanța sau pentru a adăuga funcționalități care nu sunt direct accesibile prin JavaScript.

2.3 PostgreSQL

PostgreSQL este un sistem de baze de date relaționale open source, renumit pentru arhitectura sa robustă, integritatea datelor și extensibilitate. Fiind compatibil cu toate sistemele de operare majore și respectând standardele ACID din 2001, PostgreSQL suportă tipuri de date complexe și are extensii puternice. Este apreciat pentru conformitatea sa cu standardul SQL și flexibilitatea de a defini tipuri de date personalizate și funcții. PostgreSQL este gratuit, open source și susținut de o comunitate activă, oferind soluții performante pentru gestionarea datelor de orice dimensiune [18].

2.4 Framework-uri

2.4.1 Express.js

Express.js este un framework pentru Node.js care facilitează dezvoltarea aplicațiilor web și a API-urilor prin oferirea unui set robust de funcționalități, în principal prin utilizarea unui sistem de middleware. Aceasta permite manipularea cererilor și răspunsurilor HTTP într-un mod flexibil și eficient, simplificând rutarea și gestionarea erorilor.

Utilizând Express, dezvoltatorii pot defini cu ușurință rute specifice pentru diferite tipuri de cereri HTTP, cum ar fi GET, POST, sau PUT, ceea ce îmbunătățește organizarea codului și reduce efortul necesar pentru a construi servere web scalabile. Express se integrează perfect cu diverse middleware-uri din ecosistemul Node, permitând extinderea funcționalităților aplicației [12].

2.4.2 Expo

Expo este o platformă și un set de unelte care funcționează pe baza React Native, simplificând și mai mult dezvoltarea aplicațiilor mobile. Expo oferă un set bogat de API-

uri gata de utilizat care acoperă funcționalități comune ale dispozitivelor mobile, cum ar fi camera, locația și notificările push. Dezvoltatorii pot începe rapid un proiect nou cu Expo fără a configura nativ fiecare platformă, ceea ce accelerează procesul de dezvoltare și testare [11].

Expo oferă, de asemenea, Expo Go, o aplicație care permite testarea rapidă a aplicațiilor React Native pe dispozitive reale fără a fi nevoie de instalarea lor. Aceasta facilitează testarea și partajarea aplicațiilor în fazele de dezvoltare inițială. Cu toate acestea, în timp ce Expo simplifică dezvoltarea, poate limita accesul la unele funcționalități native mai avansate, necesitând tranziția la un proiect React Native pentru acces complet la platformă.

2.5 Amazon Web Services

Amazon Web Services (AWS) este o colecție de servicii de cloud computing oferite de Amazon, lansată în 2006. AWS oferă o gamă largă de servicii de infrastructură, cum ar fi putere de calcul, opțiuni de stocare și baze de date, toate furnizate ca servicii la cerere. Aceasta permite companiilor să scaleze și să administreze infrastructura IT cu eficiență și flexibilitate crescută, reducând astfel costurile operaționale și accelerând inovația. AWS este recunoscut pentru fiabilitatea, securitatea și scalabilitatea sa, fiind liderul pieței de cloud computing [21].

2.5.1 Elastic Beanstalk

AWS Elastic Beanstalk este un serviciu *Platform as a Service* oferit de Amazon Web Services care facilitează desfășurarea și managementul aplicațiilor web și serviciilor backend pe infrastructura AWS. Este ideal pentru dezvoltatorii care doresc să se concentreze pe dezvoltarea codului fără a gestiona manual infrastructura serverelor.

Utilizând Elastic Beanstalk, dezvoltatorii pot încărca codul aplicațiilor lor, iar serviciul automatizează întregul proces de desfășurare - de la provisionarea resurselor, balansarea încărcării, auto-scalarea până la monitorizarea stării aplicației. Suportă o varietate de platforme de dezvoltare cum ar fi Java, .NET, PHP, Node.js, Python, Ruby, și Docker, permitând implementarea rapidă și eficientă a backendurilor aplicațiilor. Elastic Beanstalk oferă, de asemenea, integrare simplă cu baze de date și servicii de stocare AWS [13].

2.5.2 Relational Database Service

Amazon Relational Database Service (RDS) este un serviciu gestionat de baze de date relaționale oferit de Amazon Web Services, care simplifică configurarea, operarea și scalarea bazelor de date în cloud. RDS permite utilizatorilor să ruleze baze de date relaționale comune precum MySQL, PostgreSQL, MariaDB, Oracle, și Microsoft SQL

Server fără a necesita cunoștințe aprofundate de administrare a sistemelor de baze de date.

Serviciul gestionează sarcini de administrare consumatoare de timp, precum backup-ul datelor, patching-ul software, și scalarea hardware-ului. RDS este optimizat pentru performanță și durabilitate, oferind opțiuni de replicare a datelor pentru a crește disponibilitatea și durabilitatea [14].

2.5.3 S3 (Simple Storage Service)

Amazon Simple Storage Service (S3) este un serviciu de stocare obiecte oferit de Amazon Web Services. Acesta oferă scalabilitate, securitate, performanță și durabilitate pentru stocarea și recuperarea oricărui volum de date, de la oriunde de pe web. S3 este folosit în mod obișnuit pentru arhivarea în cloud, backup și recuperare în caz de dezastre, dar și pentru scenarii precum găzduirea site-urilor web, aplicații mobile și Big Data analytics [3].

S3 oferă o structură simplă, unde datele sunt organizate în „găleți” (buckets), fiecare identificat unic printr-un nume dat de utilizator. Serviciul suportă managementul versiunilor, ceea ce permite păstrarea mai multor versiuni ale aceluiași obiect în același bucket, facilitând astfel recuperarea versiunilor anterioare sau a datelor șterse accidental.

De asemenea, S3 oferă diverse opțiuni de control al accesului, inclusiv listele de control al accesului (ACLs) și politicile bazate pe roluri, asigurând că datele pot fi accesate sau administrate numai de utilizatorii autorizați.

2.6 Google Cloud

Google Cloud este o suită de servicii de cloud computing oferite de Google, care furnizează hosting pe aceeași infrastructură pe care Google o utilizează intern pentru produsele sale, cum ar fi Google Search și YouTube.

2.6.1 Maps JavaScript API

Maps JavaScript API este un serviciu de la Google Cloud care permite integrarea hărților interactive direct în paginile web. API-ul oferă funcționalități cum ar fi vizualizarea hărților, căutarea de locații și trasarea de rute. Este folosit pentru a adăuga elemente geografice personalizate la aplicații web, facilitând dezvoltarea de soluții care necesită interactivitate bazată pe locație [16].

2.6.2 Places API

Places API de la Google Cloud permite dezvoltatorilor să acceseze informații detaliate despre locații specifice. API-ul oferă funcții pentru căutarea locațiilor pe baza unor cuvinte cheie sau a tipurilor de locuri, recuperarea detaliilor despre un anumit loc, și sugerarea de locații în timp ce utilizatorii tastează. Este util pentru aplicații care necesită date precise despre locații, cum ar fi adrese, recenzii, și orare de funcționare [17].

2.6.3 Distance Matrix API

Distance Matrix API de la Google Cloud furnizează informații despre timpul de călătorie și distanțele între multiple locații, folosind metode diverse de transport, cum ar fi mersul pe jos, conducerea, transportul public, sau bicicleta. Acest API este util pentru aplicații care necesită calcularea optimă a rutelor sau estimarea duratei călătoriilor, oferind date precise care pot ajuta la planificarea logistică sau la îmbunătățirea experienței utilizatorilor [10].

2.6.4 Directions API

Directions API de la Google Cloud oferă rute detaliate pentru navigație între locații. Aceasta permite dezvoltatorilor să genereze direcții pas cu pas pentru diverse moduri de transport, inclusiv mersul pe jos, conducerea, ciclismul și transportul public. API-ul facilitează integrarea în aplicații de călătorie sau logistică, oferind rute precise și informații despre durata călătoriei, distanțe și posibile obstacole de traseu [9].

2.7 XGBoost și YOLOv5

XGBoost (eXtreme Gradient Boosting) este o bibliotecă open-source pentru învățare automată, proiectată pentru a fi eficientă, flexibilă și portabilă. Este implementată pentru a optimiza algoritmii de gradient boosting și este renumită pentru viteza și performanța sa în competiții de știință a datelor. XGBoost oferă suport pentru mai multe limbi de programare, inclusiv Python, R, Java, și Scala [8]. Este utilizat în mod comun pentru probleme de clasificare și regresie, oferind funcții avansate pentru gestionarea datelor lipsă, regularizarea modelelor pentru a preveni suprapotrivirea și îmbunătățirea generală a acurateței modelului. XGBoost implementează algoritmi de învățare automată sub cadrul framework-ului Gradient Boosting și oferă un boosting paralel ar arborilor cunoscut și ca GBDT sau GBM [22]

YOLOv5 (You Only Look Once, version 5) este un model de detectare a obiectelor în timp real, parte a familiei de algoritmi YOLO. Este cunoscut pentru viteza și eficiența sa în detectarea obiectelor în imagini și videoclipuri. YOLOv5 este implementat în PyTorch

și a fost optimizat pentru a avea o arhitectură mai compactă și mai rapidă comparativ cu versiunile anterioare, fără a compromite acuratețea [20]. Este utilizat pe scară largă în aplicații de supraveghere video, analiza conținutului vizual și sisteme autonome, unde capacitatea de a detecta rapid și precis obiecte este esențială.

2.8 Altele

2.8.1 OpenAI API

API-ul GPT (Generative Pre-trained Transformer) de la OpenAI este o interfață avansată pentru modele de limbaj bazate pe tehnologia de învățare automată transformer. Aceste modele sunt antrenate pe un volum masiv de texte și sunt capabile să genereze răspunsuri coerente și contextual adecvate în conversații, facilitând dezvoltarea de aplicații de chatbot inteligente.

API-ul GPT permite integrarea ușoară a capabilităților de procesare a limbajului natural în diverse aplicații, oferind suport pentru crearea de răspunsuri personalizate, summarizarea textelor, traduceri și multe alte funcții de interacțiune bazate pe text. Este o soluție robustă și versatilă pentru îmbunătățirea interacțiunilor automate cu utilizatorii în diferite contexte, de la asistență pentru clienți până la sisteme de recomandare personalizate.

Capitolul 3

Arhitectura aplicației

Arhitectura aplicației constă în două proiecte separate, unul pentru **backend** și altul pentru **frontend**.

Backend-ul este dezvoltat în Node.js, iar ca și limbaj de programare am folosit TypeScript, întrucât este un limbaj ce previne apariția erorilor la runtime, dat fiind faptul că verificarea tipurilor la compilare ajută la identificarea erorilor înainte de rularea codului. Backend-ul aplicației se ocupă de accesul la baza de date și servirea datelor către client.

Pentru frontend am folosit React Native cu framework-ul Expo, o platformă ce permite dezvoltarea de aplicații mobile cross-platform. De asemenea, și pentru frontend am folosit TypeScript. Frontend-ul gestionează interfața aplicației și interacțiunea cu datele provenite din backend.

Backend-ul aplicației este găzduit în Amazon Web Services (AWS) folosind AWS Elastic Beanstalk pentru gestionarea serviciilor web, ceea ce simplifică procesul de administrare a aplicației. Pentru stocarea datelor, am folosit Amazon RDS, care permite operarea bazei de date relaționale într-un mod eficient și accesibil. De asemenea, am utilizat Amazon S3 pentru stocarea imaginilor încărcate în aplicație.

3.1 Backend

3.1.1 Baza de date

Aplicația utilizează **TypeORM** pentru a gestiona conexiunile și operațiile cu baza de date. Integrarea TypeORM aduce după sine gestionarea automată a operațiilor CRUD direct pe obiecte și suport pentru migrații, avantaje ce optimizează procesul de dezvoltare și ușurează interacțiunea cu baza de date.

Configurația *DataSource* specifică TypeORM este setată pentru a se conecta la un server **PostgreSQL**, folosind detalii precum hostul, portul, numele utilizatorului și parola, care pot fi configurate din variabilele de mediu pentru mai multă securitate și compatibilitate cu mediul AWS RDS unde a fost hostată baza de date.

Baza de date are 2 tabele principale, una pentru datele utilizatorilor (username-ul, email-ul, parola și push token-ul necesar pentru trimitera notificărilor), *User*, iar cealaltă pentru toate detaliile mașinilor, numită *Car*, ce reține câmpuri precum producătorul, modelul, anul fabricării, kilometrajul, tipul de motorizare și alte detalii specifice. Pentru fiecare tip de document corespunzător mașinilor am reținut câte 2 tabele având următoarea structură:

- **active** - tabela unde se află în orice moment maxim o intrare per mașină, reprezentând datele actuale ale documentului respectiv;
- **history** - tabela unde se află toate documentele expirate de tipul respectiv.

Între tabela *User* și *Car* există o relație de tip *one-to-many*, astfel încât un utilizator să poată avea mai multe mașini. Același tip de relație *one-to-many* este regăsită și între tabela *Car* și tabelele de istoric pentru documentele mașinii. Pentru tabelele documentelor neexpirate însă, am folosit o relație *one-to-one* cu tabela *Car*, întrucât aşa cum am descris mai sus, fiecare mașină poate avea maxim o entitate activă per document, la fel cum și fiecare document poate apartine unei singure mașini.

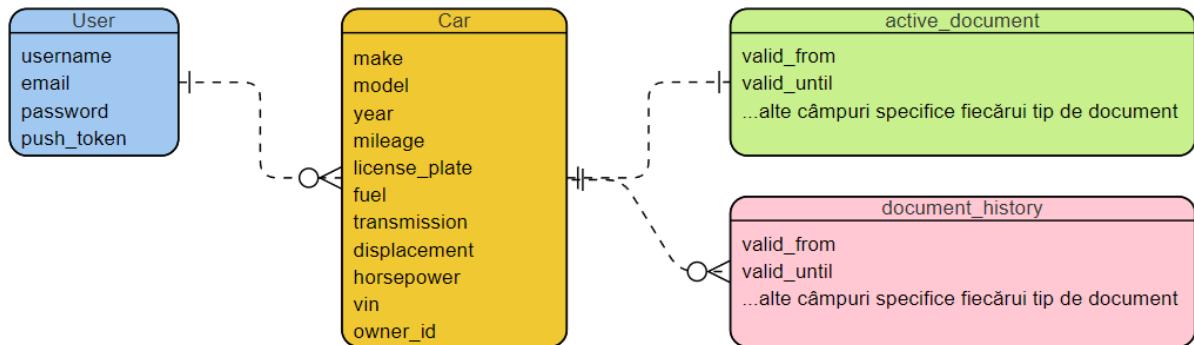


Figura 3.1: Arhitectura bazei de date

3.1.2 Controllers

Controllerele sunt componentele principale din backend-ul aplicației și cuprind operațiuni specifice pentru fiecare entitate, inclusiv operațiile CRUD¹ și alte funcționalități.

¹CRUD = create, read, update and delete

Pentru logica de autentificare am rezervat un controller separat, pentru a respecta principiul *Separation of Concerns*, astfel încât fiecare secțiune să aibă o responsabilitate unică.

Astfel, controllerele folosite sunt următoarele:

- **AuthController** - este responsabil de funcțiile de login și register ale aplicației;
- **UserController** - este responsabil de actualizarea datelor despre utilizatori (schimbarea username-ului și a parolei) și de gestionarea push token-ului necesar fiecarui user pentru a putea primi notificări pe telefon;
- **CarController** - aduce la un loc toate operațiile care se pot aplica unei mașini: pe lângă operațiile CRUD, conține și metodele prin care sunt folosite cele 2 modele de Machine Learning;
- **InspectionController**, **InsuranceController**, **ServiceController** - aceste controllere sunt responsabile de operațiile CRUD specifice fiecarui tip de document, atât pentru cele active, cât și pentru cele din tabelele destinate celor expirate.

3.1.3 Rutele pentru API-uri

Pentru fiecare metodă din controlere, am avut un API route prin care aceasta poate fi apelată la endpoint-ul aplicației backend. Pentru a eficientiza transpunerea acestor metode în API-uri și a evita codul repetitiv, am creat un fișier de utilități pentru crearea rutelor, în care fiecare controller are asociat un array cu informații pentru fiecare rută. Astfel rutele vor fi înregistrate dinamic, în loc să scriem manual fiecare rută.

```
const authRoutes = [
  {
    method: 'post',
    route: '/register',
    controller: AuthController,
    action: 'register',
    middlewares: [],
  },
  {
    method: 'post',
    route: '/login',
    controller: AuthController,
    action: 'login',
    middlewares: [],
  },
]
```

```
];
```

Astfel, fiecare rută va avea aici toate proprietățile necesare, iar în final creez array-ul cu toate rutele.

```
export const Routes = [
  ...authRoutes,
  ...userRoutes,
  ...carRoutes,
  ...insuranceRoutes,
  ...inspectionRoutes,
  ...serviceRoutes,
];
```

Pentru a înregistra aceste rute, am iterat prin fiecare obiect din acest array pentru a executa funcția asociată fiecărei rute, preluând și metoda de accesare a acesteia, precum și middleware-ul, acolo unde este cazul.

```
Routes.forEach((route) => {
  const middlewares = route.middlewares || [];
  (app as any)[route.method] (
    route.route,
    ...middlewares,
    async (req: Request, res: Response, next: express.NextFunction) => {
      const result = await new (route.controller as any)() [route.action] (
        req, res).catch(next);
      if (result !== undefined) {
        res.send(result);
      }
    },
  );
});
```

3.1.4 Middlewares

În cadrul aplicației, am folosit un middleware pentru a asigura securitatea și controlul accesului la diferitele resurse expuse prin API-uri. Acest middleware este utilizat pentru a verifica autenticitatea și autorizația utilizatorilor care accesează anumite endpoint-uri,

cu ajutorul tokenului **JWT**². JWT este un standard deschis (RFC 7519) care definește un mod compact și independent pentru a transmite în siguranță informații între părți ca un obiect JSON. Aceste informații pot fi verificate și de încredere, deoarece sunt semnate digital. JWT-urile pot fi semnate folosind o cheie secretă (cu algoritmul HMAC) sau o pereche de chei publică/privată folosind RSA sau ECDSA [4].

Middleware-ul *isAuthenticated* este esențial pentru gestionarea accesului la resursele care necesită un utilizator autentificat. Acesta extrage tokenul JWT din header-ul Authorization al cererii HTTP. Dacă tokenul este absent sau formatul său este incorect, se aruncă o eroare cu codul de stare 403, indicând că accesul este interzis din cauza lipsei acestuia.

Tokenul este apoi verificat folosind cheia secretă specificată în variabilele de environment `JWT_SECRET`. Verificarea tokenului asigură că a fost semnat de server și că nu a fost alterat. Dacă tokenul este invalid sau expirat, se aruncă o eroare cu codul de stare 500, semnalând că autentificarea a eșuat.

Dacă tokenul este valid, se extrage id-ul utilizatorului din payload-ul tokenului. Aceasta este stocat în request-ul cererii, permitând controllerelor să acceseze detaliile utilizatorului autentificat. Procesul se finalizează apelând `next()`, care transferă controlul către următorul handler în lanțul de procesare a cererii.

```
export const isAuthenticated = (req: Request, res: Response, next: NextFunction) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[0] === 'Bearer' ?
    authHeader.split(' ')[1] : null;
  if (!token) {
    throw new CustomError(403, 'No token provided');
  }
  jwt.verify(token, process.env.JWT_SECRET as string, (err, decoded) => {
    if (err) {
      throw new CustomError(500, 'Failed to authenticate token');
    }
    const payload = decoded as jwt.JwtPayload;
    if (payload.id) {
      (req as any).userId = payload.id;
      next();
    } else {
      throw new CustomError(500, 'Invalid token payload');
    }
  });
}
```

²JWT = JSON Web Token

```
};
```

Middleware-ul este integrat în sistemul de gestionare a rutelor prin specificarea lui în array-ul de configurare al rutelor. Fiecare rută are un array de middleware-uri care trebuie executate înainte de handler-ul asociat rutei. Astfel, această abordare asigură că toate rutele sunt protejate corespunzător.

3.1.5 Mecanismul de verificare și expirare al documentelor

Verificarea documentelor ce expiră se face printr-un cron job aparținând librăriei *node-cron*, ce este declanșat zilnic la miezul nopții. Scopul său este de a verifica și a expira documentelor ale căror date de valabilitate au depășit ziua curentă. Folosind mecanismul *async-await*, ne asigurăm ca operațiilor sunt finalizate succesiv și că fiecare pas este executat înainte de a începe următorul.

```
import * as cron from 'node-cron';
//...
cron.schedule('0 0 * * *', async () => {
    console.log('Running daily check for expiring services at', new Date());
    await checkAndExpireInsurances();
    await checkAndExpireInspections();
    await checkAndExpireServices();
    await scheduleExpiryNotifications();
});
```

Expirarea documentelor se face prin cele 3 funcții (*checkAndExpireInsurances()*, *checkAndExpireInspections()*, *checkAndExpireServices()*), care sunt similare ca și implementare, întrucât pentru fiecare în parte se verifică intrările cu data de expirare ce depășește ziua curentă. Pentru a realiza această verificare, am folosit librăria *date-fns* pentru a calcula data zilei de ieri în formatul *Date* suportat de limbajul JavaScript.

```
export const checkAndExpireInsurances = async () => {
    const yesterday = endOfDay(subDays(new Date(), 1));
    const insuranceController = new InsuranceController();
    const activeInsurances = await AppDataSource.getRepository(
        ActiveInsurance).find({
            where: {
                validUntil: LessThanOrEqual(yesterday),
            },
        });
    // ...
}
```

```

});  

console.log('Found insurances to expire:', activeInsurances);  

for (const insurance of activeInsurances) {  

  try {  

    const result = await insuranceController.expireById(insurance.id);  

    console.log(`Expired insurance with id ${insurance.id}: ${result}`)  

  ;  

  } catch (error) {  

    console.error('Error expiring insurance:', error);  

  }
}  

}
};
```

În ceea ce privește conceptul expirarea propriu-zisă, folosesc metoda *expireById*, prezentă în fiecare controller pentru documente, în care se caută după *id* documentul respectiv și se mută din tabela activă corespunzatoare lui în tabela de istoric.

Metoda folosește o tranzacție gestionată prin *AppDataSource.transaction*. Aceasta este o funcție care acceptă un callback, unde toate operațiunile cu baza de date executate în cadrul acestui callback sunt parte a unei singure tranzacții, pentru executarea consistentă a operațiilor. Avantajul major este că dacă oricare dintre operațiuni eșuează, toate modificările făcute în cadrul tranzacției sunt automat anulate. Astfel, evit situația în care datele despre document sunt salvate în tabela de istoric și rămân și în tabela activelor, în cazul unei erori la ștergerea acestora din tabela de active.

```

expireById = async (insuranceId: number) => {  

  await AppDataSource.transaction(async (transactionalEntityManager) =>  

{  

  const insurance = await transactionalEntityManager.findOne(  

ActiveInsurance, {  

  where: { id: insuranceId },  

  relations: ['car'],  

});  

  if (!insurance) {  

    throw new CustomError(404, 'Insurance not found');  

}  

  const history = transactionalEntityManager.create(InsuranceHistory,  

{  

  ...insurance,  

  car: insurance.car,
```

```

    });
    await transactionalEntityManager.save(InsuranceHistory, history);
    await transactionalEntityManager.remove(ActiveInsurance, insurance)
  ;
});
return 'Insurance expired';
};

```

3.1.6 Trimiterea notificărilor

Pentru trimiterea notificărilor, mecanismul dezvoltat folosește **Expo** și este gestionat printr-o funcție centrală, *scheduleExpiryNotifications*, care coordonează planificarea și trimiterea notificărilor pentru fiecare tip de document.

```

export const scheduleExpiryNotifications = async () => {
  await scheduleNotificationsFor(ActiveService, 'Service');
  await scheduleNotificationsFor(ActiveInsurance, 'Insurance');
  await scheduleNotificationsFor(ActiveInspection, 'Inspection');
};

```

Expo oferă un SDK complet pentru gestionarea notificărilor push, ceea ce include înregistrarea tokenurilor de notificare specifice dispozitivului și trimiterea notificărilor. Aceste tokenuri sunt esențiale pentru că identifică fiecare dispozitiv în parte și permit aplicației să trimită notificări specifice. SDK-ul gestionează complexitățile legate de diferitele platforme mobile, cum ar fi iOS și Android, astfel nu apar dificultăți în ceea ce privește gestionarea particularităților sistemelor de operare.

Funcția *scheduleNotificationsFor* este invocată pentru fiecare entitate (asigurare, inspecție, revizie), determinând zilele rămase până la expirare și trimînd notificări corespunzătoare. Se utilizează intervale predefinite de 7, 3, 1 și 0 zile înainte de expirarea documentului, asigurându-se astfel că utilizatorii primesc notificări la momentele importante dinaintea expirării. Pentru calculul datei exacte de notificare, se folosește funcția *addDays* din biblioteca *date-fns*, care adaugă un număr specificat de zile la data curentă. Aceasta permite stabilirea intervalului exact în care notificarea trebuie trimisă. Datele sunt apoi ajustate pentru a începe și a se termina la începutul și sfârșitul zilei respective, pentru ca verificările să acopere întreaga zi.

Prin intermediul TypeORM, se efectuează interogări pentru a găsi entitățile care expiră în intervalul stabilit. Aceste interogări includ relații necesare pentru a accesa informații despre vehicul și proprietar pentru trimiterea notificărilor.

Notificările sunt trimise folosind tokenuri de notificare venite de pe partea de client din frontend și validate pentru a asigura apartenența la un dispozitiv activ. Mesajele de notificare sunt personalizate pentru a include detalii relevante despre expirarea documentului, precum și numărul de zile rămase până la expirare.

Expo gestionează toate aspectele tehnice ale trimiterii notificărilor, inclusiv gestionarea ratelor de trimitere și reîncercările în caz de eșec, ceea ce îmbunătățește fiabilitatea sistemului de notificare. Notificările sunt împărțite în loturi (chunks) și trimise asincron pentru a optimiza procesul și pentru a reduce încărcarea pe server.

```
export const scheduleNotificationsFor = async (entity, description) => {
  const today = new Date();
  for (const daysBefore of notificationIntervals) {
    const notificationDate = addDays(today, daysBefore);
    const startOfDay = new Date(notificationDate.setHours(0, 0, 0, 0));
    const endOfDay = new Date(notificationDate.setHours(23, 59, 59, 999));
    ;
    const itemsExpiringSoon = await AppDataSource.getRepository(entity)
      .createQueryBuilder('entity')
      .leftJoinAndSelect('entity.car', 'car')
      .leftJoinAndSelect('car.owner', 'owner')
      .where(`entity.validUntil >= :startOfDay AND entity.validUntil <= :endOfDay`, { startOfDay, endOfDay })
      .getMany();
    console.log(`#${itemsExpiringSoon.length} items found expiring on ${notificationDate.toISOString().split('T')[0]}`);
    for (const item of itemsExpiringSoon) {
      if (!item.car || !item.car.owner || !item.car.owner.pushToken) {
        console.error(`Missing car, owner, or push token for item ID: ${item.id}`);
        continue;
      }
      if (!Expo.isExpoPushToken(item.car.owner.pushToken)) {
        console.error(`Invalid Expo push token for user ID: ${item.car.owner.id}`);
        continue;
      }
      await sendNotifications(
        [item.car.owner.pushToken],
        `${description} Expiration Reminder`,
        `Your ${description} for your car is expiring in ${daysBefore}`
```

```
    day(s).` ,  
  );  
}  
}  
};
```

3.1.7 Gestionaarea erorilor

Clasa *CustomError* este concepută pentru a standardiza răspunsurile de eroare în aplicație. Aceasta include un *status*, care reprezintă codul de stare HTTP asociat erorii, și un *message*, care furnizează o descriere clară a erorii pentru a fi ușor de înțeles de către partea de client care apelează backend-ul. Constructorul clasei acceptă aceste două argumente, permitând instantierea obiectelor de eroare.

```
export class CustomError {  
  message: string;  
  status: number;  
  constructor(status: number, message: string) {  
    this.status = status;  
    this.message = message;  
  }  
}
```

Un exemplu de utilizare al clasei *CustomError* în cadrul aplicației:

```
throw new CustomError(403, 'No token provided');  
//...  
throw new CustomError(404, 'User not found');
```

3.2 Frontend

3.2.1 Componente

Componențele în React Native sunt elementele de bază utilizate pentru construirea interfețelor aplicațiilor mobile. Fiecare componentă în React Native reprezintă o parte a interfeței aplicației, care poate fi independentă și reutilizabilă. Acest sistem de componente permite crearea aplicațiilor dinamice prin combinarea diferitelor blocuri într-un

mod modular. Fiecare componentă poate săține propria sa stare internă și logică, fiind capabilă să răspundă la interacțiuni din partea utilizatorilor și să proceseze datele conform necesităților.

React Native vine cu o serie de componente vizuale prestabilite, fiind blocuri de bază pentru a construi interfețe de utilizator. Câteva dintre cele mai comune componente sunt următoarele:

- **View** - cea mai fundamentală componentă, funcționează ca un container flexibil care poate conține și alte componente, ce ajută la organizarea layout-ului paginii, asemănător unui `<div>` din HTML;
- **Text** - componentă folosită pentru a afișa text;
- **TextInput** - o componentă fundamentală folosită pentru a prelua text ca și input de la tastatură;
- **ScrollView** - este precum componenta View, cu deosebirea că aceasta permite scroll-ul în conținutul acesteia;
- **FlatList** - permite redarea listelor de elemente, fiind optimizată pentru liste mari, întrucât redă doar elementele vizibile pentru a economisi resurse;
- **Image** - componentă ce încorporează și gestionează imaginile în pagină;
- **Button** - oferă un buton simplu ce poate fi personalizat;
- **TouchableOpacity** - este o alternativă la butonul standard, această componentă face orice element interactiv și răspunde la interacțiunea cu utilizatorul prin modificarea opacității. Este utilă pentru a crea elemente interactive personalizate;
- **Modal** - componentă ce face posibilă afișarea conținutului într-un strat suprapus, în fața celorlalte componente, fără a naviga spre o altă pagină sau a schimba starea curentă a aplicației.

Pe lângă aceste componente standard, React Native oferă flexibilitatea de a crea componente personalizate, în care pot fi integrate atât componente prestabilite, cât și alte componente personalizate.

Aplicația conține mai multe tipuri de componente personalizate, fiecare contribuind diferit la layout-ul și funcționalitățile screen-urilor de UI. Componentele personalizate de tip elemente din listă au rolul de a pune la un loc informațiile relevante ce urmează să fie afișate în paginile ce conțin un FlatList sau un ScrollView, în timp ce componentele care au la bază un TextInput au scopul de a personaliza câmpurile de input din formularele prezente în aplicație în funcție de tipul acestora. De asemenea, aceste formulare reprezintă componente separate la rândul lor, sub forma unor Modal-uri din React Native, iar

datele despre utilitățile vehiculelor, reprezentând asigurarea, inspecția tehnică periodică și revizia, sunt afișate în meniul principal sub forma unor widget-uri, care la rândul lor sunt integrate în componente separate. Conținutul aplicației pe partea de UI este încadrat vertical între o bară de navigare și o bară superioară, acestea fiind și ele realizate în componente individuale.

Componente pentru elemente din listă

- **CarItemDashboard** - este componenta care încorporează detaliile despre mașini ce apar în meniul principal, alături de widget-urile specifice corespunzătoare mașinii respective;
- **CarItemList** - reprezintă un element din lista mașinilor deținute de utilizator, afișând informații despre modelul mașinii, anul fabricației, kilometraj și numărul de înmatriculare. De asemenea, elementele acestei componente sunt încadrate într-un TouchableOpacity, astfel că interacțiunea cu această componentă îl va redirecționa pe utilizator la ecranul de editare al mașinii;

Componente pentru câmpuri de input

- **DateInputField** - componenta folosită pentru a prelua și manevra input de tip Date, utilitatea acesteia este pentru datele de început și expirare al utilităților mașinii. Folosește componenta **DateTimePickerModal** din librăria *react-native-modal-datetime-picker*, ce afișează un pop-up de tip calendar în care utilizatorul poate selecta exact data dorită;
- **FormAuthField** - conține, pe lângă un TextInput clasic destinat preluării datelor de la utilizator, o iconiță trimisă ca parametru în componentă, folosind **FontAwesome5** din librăria *@expo/vector-icons*. Astfel, componenta este personalizabilă nu doar din punctul de vedere al textului primit, cât și din punctul de vedere al iconiței afișate lângă text, în conformitate cu scopul utilizării câmpului. Pentru a putea folosi componenta și pentru câmpurile de tip parolă, aceasta are și parametrul *secureTextEntry*, o valoare de tip boolean care indică dacă textul introdus să fie ascuns sau nu;
- **NumberInputField** - foarte similară cu *FormAuthField*, însă este destinat tipului de date numeric, astfel încât utilizatorului îi vor aparea doar numere pe tastatura telefonului atunci când interacționează cu acest tip de câmp. Diferența constă în parametrul *keyboardType= "numeric"*, absent în componenta precedentă;
- **FormDropdownField** - folosind **Picker** din librăria *@react-native-picker/picker*, această componentă este utilizată pentru câmpurile în care utilizatorul trebuie să

aleagă dintr-o listă de valori predefinite, de exemplu pentru tipul de combustibil al mașinii (diesel, gasoline, hybrid, electric);

- **PictureInputField** - componenta cu ajutorul căreia utilizatorul poate încărca poze în formulare. Folosește librăria *expo-image-picker*, ce permite atât captarea unei fotografii pe moment, cât și selectarea uneia din galerie. Înainte de încărcarea unei fotografii, utilizatorului îi sunt cerute permisiuni pentru accesarea camerei și a galeriei foto, iar dacă acestea nu sunt oferite, o alertă corespunzătoare va apărea;

```
const verifyPermissions = async () => {
  const { status: cameraStatus } = await ImagePicker.requestCameraPermissionsAsync();
  const { status: galleryStatus } = await ImagePicker.requestMediaLibraryPermissionsAsync();
  if (cameraStatus !== "granted" || galleryStatus !== "granted") {
    Alert.alert(
      "Insufficient Permissions!",
      "You need to grant camera and photo library permissions to use this feature.",
      [{ text: "Okay" }]
    );
    return false;
  }
  return true;
};
```

Componente de tip formular

- **EditProfileFormModal** - conține două câmpuri de intrare de tipul **FormAuthField**, unul reprezentând username-ul, iar celălalt email-ul. Este folosit atunci când utilizatorul optează să-și actualizeze aceste date. Componenta primește ca parametru o funcție callback *onSave*, apelată atunci când butonul de salvare a actualizărilor este apăsat. În această situație, componenta este folosită în pagina de profil a utilizatorului, iar funcția callback actualizează aceste date efectuând un call către backend-ul aplicației, prin funcția *updateUserDataApiCall*. Dacă utilizatorul introduce doar unul dintre cele două câmpuri, atunci doar acesta va fi actualizat, iar celălalt va rămâne neschimbat;

```
const formattedFormData = {
  username: formData.username || undefined,
```

```

        email: formData.email || undefined,
    };

    console.log(`Updating user data: ${JSON.stringify(formattedFormData)}
        `);

    await updateUserDataApiCall(
        formattedFormData,
        parseInt(userId),
        userToken
    );

    Alert.alert("Success", "Profile updated successfully.");

```

- **ChangePasswordFormModal** - componentă ce încorporează formularul de schimbare a parolei, similară cu cea pentru actualizarea username-ului și a email-ului, diferită însă prin faptul ca de această dată, parametrul *secureTextEntry* din **FormAuthField**-uri are valoarea *true*;
- **InsuranceFormModal** - această componentă este folosită pentru formularul de introducere a detaliilor despre asigurarea unui vehicul. Are la bază un **Modal** ce conține un **ScrollView** cu mai multe câmpuri de input, pentru numărul poliței de asigurare, compania de asigurări corespunzătoare, data de început și data de expirare a asigurării, alături de un câmp pentru încărcarea unei fotografii cu asigurarea propriu-zisă. La fel ca la formularele pentru actualizarea datelor utilizatorului, și acesta primește ca parametru o funcție callback *onSave*, apelată la apăsarea butonului de trimitere al formularului;
- **ITPFormModal** - similară cu componenta precedentă, de data aceasta destinată pentru datele legate de inspecția tehnică periodică a mașinii. Conține cele două câmpuri de tip **DateInputField** ce marchează valabilitatea acestui document, *validFrom* și *validUntil*. În cazul în care utilizatorul nu selectează nicio dată, atunci valabilitatea va fi considerată începând cu ziua curentă, până anul următor, aşa cum este în mod obișnuit valabilitatea pentru un astfel de document;
- **ServiceFormModal** - această componentă este și ea similară cu precedentele două, fiind destinată pentru datele despre revizia mașinii. În ceea ce privește câmpurile de intrare, revizia se efectuează, de cele mai multe ori, fie din 10,000 în 10,000 sau din 15,000 în 15,000 de kilometri (depinzând de vechimea și starea de întreținere a mașinii), fie periodic la un an sau doi, în cazul în care intervalul de kilometri nu a fost atins. Astfel, în această componentă, utilizatorul poate introduce atât o dată estimativă pentru următoarea revizie, cât și un interval de kilometri. Pe lângă

aceste date, utilizatorul poate introduce și numele service-ului auto unde a efectuat revizia, dar și diverse notițe despre schimbările efectuate la revizie, într-o zonă de text extinsă care îi permite acest lucru;

```
<Text style={styles.editField}>Next service in:</Text>
<FormDropdownField
  iconName="tachometer-alt"
  selectedValue={service.mileageInterval?.toString() || ""}
  items={[
    { label: "10,000", value: "10000" },
    { label: "15,000", value: "15000" },
  ]}
  onValueChange={(mileageInterval) =>
    setService({
      ...service,
      mileageInterval,
    })
  }
/>

```

- **OptionsModal** - este componenta care afișează cele trei opțiuni de mai sus pentru utilitățile mașinii, sub forma unui **Modal** ce conține un **FlatList**. Primește ca și parametru o funcție ce rindează cele trei servicii, iar în funcție de opțiunea aleasă de utilizator, acest modal va dispărea de pe pagină, iar în locul lui va apărea formularul specific variantei alese;

```
<Modal
  animationType={animationType}
  transparent={transparent}
  visible={visible}
  onRequestClose={onRequestClose}
>
  <TouchableOpacity
    style={styles.modalOverlay}
    activeOpacity={1}
    onPressOut={onRequestClose}
  >
    <View style={styles.modalContainer}>
      <FlatList
        ...
      </FlatList>
    </View>
  </TouchableOpacity>
</Modal>
```

```

        data={options}
        renderItem={renderOption}
        keyExtractor={(item) => item}
      />
    </View>
  </TouchableOpacity>
</Modal>

```

Componente pentru widget-uri

Cele trei componente de acest tip, **InsuranceWidget**, **ITPWidget** și **ServiceWidget** sunt afișate în panoul principal de control al aplicației, iar în ele sunt regăsite detaliile principale despre utilitățile fiecărei mașini. Pentru fiecare widget este afișată o bară de progres reprezentând timpul rămas până la expirarea serviciului respectiv, colorată în funcție de numărul de zile rămase, ca și procentaj, după praguri diferite.

- **Verde** - dacă au trecut mai puțin de 75% din zilele dintre data de început și data de expirare;
- **Galben** - dacă au trecut mai mult de 75% din zile, dar nu mai mult de 90%;
- **Roșu** - dacă au trecut mai mult de 90% din zile;

Progresul este calculat ca și procentajul reprezentat de timpul trecut până în momentul de față din timpul total existent între cele 2 date de valabilitate.

```

const calculateProgress = () => {
  const start = new Date(carInsurance.validFrom!).getTime();
  const end = new Date(carInsurance.validUntil!).getTime();
  const now = new Date().getTime();
  const totalDuration = end - start;
  const timeElapsedSinceStart = now - start;
  const progress = (timeElapsedSinceStart / totalDuration) * 100;
  return progress > 100 ? 100 : progress;
};

```

3.2.2 Ecrane

Asemănător componentelor, ecranele reprezintă secțiuni modulare din aplicație ce pot cuprinde mai multe componente care împreună compun conținutul necesar pentru o anu-

mită parte a aplicației, fiecare ecran reprezentând o pagină cu o funcționalitate diferită. Ecranele pot interacționa între ele prin navigare și pot avea propriile stări, datele putând fi izolate și gestionate individual pentru fiecare ecran.

Funcționalitățile aplicației „Driver’s Hub” sunt reprezentate de către un ecran cu scopul său propriu, acestea fiind cuprinse într-o componentă **NavigationContainer**, ce gestionează stările navigării între ecrane și interacțiunile dintre acestea. Fiecare ecran este inclus în container printr-o componentă de tip **Stack.Screen**, ce conține ecranul de randat și diverse opțiuni pentru randarea acestuia.

```
const AppNavigator = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Login">
        <Stack.Screen
          name="Login"
          component={LoginScreen}
          options={{ headerShown: false }}
        />
        <Stack.Screen
          name="Register"
          component={RegisterScreen}
          options={{ headerShown: false }}
        />
        // .
        // .
        // .
        <Stack.Screen
          name="ServiceScreen"
          component={ServiceScreen}
          options={{ headerShown: false }}
        />
        <Stack.Screen
          name="NearbyCarServicesScreen"
          component={NearbyCarServicesScreen}
          options={{ headerShown: false }}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

```
};
```

Ecranele ce stau la baza aplicației sunt următoarele:

- **LoginScreen** - este ecranul care apare atunci când aplicația este deschisă și conține un View ce are la bază câmpurile necesare pentru login, anume email-ul și parola. Conține și un buton ce îl redirecționează pe utilizator către ecranul de înregistrare, în cazul în care acesta nu are cont;
- **RegisterScreen** - constituie ecranul responsabil de înregistrarea utilizatorilor noi. Conține câmpuri pentru username, email și parolă, și încă un câmp adițional pentru confirmarea parolei. Pentru a asigura securitatea conturilor utilizatorilor, sunt făcute validări pentru fiecare din aceste câmpuri, astfel username-ul trebuie să aibă minimum 6 caractere, adresa de email este verificată să respecte structura corectă (să conțină „@” și „.” după acesta), iar parola trebuie să aibă minim 8 caractere, dintre care minim o literă majusculă, minim o cifră și minim un caracter special (!, @, #, \$, %, ^, &). Aceste verificări sunt realizate folosind RegEx³-uri;

```
const validateField = (field: string, value: string) => {
  let error = "";
  switch (field) {
    case "username":
      if (value.length < 6) {
        error = "Username must be at least 6 characters long";
      }
      break;
    case "email":
      if (!/^\\S+@[\\S+\\.\\S+$/.test(value)) {
        error = "Please enter a valid email address.";
      }
      break;
    case "password":
      if (
        value.length < 8 ||
        !/[A-Z]/.test(value) ||
        !/[0-9]/.test(value) ||
        !/[!@#$%^&*]/.test(value)
      ) {
```

³RegEx = Regular Expressions

```

        error =
            "Password must be at least 8 characters long and
            include an uppercase letter, a number, and a special character (!
            @#$%^&*).";
    }
    break;
case "confirmPassword":
    if (password !== value) {
        error = "Passwords do not match.";
    }
    break;
default:
    break;
}
setErrors((prev) => ({ ...prev, [field]: error }));
};

```

- **ProfileScreen** - este ecranul ce afișează datele utilizatorului și încorporează componentele responsabile de editarea acestora. De asemenea, tot în ecranul profilului se afișează și numărul de mașini deținute de utilizator și tot din acest ecran utilizatorul se poate deloga din aplicație;
- **AddCarScreen** - acest ecran cuprinde diverse tipuri de componente de tip input (*FormInputField*, *FormDropdownField*, *NumberInputField*) prin care utilizatorul poate introduce toate datele corespunzătoare pentru vehiculul său (producătorul, modelul, anul fabricației, numărul de înmatriculare, kilometrajul, tipul de combustibil acceptat de mașină, transmisia, capacitatea cilindrică, numărul de cai putere și seria VIN⁴);
- **CarScreen** - similar cu ecranul precedent, în acest ecran utilizatorul poate modifica datele despre mașină;
- **GarageScreen** - ecranul în care sunt afișate datele esențiale despre mașini sub forma unui *FlatList*, ce răndează componenta *CarItemList*. Din această componentă se poate accesa ecranul **CarScreen**, dacă utilizatorul interacționează cu un element de tip *CarItemList*;
- **DashboardScreen**

⁴VIN = Vehicle Identification Number

Este ecranul ce cuprinde panoul principal al aplicației. Ecranul încorporează o componentă **Carousel** din librăria *react-native-reanimated-carousel*, în care elementele sunt randate astfel încât utilizatorul să poată să deruleze printre ele pe orizontală. Fiecare element din Carousel reprezintă o componentă de tip *CarItemDashboard* și cuprinde atât date principale despre mașină, cât și widget-urile specifice pentru utilitățile acesteia. De asemenea, aici pot fi găsite și widget-uri cu utilitățile expirate și tot de aici pot fi adăugate widget-uri noi. Tot de aici poate fi estimat și prețul mașinii folosind modelul de machine learning (aici o sa pun redirect la el).

Pentru a reține și afișa widget-urile corespunzătoare serviciilor mașinii, am folosit o variabilă de stare *carWidgets*, definită folosind hook-ul *useState*, în care fiecare cheie este un identificator unic al unei mașini (id-ul acesteia), iar valoarea asociată fiecărei chei este un array de *string*-uri reprezentând numele serviciilor sale existente. Atunci când acest ecran este accesat, se verifică pentru fiecare mașină dacă are o asigurare activă, ITP⁵ sau revizie, iar în cazul pozitiv, se populează variabila *carWidgets* cu utilitățile existente. Același lucru se întâmplă și pentru a obține widget-urile corespunzătoare serviciilor expirate, reținute în variabila de stare *historyWidgets*.

```
const carWidgetsMap = fetchedCars.reduce((acc, car) => {
  const carIdStr = car.id!.toString();
  const widgets = [];
  if (carHasInsurance(car)) {
    widgets.push("Insurance");
  }
  if (carHasITP(car)) {
    widgets.push("ITP (Technical Inspection)");
  }
  if (carHasService(car)) {
    widgets.push("Service & Maintenance");
  }
  return { ...acc, [carIdStr]: widgets };
}, {});
setCarWidgets(carWidgetsMap);
```

Ulterior, pentru fiecare mașină, se verifică pentru fiecare tip de serviciu dacă este existent în array-ul *carWidgets[selectedCarId]* pentru a se verifica dacă se randează și widget-ul (prin componenta *InsuranceWidget*, *ITPWidget* sau *ServiceWidget*) corespunzător acestuia. În cazul în care acel tip de document a fost deja adăugat, o alertă va apărea cu un mesaj corespunzător.

⁵ITP = inspecție tehnică periodică

```

const widgetExists = (widgetName: string) => {
  if (selectedCarId) {
    const currentWidgets = carWidgets[selectedCarId.toString()] || []
    if (currentWidgets.includes(widgetName)) {
      Alert.alert("Duplicate Widget", "This widget has already been added.");
      return true;
    }
  }
  return false;
};

```

De asemenea, tot din acest ecran se pot introduce documente noi pentru mașină, prin componenta *OptionsModal*.

- **SymbolsScreen** - este ecranul în care utilizatorul poate încărca o poză cu indicatorii de eroare din bord, iar aceștia sunt detectați cu ajutorul modelului de machine learning (redirect);
- **ChatScreen** - ecranul ce conține chatbot-ul integrat cu ajutorul API-ului de la GPT-ul OpenAI. Mesajele utilizatorului sunt trimise către modelul GPT 3.5 iar răspunsurile sunt afișate în interfața de chat. Pentru a asigura cursivitate și flux logic conversației, mesajele sunt păstrate într-un array ce stocă conversația până în momentul respectiv și trimise modelului la fiecare întrebare nouă adresată;
- **LocationsScreen** - conține două funcționalități, atât afișarea service-urilor auto din apropiere, cât și posibilitatea de a marca locul de parcare pe hartă;
- **InsuranceScreen, ITPScreen, ServiceScreen** - sunt ecranele ce oferă posibilitatea de a edita informațiile despre serviciile mașinii, similare cu formularele de adăugare ale acestora;

3.2.3 Funcționalitățile bazate pe hartă

Marcarea locului de parcare

La implementarea funcționalității de marcarea locului de parcare a stat la bază librăria *react-native-maps* prin componenta *MapView*, iar pentru a afișa drumul până la acesta am folosit componenta *MapViewDirections* apartinând librăriei *react-native-maps-directions*.

Atunci când ecranul este încărcat, componenta *MapView* este inițializată și este setată locația curentă a utilizatorului prin proprietatea *region*. Locația curentă este determinată folosind coordonatele de latitudine și longitudine obținute cu ajutorul librăriei *expo-location*, după ce utilizatorul confirmă permisiunile de obținere a locației.

Pentru a marca locul de parcare propriu-zis, o componentă de tip *Marker* este plasată pe hartă la locația aleasă de utilizator, iar coordonatele acestuia sunt stocate în variabila de stare de tip *Region* a componentei și setează proprietatea *coordinate* a *Marker*-ului. Locația locului de parcare este păstrată în *AsyncStorage* pentru ca acesta să fie salvat și după închiderea aplicației.

După ce utilizatorul marchează un loc de parcare, direcțiile către acesta sunt afișate cu ajutorul *MapViewDirections*, ce trasează o rută între locația curentă și locația locului de parcare marcat, folosind API-ul *Google Directions*.

Afișarea service-urilor auto din apropiere

Pentru a afișa pe hartă service-urile auto din apropiere, aplicația începe, similar serviciului de marcare al locului de parcare, prin a solicita permisiunile de localizare de la utilizator, ulterior obținând locația actuală a utilizatorului pe hartă.

Service-urile auto din proximitate sunt obținute printr-un request către API-ul *Google Places* utilizând latitudinea și longitudinea curente. Acest request include și un parametru de rază exprimat în metri, pe care l-am setat la 3000, ce limitează căutarea locațiilor la o zonă specifică în jurul utilizatorului. De asemenea, pentru a obține toate locațiile de tip service auto, am folosit și parametrul *type* în request, pe care l-am trimis drept categoria *car_repair* pentru a cuprinde toate service-urile auto ce sunt înregistrate cu acest tip pe Google Maps.

Răspunsul primit de la API este un array de locații, fiecare locație descriind un service auto prin nume, adresă și coordonatele geografice. Folosind aceste detalii, sunt plasate markere pe hartă în locațiile respective, fiecare marker fiind interactiv și oferind informații suplimentare la atingere. În plus, utilizatorul poate deschide locația unui service selectat direct în aplicația de hărți a dispozitivului, pentru a facilita navigarea către locația respectivă.

```
const fetchNearbyCarServices = async (region: Region) => {
  const { latitude, longitude } = region;
  const apiKey: string = process.env.GOOGLE_API_KEY!;
  const radius = 4000;
  const apiUrl = `https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=${latitude},${longitude}&radius=${radius}&type=car_repair&key=${apiKey}`;
  try {
```

```

        const response = await fetch(apiUrl);
        const json = await response.json();
        console.log("Fetched nearby services");
        setServices(json.results);
    } catch (error) {
        Alert.alert("Failed to fetch", "Could not fetch nearby services")
    ;
        console.error(error);
    }
};


```

3.2.4 Comunicarea cu backend-ul

Gestionarea comunicării dintre frontend și backend a fost realizată printr-un fișier *api-service.ts*, ce centralizează interogările de API către backend într-un singur loc. Fiecare interogare a fost făcută prin câte o funcție individuală, pentru a separa mai bine responsabilitățile și a obține o structură mai organizată. Pentru a efectua apelurile API, am folosit librăria *Axios*.

Câteva avantaje ale librăriei *Axios* față de metoda nativă *fetch* din React Native sunt transformarea automată a datelor în JSON⁶, configurarea unui timeout pentru gestionarea timpurilor de răspuns și o mai ușoară gestionare a erorilor.

În ceea ce privește gestionarea erorilor, fiecare funcție din *api-service.ts* a avut un bloc principal *try...catch*, pentru a manevra erorile în cazul în care acestea există. Astfel, am verificat mai întâi dacă serverul a răspuns la cerere, dar a indicat o eroare (cu un status code cunoscut, trimis din serverul de backend, de exemplu 400, 404 sau 500) sau dacă a fost o altă eroare de rețea. Indiferent de tipul erorii primite, aceasta a fost aruncată înapoi către locul unde a fost apelată funcția, pentru ca utilizatorul să beneficieze de o întâmpinare mai prietenoasă a erorii în interfața aplicației.

```

export const loginApiCall = async (email: string, password: string) => {
    try {
        const response = await axios.post(` ${BASE_URL}/login`, {
            email,
            password,
        });
        return response.data;
    } catch (error: any) {

```

⁶JSON = JavaScript Object Notation

```

        console.log(JSON.stringify(error));
        throw error.response
            ? error.response.data
            : new Error("An error occurred during login");
    }
};


```

3.3 Machine Learning

3.3.1 Estimarea prețului de revânzare al mașinilor

Preprocesarea datelor

Setul de date pentru a antrena modelul de estimare al prețului mașinii a fost alcătuit din concatenarea a două seturi de date diferite, ambele de pe platforma *Kaggle*, întrucât am optat pentru extinderea acestuia cât mai mult. Cele două seturi de date găsite erau alcătuite într-un mod similar, multe din proprietăților datelor fiind de același tip.

Primul set de date folosit a fost *Used Car Listings: Features and Price Prediction* de pe platforma Kaggle. Acesta are aproximativ 17000 date de antrenare despre mașini, împărțite în 36 de coloane, fiecare reprezentând o proprietate a mașinii respective. Pentru a extinde aceste informații, am folosit un alt set de date de pe *Kaggle*, anume *Used Cars Dataset*, ce conține aproximativ 427000 de intrări, cu 26 de tipuri de proprietăți, date extrase de pe platforma *craigslist.org*.

Având în vedere că proprietățile datelor nu coincid în totalitate, dar ținând cont și de nevoia ca setul final de date să fie relevant pentru informațiile despre mașini introduse de utilizatorii în aplicație, a trebuit să realizez o filtrare a acestora, păstrând doar proprietățile legate de producătorul și modelul mașinii, anul de fabricație al acesteia, numărul de kilometri din bord și prețul în euro.

Am observat prin diverse experimente că pe măsură ce prețul mașinilor creștea, setul de date avea din ce în ce mai puține intrări, așa cum se vede și din figura 3.2. Din cauză că acestea influențau modul în care modelul făcea predicții, am decis să renunț la intrările cu prețul mai mare de pragul de 100000 euro, rămânând în final cu aproximativ 64700 de intrări.

Deși erau mai puține neregularități în setul de date după ce setul de date a fost truncat, încă se puteau observa valori extreme ale prețurilor care distorsionau rezultatele, așa că a fost nevoie să aduc o normalizare acestora. Varianta aleasă a fost să identific și să elimin valorile extreme din coloane *price* a setului de date, folosind un interval calculat ca 1.5 ori diferența dintre al treilea și primul quartil (IQR), eliminând datele care nu se

încadrează între aceste limite.

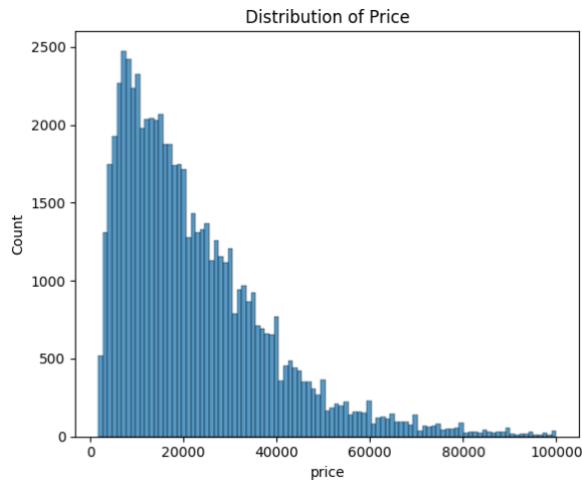


Figura 3.2: Distribuția prețurilor mașinilor din setul de date de antrenare

Arhitectura modelului

În ceea ce privește arhitectura modelului, cerința fiind una de regresie, am început prin a experimenta cu un model de tip Linear Regression. Acest tip de model presupune găsirea unei relații liniare între variabilele independente și variabila dependentă [15] (prețul în cazul nostru). Limitările acestui tip de model sunt reprezentate de presupunerea de liniaritate și de dificultatea în captarea relațiilor complexe sau non-liniare din date.

În urma experimentelor cu Linear Regression, am decis să optez pentru un model XGBoost (eXtreme Gradient Boosting), întrucât acest tip de model are capacitatea de a modela interacțiuni mai complexe dintre caracteristici.

Antrenarea modelului

Pentru a ușura găsirea hiperparametrilor potriviti pentru un model cât mai performant, am folosit Optuna, un software open-source pentru optimizarea hiperparametrilor, în care definim o funcție personalizată *objective* ce permite aplicarea diverselor strategii de căutare și alegerea dinamică a spațiului de căutare [2]. Ca și metrică pentru evaluarea modelului am folosit metrica MSE⁷. Formula pentru MSE este constituită din suma pătratelor diferențelor dintre valorile prezise și cele reale, împărțită la numărul de observații:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Funcția *objective* folosită pentru a căuta hiperparametrii potriviti:

⁷MSE = Mean Squared Error

```

def objective(trial):
    params = {
        "verbosity": 0,
        "objective": "reg:squarederror",
        "tree_method": "gpu_hist",
        "n_estimators": trial.suggest_int('n_estimators', 50, 500, 50),
        "max_depth": trial.suggest_int("max_depth", 3, 100),
        "learning_rate": trial.suggest_float("learning_rate", 0.005, 0.1,
log=True),
        "colsample_bytree": trial.suggest_float("colsample_bytree", 0.2,
0.6, log=True),
        "subsample": trial.suggest_float("subsample", 0.4, 0.8, log=True)
    ,
        "alpha": trial.suggest_float("alpha", 0.01, 10.0, log=True),
        "lambda": trial.suggest_float("lambda", 1e-8, 10.0, log=True),
        "gamma": trial.suggest_float("gamma", 1e-8, 10.0, log=True),
        "min_child_weight": trial.suggest_float("min_child_weight", 10,
1000, log=True),
        "seed": 42
    }

```

Rezultate

Așa cum am menționat mai devreme, metrica folosită pentru a evalua performanțele modelului a fost MSE, ajutat fiind și de un grafic în care se poate observa unde se situează predicțiile comparativ cu valoarea lor reală.

Astfel, am obținut următoarele rezultate, depinzând și de alcătuirea setului de date de la momentele respective. Figura 3.3 reprezintă rezultatele atunci când nu am eliminat extremele de preț, spre deosebire de figura 3.4.

3.3.2 Detectarea și recunoașterea indicatorilor din bord

Preprocesarea datelor

Pentru a putea realiza detecția și recunoașterea simbolurilor de bord pentru mașini, a fost nevoie de un set de date ce conține o multitudine de imagini cu astfel de simboluri, care să fie evidențiate și marcate. Am găsit folosind platforma Roboflow un set de date de acest tip, numit *dataset_dashboard_Image Dataset*, ce conține atât poze făcute la bordul

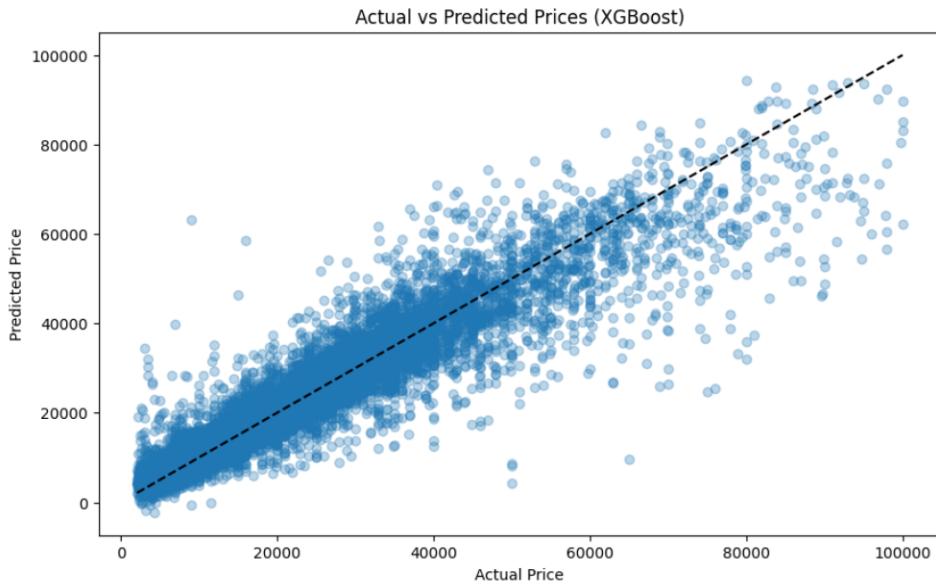


Figura 3.3: $\text{MSE} = 5677$. Fără eliminarea extremelor de preț



Figura 3.4: $\text{MSE} = 4385$. Cu eliminarea extremelor de preț

unei mașini reale, cât și poze ce conțin diverse simboluri puse la întâmplare și marcate, clasificate în 14 categorii.

Motivele principale pentru care am ales acest set de date au fost atât faptul că resursele online pentru un astfel de tip de imagini erau limitate, cât și avantajul oferit de platforma Roboflow ce oferă setul de date direct în formatul necesar prelucrării de către un model de arhitectură YOLO v5, acesta fiind și modelul utilizat de mine.

Arhitectura modelului

Așa cum am menționat mai devreme, modelul ales a fost un YOLOv5⁸. Majoritatea metodelor de detectare a obiectelor reutilizează clasificatori pentru a efectua detectarea. În schimb, YOLO este un model preantrenat ce abordează detectarea obiectelor ca o problemă de regresie, unde o rețea neurală unică este utilizată pentru a determina atât pozițiilor casetelor de delimitare, cât și probabilităților claselor de apartenență a obiectelor detectate, într-o singură evaluare [19].

YOLOv5 are mai multe configurații și arhitecturi, de exemplu YOLOv5s, YOLOv5m sau YOLOv5x. Diferența dintre aceste versiuni o constituie parametrii de scalare *depth_multiple* și *width_multiple*, ce reprezintă complexitatea și dimensiunea modelului. De asemenea, YOLO vine și cu trei fișiere YAML prestatibile (*hyp.scratch-low.yaml*, *hyp.scratch-med.yaml* și *hyp.scratch-high.yaml*) ce conțin hiperparametrii configurabili pentru antrenarea modelului. Aceste fișiere conțin setări precum rata de învățare, regularizare, parametri pentru augmentarea datelor și alți parametri ce pot influența eficiența modelului.

Antrenarea modelului

Pentru a obține rezultate cât mai bune cu acest model, l-am antrenat pe setul de date de mai devreme, experimentând cu diverse configurații de hiperparametri și arhitecturi dintre cele descrise mai sus, existente în librăria YOLO, dar și cu batch size-ul și numărul de epoci de antrenare. YOLO fiind un model preantrenat, antrenarea se realizează apelând fișierul *train.py* din librărie cu parametri doriti, astfel:

```
!python train.py --img 640 --batch 32 --epochs 100 --data {dataset.  
location}/data.yaml --cfg ./models/yolov5x.yaml --weights yolov5x.pt  
--name yolov5x_results --cache --hyp data/hyps/hyp.scratch-high.yaml  
--patience 15
```

Rezultate

În ceea ce privește rezultatele, cel mai bun model a fost YOLOv5x, având parametri existenți în fișierul *hyp.scratch-high.yaml* și a atins o precizie de 0.82 și un recall de 0.73, în timp ce rezultatele cele mai slabe au fost obținute cu YOLOv5s, cu o precizie de 0.65 și un recall de 0.63.

⁸YOLO = You Only Look Once

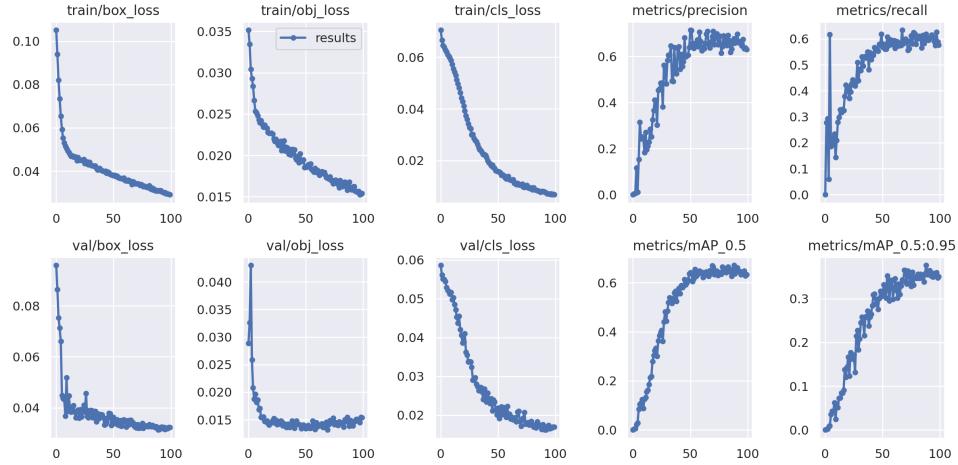


Figura 3.5: YOLOv5s

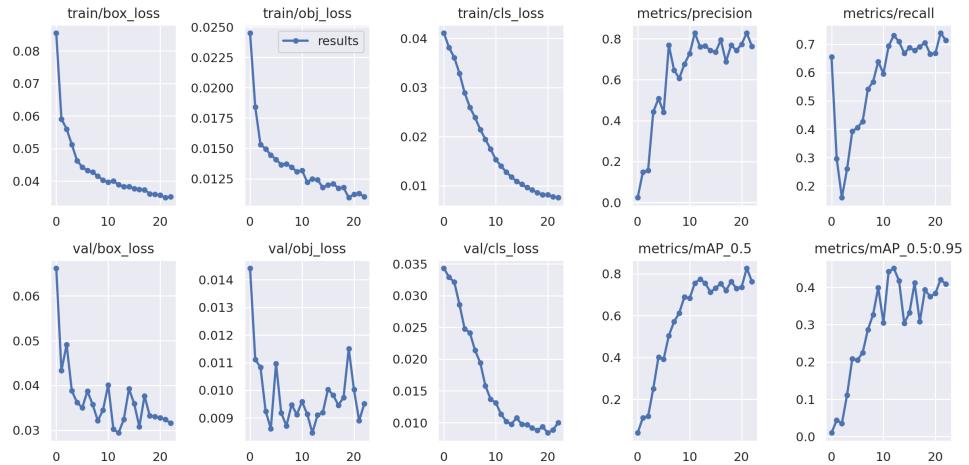


Figura 3.6: YOLOv5x

3.3.3 Integrarea modelelor în backend

După realizarea modelelor de machine learning în Python, a trebuit ca acestea să fie integrate în backend pentru a putea fi puse în aplicare în contextul aplicației. Faptul că backend-ul a fost implementat folosind Node.js, iar modelele pot fi folosite folosind librării specifice limbajului Python a reprezentat un obstacol în ceea ce privește interacțiunea dintre acestea.

Astfel, a trebuit să rulez scripturi Python direct din codul de Node.js, folosind biblioteca *child_process* pentru a crea procese copil ce execută aceste scripturi. Fiecare script este invocat folosind funcția *spawn*, ce inițiază un proces nou pentru fiecare sarcină de predicție, asigurând astfel izolarea proceselor.

Datele necesare predicțiilor sunt transmise scripturilor prin fluxurile de intrare (*stdin*) ale proceselor. Rezultatele predicțiilor sunt apoi citite din fluxurile de ieșire (*stdout*) ale acestora, iar în cazul erorilor, informațiile sunt captate din fluxul de erori (*stderr*) și sunt

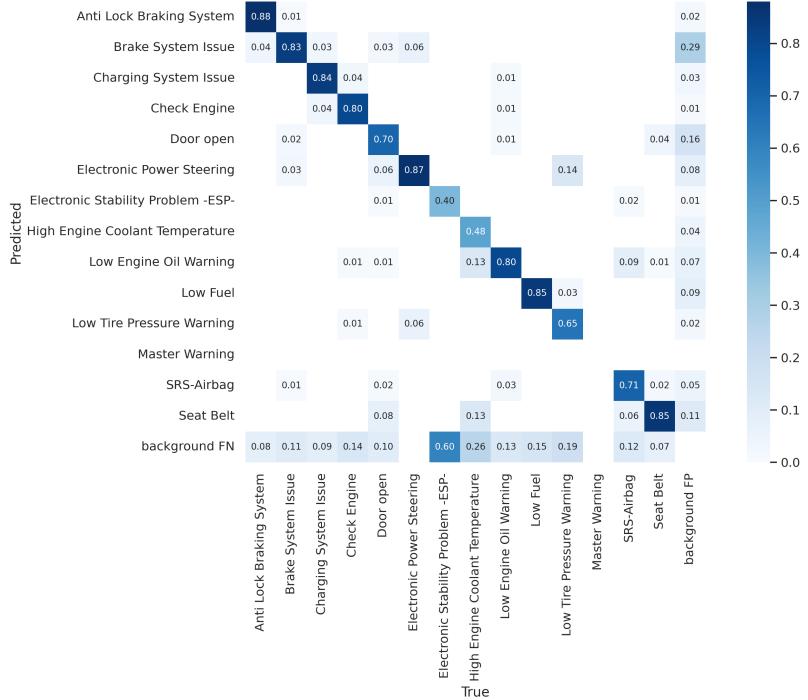


Figura 3.7: Matricea de confuzie

propagare mai departe în Node.js, acolo unde sunt gestionate corespunzător.

În ceea ce privește predicțiile de preț, scriptul Python procesează caracteristicile vehiculului și returnează un preț estimat, în timp ce pentru detectia simbolurilor, scriptul analizează imaginea furnizată și returnează detaliile simbolurilor recunoscute, dar și calea către imaginea actualizată ce conține simbolurile detectate încadrate în chenare.

Având în vedere că este inefficient din punctul de vedere al resurselor să stochez permanent pe server imaginile încărcate de utilizator pentru a recunoaște simbolurile din bord, am ales să folosesc biblioteca *multer*, un middleware pentru Node.js ce facilitează încărcarea fișierelor. *Multer* este configurat pentru a salva fișierele încărcate temporar într-un director *uploads*, folosind o funcție de stocare ce definește atât destinația cât și numele fișierului.

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, `.${Date.now()}-${file.originalname}`);
  },
});
```

Când o imagine este încărcată prin API-ul dedicat detecției de indicatori de bord, aceasta este procesată printr-un apel la funcția ce execută script-ul Python descris mai sus. Scriptul de Python returnează rezultatele, inclusiv etichetele detectate și calea către o imagine procesată care este salvată într-un director *processed_images*. După procesare, calea către imaginea procesată și etichetele sunt trimise înapoi către client, iar fișierul original este șters de pe server pentru a elibera spațiul de stocare.

Imaginiile procesate sunt accesibile prin intermediul unui endpoint static configurat în *Express*, care servește fișierele din directorul *processed_images*. Astfel, utilizatorii pot accesa imaginile prin URL-uri directe.

```
app.use('/processed_images', express.static('processed_images'));
```

Pentru a nu stoca datele mai mult decât este necesar, există un endpoint și pentru ștergerea imaginilor procesate, ce le elimină din directorul *processed_images* la cerere.

```
app.delete('/delete_image/:filename', (req, res) => {
  const filename = req.params.filename;
  const filePath = path.resolve('processed_images', filename);
  try {
    cleanupFile(filePath);
    res.send('File deleted successfully');
  } catch (error) {
    res.status(500).send('Failed to delete file');
  }
});
```

Astfel, prin acest flux de lucru, imaginile încărcate nu sunt păstrate permanent pe server, iar predictia simbolurilor din bord este făcută într-un mod eficient pentru resursele aplicației.

3.4 Cloud

Motivat de faptul că serviciul de expirare al documentelor are nevoie de verificări zilnice, am recurs la decizia de a hosta backend-ul într-un server cloud accesibil oricând, spre deosebire de a rula aplicația local, situație în care verificarea expirării documentelor se realiza doar atunci când server-ul backend era pornit.

Întrucât serviciul Elastic Beanstalk din AWS oferă un mediu simplificat, gestionând automat detaliiile infrastructurii, cum ar fi scalarea, balansarea încărcarii și monitorizarea stării aplicației, acesta a fost alegerea potrivită pentru hostarea backend-ului.

Pentru stocarea datelor am folosit Amazon RDS (Relational Database Service), mi-grând astfel baza de date locală PostgreSQL în cloud pentru a fi accesibilă din backend-ul hostat. Pe lângă beneficiul acesta, Amazon RDS oferă posibilitatea backup-urilor automate ale bazei de date, ceea ce asigură o măsură în plus de prevenție a datelor. Integrarea dintre backend și cloud în baza de date a fost realizată incluzând ambele instanțe ale lor sub același VPC⁹, pentru a comunica în aceeași rețea.

Integrarea AWS S3 pentru upload-ul de imagini în cloud

Atunci când utilizatorii introduc în aplicație informații despre asigurările mașinilor, au optiunea de a încărca și o poză cu documentul acesteia. Pentru a nu încărca baza de date cu poze stocate direct în format base64, am optat pentru folosirea serviciului AWS S3, pentru a reduce încărcătura în baza de date, unde pentru fiecare poză am stocat URL-ul către calea acesteia din bucket-ul din S3 configurat pentru a stoca pozele încărcate de utilizatori.

Upload-ul către bucket-ul din AWS S3 a fost realizată direct din partea de frontend a aplicației, folosind librăria AWS-SDK pentru JavaScript. Aceasta încorporează mai multe servicii din AWS oferind o colecție de biblioteci, care ajută la abstractizarea integrării lor în aplicație [5]. Conexiunea la AWS din frontend a fost realizată folosind un acces key id și un secret access key, accesibile din panoul de control IAM¹⁰ din AWS. Pentru a asigura faptul că fiecare poză este unică și nu apar conflicte de nume, am numit fiecare poză după data momentului în care aceasta a fost încărcată, iar pozele au fost încărcate în format .jpg.

```
import AWS from "aws-sdk";

export const configureAWS = (): void => {
    AWS.config.update({
        accessKeyId: process.env.AWS_ACCESS_KEY_ID,
        secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
    });
};
```

⁹VPC = Virtual Private Cloud

¹⁰IAM = Identity and Access Management

Capitolul 4

Prezentarea aplicației

Design-ul aplicației „Driver’s Hub” a fost realizat astfel încât fiecare componentă individuală de UI să fie un floating widget, pentru a oferi un design cat mai modern aplicației. Astfel, atât bara de navigare, cât și chenarele ce conțin detalii despre mașini, dar și widget-urile pentru documente au fost încadrate în chenare separate și bine diferențiate.

4.1 Pagina de înregistrare și logare

Atunci când utilizatorul deschide pentru prima dată aplicația, primul contact va fi cu pagina de autentificare, de unde se poate fie autentifica, fie poate alege să își creeze un cont nou prin redirecționare către pagina de înregistrare. În cazul în care autentificarea eşuează din cauza credențialelor, o alertă va fi afișată.



Figura 4.1: Autentificarea și validarea credențialelor

În pagina de înregistrare, utilizatorul trebuie să introducă numele de utilizator, adresa de email, parola și confirmarea parolei. Fiecare câmp este validat în timp real, pentru a oferi un plus de securitate datelor utilizatorului.

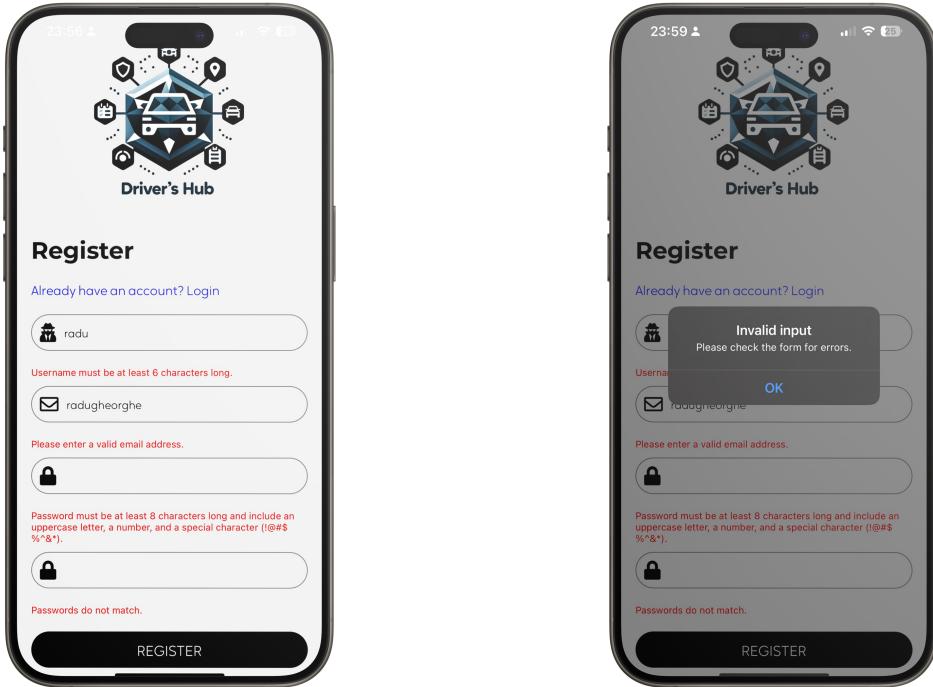


Figura 4.2: Validarea datelor de input

4.2 Panoul de control

4.2.1 Bara de navigare

Navigarea între paginile aplicației se realizează din bara de navigare situată în partea de jos a ecranului, în care există o scurtătură pentru fiecare funcționalitate. Atunci când utilizatorul navighează între paginile din bara de navigare, aceasta va rămâne fixă, pentru a facilita experiența utilizatorului și a face funcționalitățile ușor accesibile.

4.2.2 Adăugarea unui document

Pentru a adăuga un document nou pentru mașină, utilizatorul apasă pe butonul de plus din partea superioară a aplicației, afișându-se apoi opțiunile pentru tipurile de documente. Atunci când utilizatorul alege tipul dorit, un formular sub formă de modal este afișat din partea de jos a dispozitivului. Ulterior, după ce formularul este încărcat, un widget apare în panoul principal, reprezentând informații despre tipul de document ales. În cazul în care utilizatorul încearcă să adauge un document cu un tip deja existent, acesta va primi o alertă corespunzătoare.

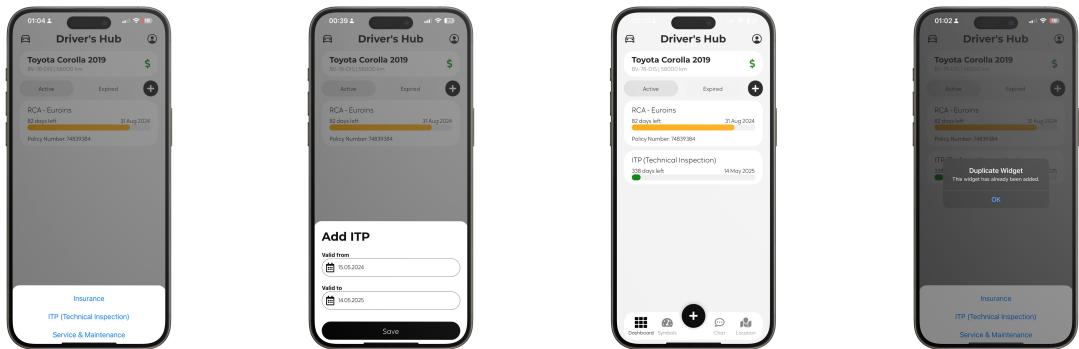


Figura 4.3: Adăugarea unui document

4.2.3 Estimarea prețului second-hand al mașinii

Tot din panoul principal de control, utilizatorul poate opta pentru estimarea prețului mașinii pe piața second-hand folosind butonul verde în forma de dolar din partea de sus a paginii, din care va apărea un pop-up cu prețul estimat.

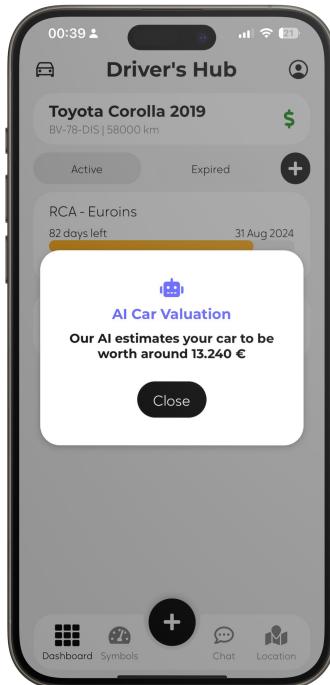


Figura 4.4: Pop-up-ul de afișare al prețului estimat

4.3 Pagina pentru detecția simbolurilor din bord

În această pagină utilizatorul poate încărca o poză cu indicatorii de eroare din bordul mașinii pentru a fi detectați și recunoscuți. Ca și răspuns, vor apărea atât poza, cu simbolurile detectate încadrate în chenare, cât și o listă cu denumirile erorilor recunoscute.

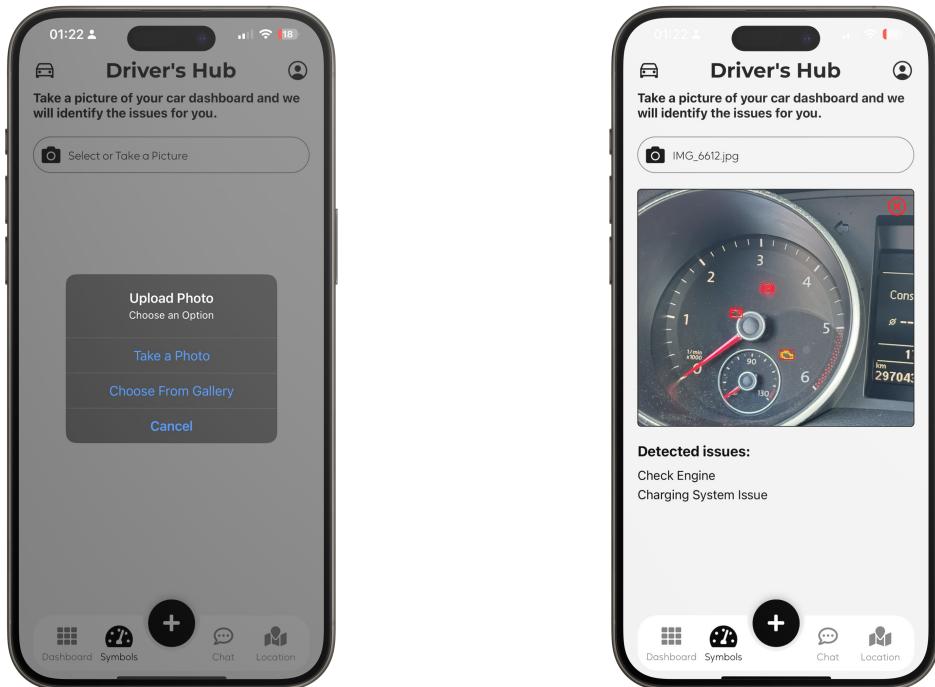


Figura 4.5: Detectarea indicatorilor din bord

4.4 Adăugarea unei noi mașini

Pentru a adăuga o mașină nouă, utilizatorul trebuie să completeze un formular ce conține date despre vehicul și care poate fi accesat din bara de navigație, interacționând cu butonul din mijlocul acesteia. Apoi, o pagină separată pentru acest formular va fi deschisă, iar pentru a naviga înapoi, utilizatorul poate folosi comanda nativă a telefonului său. De asemenea, lista de mașini a utilizatorului poate fi accesată folosind iconița din stânga sus a paginii, în formă de mașină.

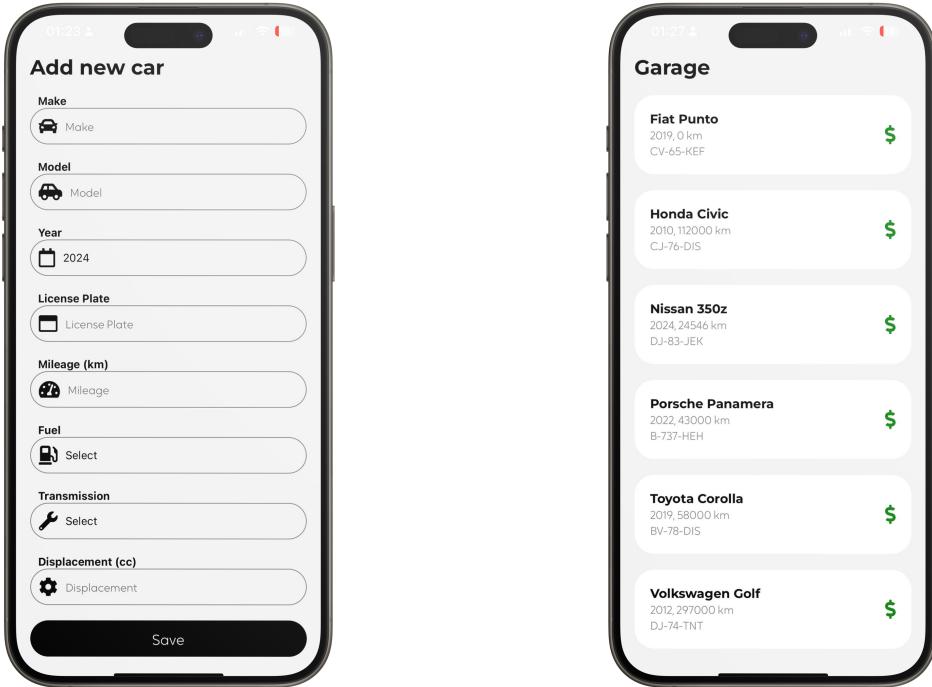


Figura 4.6: Adăugarea și accesarea mașinilor

4.5 Chatbot-ul pentru depistarea problemelor mașinii

În ceea ce privește chatbot-ul, design-ul acestuia este unul destul de simplu, constând într-o bară de introducere a mesajului și un buton, iar pentru partea de mesaje, vor apărea pe rând, folosind culori diferite, atât mesajele trimise de utilizator, cât și răspunsurile venite de la chatbot.

4.6 Pagina pentru funcționalitățile ce folosesc locația

Cele două funcționalități ce au la bază locația împart pagina, având harta direct încorporată. Pe lângă butonul de recentrare al locației, există două opțiuni în partea de sus a aplicației, fiecare având un rol separat. Astfel, butonul „*Park*” este folosit pentru a marca locul de parcare și a fi ghidat către el, iar butonul „*Services*” va ajuta utilizatorul să găsească service-urile din apropiere, direct pe hartă.

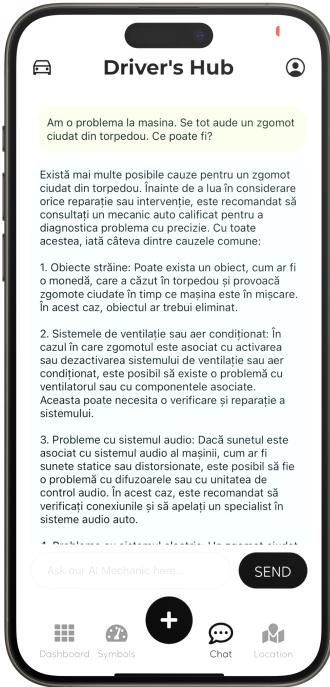


Figura 4.7: Pagina pentru Chatbot

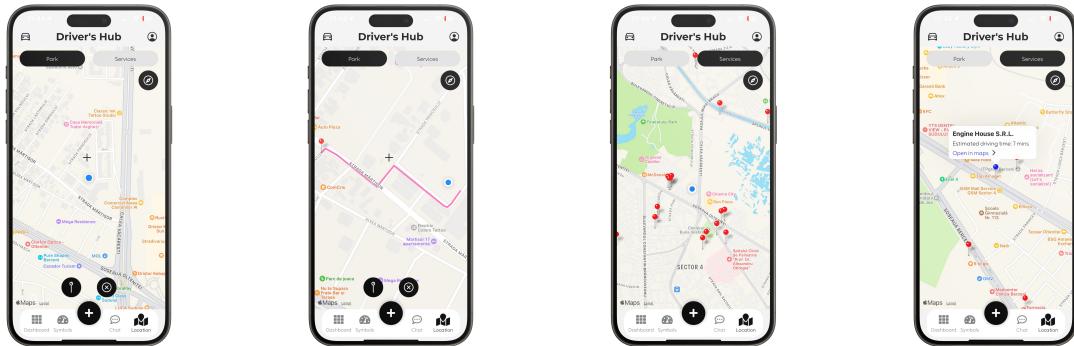


Figura 4.8: Secțiunea dedicată serviciilor de locație

4.7 Secțiunea de profil

În cadrul secțiunii de profil utilizatorul poate accesa informațiile cu privire la contul acestuia, respectiv și numărul de mașini. De asemenea, acesta are și posibilitatea de a efectua modificări profilului, de a schimba parola sau de a ieși din cont.

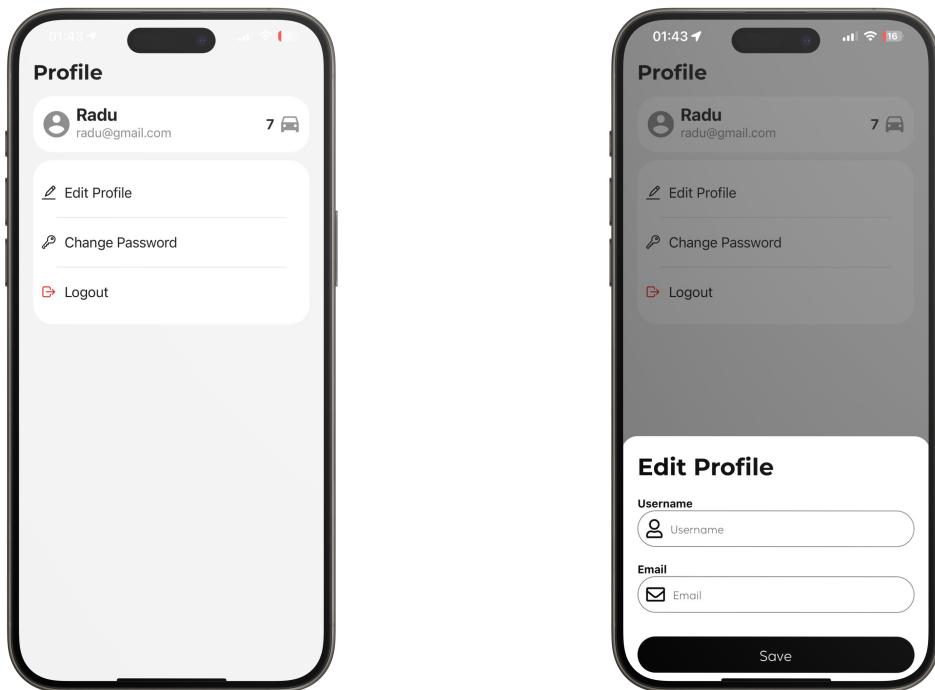


Figura 4.9: Secțiunea de profil

4.8 Notificările

Atât atunci când aplicația este deschisă, cât și atunci când este în background, utilizatorul primește notificare atunci când unul dintre documentele mașinilor sale se apropie de expirare.



Figura 4.10: Notificarea primită pe dispozitiv

Capitolul 5

Concluzii

Dezvoltarea aplicației „Driver’s Hub” a fost o experiență plăcută, dar și foarte educativă. Pe parcursul dezvoltării acesteia, am aprofundat tehnologiile cu care eram deja familiar din diverse surse, spre exemplu Node.js și AWS, dar am și învățat să folosesc React Native pentru a realiza aplicații mobile, lucru care sunt convins că îmi va folosi și pe viitor, întrucât dezvoltarea aplicațiilor mobile mă pasionează de mult timp. În plus, pe lângă lucrul individual cu aceste tehnologii, am reușit să înțeleg mai bine ce presupune crearea unei aplicații de la zero, atât în ceea ce privește arhitectura acesteia (fie că este vorba de backend, frontend sau de machine learning), tehnologiile alese, dar și importanța constanței în dezvoltarea acesteia.

Deși implementarea aplicației a fost o activitate constantă, au existat și provocări și puncte de cotitură, unul dintre cele mai dificile fiind procesul de hostare al backend-ului în AWS, întrucât a fost nevoie de configurarea unui întreg mediu cloud care să susțină aplicația. Altă provocare a constat în integrarea modelelor de machine learning în backend, fiind un proces anevoios din cauza suportului redus pentru asigurarea compatibilităților dintre cele două medii, Python și Node.js. Chiar dacă pe moment simteam că aceste dificultăți îngreunează dezvoltarea aplicației propriu-zise, acum pot spune că ele au ajutat la dezvoltarea mea, întrucât am învățat multe lucruri din aceste piedici.

Cu toate acestea, aplicația „Driver’s Hub” reprezintă un asistent de mare ajutor pentru posesorii de vehicule, atingându-și astfel scopul de a ușura experiența șoferilor atât prin informațiile utile furnizate, cât și prin soluțiile oferite la problemele întâmpinate în gestionarea și întreținerea autovehiculelor.

Listă de figuri

3.1	Arhitectura bazei de date	18
3.2	Distribuția prețurilor mașinilor din setul de date de antrenare	41
3.3	MSE = 5677. Fără eliminarea extremelor de preț	43
3.4	MSE = 4385. Cu eliminarea extremelor de preț	43
3.5	YOLOv5s	45
3.6	YOLOv5x	45
3.7	Matricea de confuzie	46
4.1	Autentificarea și validarea credențialelor	49
4.2	Validarea datelor de input	50
4.3	Adăugarea unui document	51
4.4	Pop-up-ul de afișare al prețului estimat	51
4.5	Detectarea indicatorilor din bord	52
4.6	Adăugarea și accesarea mașinilor	53
4.7	Pagina pentru Chatbot	54
4.8	Secțiunea dedicată serviciilor de locație	54
4.9	Secțiunea de profil	55
4.10	Notificarea primită pe dispozitiv	56

Bibliografie

- [1] *About npm / npm Docs*, (accesat în 11.6.2023), URL: <https://docs.npmjs.com/about-npm>.
- [2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta și Masanori Koyama, *OpTUNA: a next-generation Hyperparameter Optimization Framework*, Iul. 2019, URL: <https://arxiv.org/abs/1907.10902>.
- [3] *Amazon S3 - Cloud Object Storage - AWS*, (accesat în 11.6.2023), URL: <https://aws.amazon.com/s3/>.
- [4] auth0.com, *JWT.IO - JSON Web Tokens Introduction*, (accesat în 7.6.2023), URL: <https://jwt.io/introduction>.
- [5] *AWS SDK for JavaScript*, (accesat în 9.6.2023), URL: <https://aws.amazon.com/sdk-for-javascript/>.
- [6] Wikipedia contributors, *Node.js*, (accesat în 11.6.2023), Iun. 2024, URL: <https://en.wikipedia.org/wiki/Node.js>.
- [7] Wikipedia contributors, *React native*, (accesat în 11.6.2023), Mai 2024, URL: https://en.wikipedia.org/wiki/React_Native.
- [8] Wikipedia contributors, *XGBoost*, (accesat în 8.6.2023), Iun. 2024, URL: <https://en.wikipedia.org/wiki/XGBoost>.
- [9] *Directions API overview*, (accesat în 11.6.2023), URL: <https://developers.google.com/maps/documentation/directions/overview>.
- [10] *Distance Matrix API overview*, (accesat în 11.6.2023), URL: <https://developers.google.com/maps/documentation/distance-matrix/overview>.
- [11] Expo, *GitHub - expo/expo: An open-source framework for making universal native apps with React. Expo runs on Android, iOS, and the web.* (accesat în 11.6.2023), URL: <https://github.com/expo/expo>.
- [12] *Express - Node.js web application framework*, (accesat în 11.6.2023), URL: <https://expressjs.com/>.

- [13] *Introduction to AWS Elastic Beanstalk* (2:14), (accesat în 11.6.2023), URL: .
- [15] Dastan Maulud și Adnan Mohsin Abdulazeez, „A Review on Linear Regression Comprehensive in Machine Learning”, în *Journal of Applied Science and Technology Trends* 1 (Dec. 2020), pp. 140–147, DOI: [10.38094/jastt1457](https://doi.org/10.38094/jastt1457).
- [16] *Overview*, (accesat în 11.6.2023), URL: .
- [17] *Overview*, (accesat în 11.6.2023), URL: .
- [18] *PostgreSQL: about*, (accesat în 11.6.2023), URL: .
- [19] Joseph Redmon, Santosh Divvala, Ross Girshick și Ali Farhadi, „You Only Look Once: Unified, Real-Time Object Detection”, în *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788, DOI: [10.1109/CVPR.2016.91](https://doi.org/10.1109/CVPR.2016.91).
- [20] Ultralytics, *Home*, (accesat în 11.6.2023), Iun. 2024, URL: .
- [21] *what-is-aws*, (accesat în 11.6.2023), URL: .
- [22] *XGBoost Documentation — xgboost 2.0.3 documentation*, (accesat în 8.6.2023), URL: .