**Sistem client-server pentru managementul**

**locurilor de parcare**

|  |  |  |
| --- | --- | --- |
|  | Absolvent: | **Masterand Ing. Irina Andrada RADU** |
|  |  |  |
|  | Coordonator ştiinţific: | **Asist. Univ. Dr. Ing.**  **Lorand BOGDANFFY** |

**2021**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | |  | |
| DECAN, |  | | DIRECTOR DEPARTAMENT, | |
| **Prof. dr. ing.** |  | **Prof. dr. ing.** | |

Absolvent: **Irina Andrada RADU**

**SISTEM CLIENT-SERVER DE GESTIUNE AL LOCURILOR DE PARCARE**

1. **Enunţul temei:** *Sistem client-server de gestiune al locurilor de parcare*
2. **Conţinutul lucrării:** *Introducere, Obiectivele Proiectului, Strudiu Bibliografic, Analiză și fundamentare teoretică, Proiectare în detaliu și Implementare, Testare și Validare, Manual de instalare și utilizare, Concluzii, Bibliografie, Anexa 1.*
3. **Locul documentării**: *Universitatea Petroșani, facultatea de Inginerie Mecanică și Electrică*
4. **Consultanţi**:
5. **Data emiterii temei: 2**1 noiemvrie 2020
6. **Data predării:** 25 iunie 2021

|  |  |
| --- | --- |
| Absolvent: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |
| Coordonator ştiinţific: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Declaraţie pe proprie răspundere privind**

**autenticitatea lucrării de disertație**

Subsemnatul(a)**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**, legitimat(ă) cu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ seria \_\_\_\_\_\_\_ nr. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
CNP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, autorul lucrării \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_elaborată în vederea susţinerii examenului de finalizare a studiilor de master la Facultatea de Inginerie Mecanică șo Electrică din cadrul Universităţii Petroșani, sesiunea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a anului universitar \_\_\_\_\_\_\_\_\_\_, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

|  |  |  |
| --- | --- | --- |
| Data  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  | Nume, Prenume  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |  |
|  |  | Semnătura |

Cuprins

[Capitolul 1. Introducere – Contextul proiectului 1](#_Toc384994105)

[1.1. Contextul proiectului 1](#_Toc384994106)

[Capitolul 2. Obiectivele proiectului 2](#_Toc384994108)

[2.1. Înregistrarea conturilor utilizator și păstrarea informațiilor legate de modalitățile de plată ale clienților confidențiale 2](#_Toc384994106)

[2.2. Facilitarea unei modalități de rezervare a unui loc de parcare pentru o perioadă dată ..................... 2](#_Toc384994106)

[2.2.1. Neocuparea locului de parcare la începerea rezervării 2](file:///C:\Users\Andrada\AppData\Local\Temp\TemplateLicentaRom2018.doc#_Toc384994107)

[2.2.2. Reținerea locului de parcare la încheierea rezervării 3](file:///C:\Users\Andrada\AppData\Local\Temp\TemplateLicentaRom2018.doc#_Toc384994107)

[2.2.3. Prelungirea timpului alocat rezervării 3](file:///C:\Users\Andrada\AppData\Local\Temp\TemplateLicentaRom2018.doc#_Toc384994107)

[2.3. Facilitarea înregistrării pentru primirea unui abonament pentru un loc de parcare ...............................................................................................................................3](#_Toc384994106)

[2.4. Oferirea unei platforme care poate fi utilizată cu ușurință pentru a configura și gestiona o parcare 3](#_Toc384994106)

[Capitolul 3. Studiu bibliografic 4](#_Toc384994109)

[3.1. Smart Community Parking: sharing parking locations 4](#_Toc384994106)

[3.2. Parktopedia ..................... 5](#_Toc384994106)

[3.3. SMS Parking 6](#_Toc384994106)

[Capitolul 4. Analiză şi fundamentare teoretică](#_Toc384994110) 7

[4.1. Serverul 7](#_Toc384994106)

[4.2. Clientul WEB ..................... 1](#_Toc384994106)3

[4.3. Clientul Android 14](#_Toc384994106)

[Capitolul 5. Proiectare de detaliu si implementare 15](#_Toc384994111)

[3.1. Baza de date 1](#_Toc384994106)6

[3.2. Serverul ..................... 1](#_Toc384994106)7

[3.3. Clientul WEB 21](#_Toc384994106)

[3.1. Clientul Android 22](#_Toc384994106)

[Capitolul 6. Testare şi validare 26](#_Toc384994112)

[Capitolul 7. Ghid de utilizare 27](#_Toc384994113)

[7.1. Partea de client 27](#_Toc384994106)

[7.2. Partea de administrator 32](#_Toc384994106)

[Capitolul 8. Concluzii 34](#_Toc384994114)

[Bibliografie 35](#_Toc384994115)

[Anexa 1 Codul aplicației 36](#_Toc384994116)

# Introducere – Contextul proiectului

Proiectul propus se situează in contextul problemelor actuale cu care se confruntă în general orașele mari și mai recent chiar și cele de dimensiuni mai mici, mai precis acela al lipsei locurilor de parcare. Acest proiect se adresează persoanelor care își doresc să-și găsească un loc de parcare atunci când merg la cumpărături fără să stea cu frică deoarece mașina le-ar putea fi lovită sau ar putea fi amendați.

Orașe precum Timișoara, Cluj sau București sunt recunoscute la nivel național datorită locurilor de parcare sau mai bine spus a lipse acestora după cum descriu numeroase articole de știri precum ”Cele 7 probleme uriașe din centrul Timișoarei, cauzate de lipsa locurilor de parcare”[[1](#_Bibliografie)]. Un studiu realizat de o companie startup din Cluj a concluzionat că 80% din locurile de parcare ale parcărilor private sunt libere pe timpul nopții și ar putea fi gestionate astfel încât un profit considerabil pentru oraș[[2](#_Bibliografie)].

Proiectul propus vine cu o soluție nouă și inovatoare pentru a gestiona dificultatea pe care utilizatorii o pot întâmpina în a găsi un loc de parcare mai ales la orele de vârf.

## Contextul proiectului

Recent numărul de mașini din orașele mari a crescut și continuă să crească conform unui articol publicat în 2019, s-a înregistrat o creștere de 15.04% în iunie 2019 față de aceiași lună a anului precedent[[3](#_Bibliografie)]. Una dintre activitățile preferate de români în timpul weekend-urilor sunt vizitele la mall. De multe ori la sfârșit de săptămână parcările mall-urilor sunt pline până la refuz.

Aplicația prezentată în această lucrare propune un sistem prin care clienții își pot reține un loc de parcare din timp astfel încât să fie capabili să meargă la cumpărături fără să se fie necesar să piardă timp pentru căutarea unui loc. Sistemul vine cu posibilitatea de procesa plăți online sau chiar a închiria locuri de parcare pentru angajații mall-urilor/supermarket-urilor.

Unul din scopurile sistemul este de a implica un minim de echipament necesar (un telefon mobil) și de a permite flexibilitate în ceea ce privește anulare, amânarea programărilor. Dificultatea de a configura o parcare a fost de asemenea luată în considerare în faza de proiectare a proiectului, administratorii sistemului trebuie să ai posibilitatea de a modifica detalii legate de locurile de parcare fără să dețină un bagaj de cunoștințe foarte bogat în domeniul IT-ului.

# Obiectivele Proiectului

O problemă majoră a țării noastre o reprezintă lipsa locurilor de parcare, mai ales în zonele centrale ale orașelor mari. Proiectul de față își propune să optimizeze timpul petrecut pe locurile deja existente astfel încât timpii de așteptare din trafic să fie reduși.

Această situație este o problemă de actualitate și lucrurile par să se înrăutățească. Există numeroase proiecte care au primit alocări cu fonduri Europene pentru a gestiona aglomerația. Aceste soluții pot primi suport din domeniul IT-ului astfel încât îmbunătățirile să apară cât mai repede.

Sistemul de gestiune al locurilor de parcare are următoarele obiective:

1. Înregistrarea conturilor utilizator și păstrarea informațiilor legate de modalitățile de plată ale clienților confidențiale.
2. Facilitarea unei modalități de rezervare a unui loc de parcare pentru o perioadă dată
3. Facilitarea înregistrării pentru primirea unui abonament pentru un loc de parcare
4. Oferirea unei platforme care poate fi utilizată cu ușurință pentru a configura și gestiona o parcare

## Înregistrarea conturilor utilizator și păstrarea informațiilor legate de modalitățile de plată ale clienților confidențiale

Acest obiectiv are scopul de a garanta siguranța de utilizare a sistemului. Sistemele deja existente permit plata prin SMS pentru parcări însă proiectul de față vine cu o soluție alternativă de plată și anume cea cu cardul. Datele aferente cardurilor însă sunt confidențiale și sistemul trebuie să garanteze siguranța înregistrării unor astfel de informații în mediul online.

Există de asemenea și altă justificare pentru acest obiectiv și anume acela de a putea lua legătura cu clienții care depășesc perioada alocată pentru o rezervare. Utilizatorii au acces la informații generale precum numărul de telefon sau contul de utilizator și au posibilitatea de a raporta pe oricine încalcă regulamentul de utilizare al sistemului.

Pot fi menționate și alte avantaje precum: faptul că numărul de înmatriculare al mașinii trebuie înregistrat o singură dată și un client poate înregistra mai multe autovehicule.

## Facilitarea unei modalități de rezervare a unui loc de parcare pentru o perioadă dată

Rezervarea unui loc de parcare este funcționalitatea de bază a sistemului. O rezervare pentru un loc de parcare poate fi făcută din timp (cu până la o lună) înainte și poate fi anulată în orice moment (înainte de momentul în care începe). Ea poate fi plătită în avans sau după încheierea acesteia. De asemenea, utilizatorii se pot folosi de diferite filtre pentru a-și alege locul de parcare precum: etajul (dacă este cazul), proximitatea față de intrarea în parcare respectiv de ieșire, etc.

Acest obiectiv ridică numeroase probleme pe care sistemul trebuie să le gestioneze fără să creeze conflicte. Printre acestea se numără:

### Situația în care clienții nu își ocupă locul de parcare la momentul începerii rezervării

Soluția pentru acest scenariu este implementarea unui cronometru care să fie oprit de către utilizator în momentul în care ajunge pe locul de parcare. Acest cronometru poate fi configurat cu diferite perioade de timp. Clienții îl pot opri în orice moment, chiar dacă încă nu au ajuns la locul de parcare sau știu ca vor ajunge acolo cu o întârziere.

Dacă acest cronometru nu va fi totuși oprit de către client, sistemul va considera că locul de parcare este liber și le va permite altor utilizatori să-l rezerve.

### Reținerea locului la terminarea unei rezervări

Această problemă are o soluție similară cu cea a revendicării locului de parcare. Utilizatorii au responsabilitatea de a opri un cronometru la eliberarea locului de parcare. Sistemul permite prelungirea șederii pentru a evita alocarea aceluiași loc de parcare mai multor clienți.

Un loc de parcare nu va fi luat în considerare pentru rezervările din viitor până ce acest cronometru nu va fi oprit.

### Prelungirea pe loc a șederii pe un loc de parcare

Această situație este direct corelată cu situația menționată mai sus. S-a ales această abordare flexibilă pentru a le da libertate utilizatorilor.

## Facilitarea înregistrării pentru primirea unui abonament pentru un loc de parcare

Acest obiectiv se adresează angajaților magazinelor deoarece ei sunt cei care au nevoie de un loc de parcare stabil. Abonamentele puse la dispoziție de sistem sunt desigur pe perioade diferite de timp: permanente, anuale, de sezon sau lunare. Plata pentru aceste abonamente este gestionată de sistem. Ea poate fi făcută în mod automat la expirarea perioadei de grație banii sunt extrași din contul de utilizator și acesta este notificat de către sistem legat de această acțiune sau în mod manual, clientul va fi rugat să plătească suma restantă la momentul expirării valabilității ultimelei plăți efectuate.

## Oferirea unei platforme care poate fi utilizată cu ușurință pentru a configura și gestiona o parcare.

O platformă ușor de utilizat pentru configurarea unei parcări este un aspect important al sistemului propus. Mentenanța aplicației poate fi efectuată cu o astfel de funcționalitate de către o persoană fără cunoștințe semnificative în domeniul IT-ului.

Obiectivul le permite administratorilor să adauge locuri pentru parcare cu abonament sau să le anuleze, să vizualizeze și gestioneze abonamente din conturile de utilizator sau să verifice disponibilitatea locurilor de parcare.

Administratorul se poate folosi de platformă și cu scopul de a adăuga noi parcări. Pentru o astfel de operație el trebuie să selecteze doar numărul de nivele (cel puțin unul), numărul de zone pentru aceste nivele și numărul de locuri alocate fiecărei zone.

# Studiu Bibliografic

Problema aglomerației în trafic și implicit a absenței locurilor de parcare se situează printre prioritățile statului român. Cu această situație se confruntă primăriile orașelor mari din țară. Spre exemplu Bucureștiul este cotat ca fiind al patrulea cel mai aglomerat oraș din Europa[[4](#_Bibliografie)]conform unui studiu întocmit de compania TomTom. Traficul a crescut considerabil în ultimul deceniu iar, creșterea a fost semnificativă în special de la anul 2018 spre 2019 conform aceluiași studiu. Alte orașe cunoscute la nivel național pentru un trafic de coșmar sunt Clujul, Timișoara sau Constanța.

Lipsa locurilor de parcare poate fi considerată o cauză a acestei probleme cât și o urmare. De multe ori șoferii sunt nevoiți să conducă în plus pentru a găsi un loc liber în care să-și lase mașina. Pe de altă parte, nu sunt locuri libere datorită faptului ca este prea aglomerat.

Proiectul de față nu presupune crearea unor locuri noi ci doar optimizarea celor deja existente. Sistemul nu poate rezolva în totalitate o astfel de problemă însă poate veni cu un suport considerabil în sprijinul șoferilor.

Având în vedere situația actuală, necesitatea unor aplicații de gestiune a locurilor de parcare este evidentă. Piața deține momentan câteva aplicații care vin cu soluții pentru această problemă însă acestea nu sunt suficient de populare încât să facă o diferență observabilă. Secțiunile următoare sunt dedicate descrierii acestor aplicații precum și a diferențelor dintre acestea și sistemul propus.

## Smart Community parking: sharing parking locations

Această aplicație a fost creată de SCP2018 și este disponibilă în magazinul Play. După cum sugerează și numele *community* idea care stă la baza ei este aceea de a implica utilizatori pentru a obține o hartă detaliată a locurilor de parcare. Utilizatorii marchează un loc de parcare înainte de a-l părăsi astfel încât alți utilizatori care se află în aceeași zonă să poată să vină și să îl ocupe. Aplicația se folosește de locurile de parcare publice.

*Smart Community* implementează următoarele funcționalități de bază:

* Parcare imediată (*Immediate parking*): prin folosirea acestei funcții sistemul va căuta un loc de parcare în zona apropiată și va direcționa utilizatorul spre acesta în momentul în care un loc disponibil va fi găsit.
* Parcare întârziată (*Delayed parking)*: pentru a efectua o căutare pe o rază de acțiune mai mare. Șoferii sunt sfătuiți să apeleze la această funcție cu până la cinci minute înainte de a ajunge la destinație. Locul de parcare trebuie rezervat.
* Plecare (*Leaving now*): funcție ce trebuie apelată la părăsirea unui loc de parcare. Utilizatorii sunt rugați să aștepte câteva secunde după folosirea acestei opțiuni astfel încât să poată fi notificați dacă sunt alți utilizatori care caută locuri de parcare în zonă sau dacă altcineva a rezervat locul pe care l-au ocupat.
* Plecare imediată (*Leaving later*): funcția pereche a parcării întârziate, similar trebuie utilizată cu două – cinci minute înainte de a părăsi locul de parcare. Responsabilitatea utilizatorilor este a selecta durata întârzierii, acțiune care va lansa un cronometru. Utilizatorii vor primi o notificare dacă altcineva a rezervat locul eliberat sau vor fi informați dacă cineva a ajuns deja la acest spațiu.

Această abordare prezintă numeroase avantaje precum:

* Ușurința de utilizare.
* Posibilitatea de înregistra un loc de parcare la o locație oarecare fără să fie necesară o structură fixă.
* Este puțin probabil să apară conflicte precum nerespectarea timpului alocat de staționare.

Pe de altă parte, nu este lipsită de dezavantaje:

* Implică foarte mult utilizatorii și pentru a fi eficientă este nevoie ca sistemul să fie suficient de popular în rândurile utilizatorilor.
* Este restricționată la locurile de parcare fără plată și numeroase zone aglomerate oferă mai degrabă parcări cu plată.

Diferențele majore dintre proiectul propus și acest sistem sunt prezentate în următorul tabel.

Tabelul 3.1 *Diferențele celor două sisteme*

|  |  |
| --- | --- |
| **Proiectul propus** | **Smart Community Parking: sharing parking locations** |
| Parcare cu structură fixă (mall, supermarket) | Locuri de parcare în zone diferite |
| Parcare cu abonament | - |
| Locuri de parcare cu plată | Locuri de parcare fără plată |
| Rezervare din timp | Rezervare cu maxim 5 minute înainte |
| Plată cu cardul | - |

În concluzie sistemul propus vine cu abordare diferită față de această aplicație. Cele două ar putea fi folosite împreună.

## Parktopedia Parking

O altă aplicație ce poate fi găsită în magazinul Play, de data acesta dezvoltată de *Google Ltd.* și cu o bază de utilizare mult mai mare decât sistemul prezentat mai sus.

Numele provine din combinarea cuvintelor *parking* (parcare) și *encyclopedia* (enciclopedie) ca și *Wikipedia* dar pentru parcări. Această aplicație oferă peste 70 de milioane de locuri de parcare datorită utilizatorilor pe care îi are.

Aplicația oferă următoarele funcții:

* Găsirea unui loc de parcare pe baza unei locații sau a unei adrese.
* Obținerea de informații spre intrarea într-o parcare sau spre un loc
* Informații legate despre disponibilitatea locurilor de parcare (doar versiunea premium și doar în anumite zone).
* Informații legate de orarul, prețurile sau modalitățile de plată ale parcărilor.
* Numeroase filtre de căutare pentru simplificarea căutării.

Sistemul prezentat deține numeroase avantaje precum:

* Nu număr considerabil de locuri de parcare.
* Ușurința de utilizare.
* Numeroase informații utile pentru șoferi.

Se pot de asemenea identifica unele dezavantaje ale aplicației:

* Plata nu poate fi realizată direct din cadrul aplicației.
* Posibilitatea de a închiria un loc de parcare este imposibilă.

Sistemul propus vine cu funcționalități similare însă se adresează unei zone fixate. De asemenea proiectul dezvoltat gestionează rezervarea unui loc de parcare, funcționalitate care este inexistentă în cadrul acestei aplicații.

## SMS Parking

Aplicație dezvoltată de NACHEV Development disponibilă și ea în Play Store. Spre deosebire de sistemele prezentate mai sus, această aplicație este dedicată plății unui loc de parcare public. Sistemul detectează locația și trimite mesajul SMS corespunzător la numărul de telefon folosit.

Funcționalitățile oferite sunt următoarele:

* Profiluri multiple pentru mașină: un profil pentru fiecare mașină și trimiterea SMS-urilor prin alegerea unuia dintre ele. Mai mult, mesajul poate fi trimis în mod automat la o oră selectată de către utilizator (ora poate fi selectată cu 24 de ore înainte).
* Sugestie de zone de parcare automată: aplicația direcționează utilizatorul spre zone cu parcări pe baza locației lui curente.
* Suport pentru Dual-SIM: aplicație le permite utilizatorilor să selecteze SIM-ul de pe care vor fi trimise mesajele
* Suport pentru WEAR OS: setările din aplicație sunt accesibile de pe wearables.
* Adăugarea de zone și coduri: aplicația le permite utilizatorilor să adauge zone care neexistente împreună cu numele de telefon la care trebuie trimise SMS-urile.

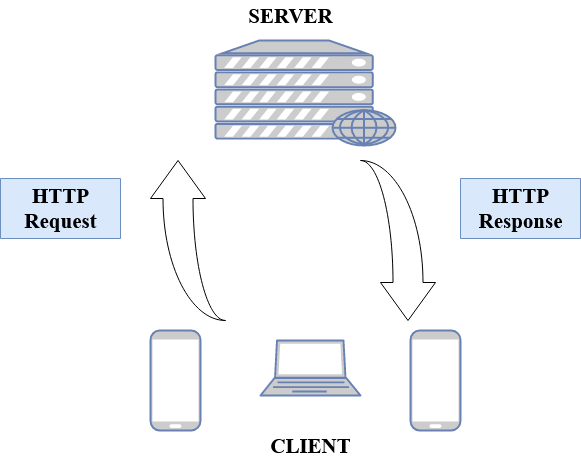
Printre principalele avantaje ale acestei aplicații se numără:

* Ușurința de utilizare
* Oferă o modalitate de plată rapidă și la care utilizatorii au acces rapid

Atât această aplicație cât și proiectul propus se adresează unor zone de parcare predefinite. Diferența majoră dintre cele două fiind doar aceia a modalității de plată. Din punct de vedere al celor două aplicații, cele două au scopuri diferite, aplicația propusă în această lucrare vrea să ofere o modalitate de a rezerva locuri de parcare în timp ce, *SMS Parking* acoperă doar problema plății.

# Analiză şi Fundamentare Teoretică

Sistemul propus este constituit din trei elemente de bază: un server, un client WEB și un client Android. Fiecare dintre aceste elemente servește un rol diferit. Figura următoare prezintă structura proiectului propus.



**Figura 4.0.1 Structura sistemului**

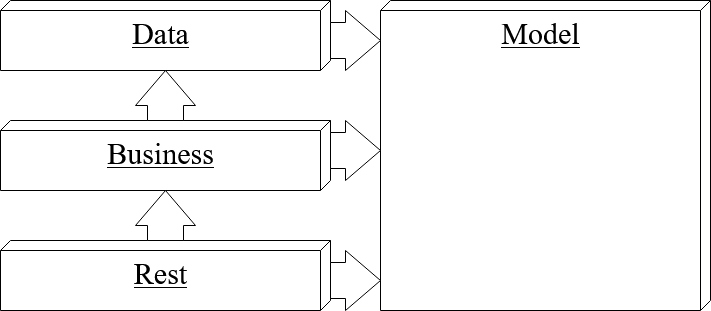
Proiectul este construit folosind o arhitectura client-server în cadrul căruia server-ul răspunde cererilor venite de la cei clienți cu resursele solicitate.

## SERVER-UL

Conform definiției, server-ul sistemului oferă servicii altor aplicații.

Acest element este cel mai important din cadrul sistemului, el fiind responsabil cu buna funcționarea aplicației. De asemenea, această componentă determină performanța și eficiența întregului sistem. Dacă server-ul este oprit sistemul nu poate funcționa.

Din punct de vedere arhitectural, s-a ales o arhitectură bazată pe nivele. Fiecare dintre acestea fiind dedicat unei funcționalități diferite. Următoarea figură prezintă structura internă a acestei componente.



**Figura 4.1.0 Structura internă a serverului**

După cum sugerează și figura 4.1.0, nivelele inferioare nu au acces la informațiile conținute în partea superioară a server-ului. Pentru o delimitare clară între nivelele aplicației fiecare dintre acestea este un proiect diferit, dependențele funcționale dintre acestea precum și dependențele pentru librăriile adiționale, s-a folosit unealta denumită *Gradle* [[5](#_Bibliografie)]*.*

*Gradle* se ocupă de gestionarea librăriilor și a framework-urilor adiționale de care depinde un proiect. S-a ales utilizarea acestui tool pentru a crește portabilitatea sistemului și pentru a simplifica problema includerii bibliotecilor ajutătoare.

Proiectele care alcătuiesc server-ul sunt scrise folosind limbajul de programare orientat pe obiect, *Java*. Pentru a putea îndeplini funcția de server, s-a folosit framework-ul *Spring Boot* [[6](#_Bibliografie)]. Cu suportul oferi de această tehnologie, aplicația poate fi creata si rulată pe un server dedicat *Tomcat*[[7](#_Bibliografie)]la adresa de IP a sistemului gazdă. Dintre framework-urile, folosite pentru a crea aplicații J2EE[[8](#_Bibliografie)], *Spring Boot* este cea mai cunoscută și folosită tehnologie.

Rolurile pe care le îndeplinește sever-ul în cadrul proiectului propus sunt următoarele:

* Comunicarea cu baza de date.
* Implementarea funcționalităților de bază.
* Efectuarea plăților în siguranță.
* Gestionarea erorilor.
* Oferirea unui punct de acces global.
* Securizarea datelor.

### Comunicarea cu baza de date

O baza de date proiectată corect determină funcționarea oricărui sistem. Acesta este fundația pe care se construiește întregul sistem. Este foarte important ca datele să fie structurate și stocate persistent în condiții optime.

În cadrul proiectului propus accesul la baza de date trebuie să se facă rapid, sistemul are nevoie de informații în timp real pentru a putea determina care locuri de parcare sunt libere în orice moment. Un alt aspect care face baza de date atât de importantă în cadrul sistemului sunt conturile de utilizator, datele clienților sunt private.

Baza de date a sistemului a fost proiectată folosind PostgreSQL[[9](#_Bibliografie)]. Detalii legate de aceasta vor fi prezentate în capitolul următor în timp ce pentru acest subcapitol ne vom axa pe descrierea conexiunii cu aceasta, la nivelul server-ului.

Layer-ul *Data* al server-ului este nivelul la care se realizează conexiunea dintre server și baza de date. Pentru legarea aplicației la o baza de date sistemul se folosește de *Hibernate*[[10](#_Bibliografie)]. Aceasta este o tehnologie extrem de cunoscută și folosită în mediile de dezvoltare software. *Hibernate* este un framework ORM (Object/Relational Mapping) care poate fi folosit pentru bazele de date relaționale.

Această tehnologie a fost aleasă din motive legate de performanță și ușurință de utilizare. Rolul acestei tehnologii este de a reprezenta entitățile prezente în baza de date la nivelul server-ului.

Maparea dintre entități și clase este o mapare 1:1 realizată folosind adnotări la nivelul clasei. Pentru a extrage coloanele și datele stocate într-un tabel la nivelul bazei de date se folosește adnotarea @Entity aparținând pachetului *javax.pesistence* . Din același pachet fac par și celălalte adnotări necesare pentru a mapa o entitate din baza de date anume: @Table, @Id și @GeneratedValue. Numele acestora fiind reprezentative pentru scopul pe care îl îndeplinesc. *Table* este folosită pentru a da numele unui tabel din baza de date, *Id* marchează identificatorul unic pe care entitatea îl are iar, *GeneratedValue* face ca o valoare să fie generată automat la nivelul bazei de date fără a fi necesară introducerea acesteia manual. *Hibernate* va știi să gestioneze această clasă la nivel de bază de date mulțumită acestor adnotări.

Maparea entităților din baza de date este insuficientă dacă acestea nu pot fi accesate. Operațiile CRUD (create, read, update și delete) sunt efectuate folosind un modul de dimensiuni mai mici aparținând de *Spring Data* și anume *Spring Data JPA* [[11](#_Bibliografie)]. În acest modul se găsesc pachetele care conțin clasele ce trebuie folosite pentru a crea DAO-urile (data access objects) ce efectuează operații de citire și scriere cu baza de date.

### Implementarea funcționalităților de bază

Această parte este gestionată la nivelul de business al server-ului. Logica de business a sistemului include următoarele operații:

* Crearea/modificarea/ștergerea conturilor
* Logarea/delogarea
* Crearea/ștergerea parcărilor
* Modificarea locurilor de parcare
* Crearea/ștergerea/modificarea abonamentelor
* Alegerea/modificarea unei modalități de plată
* Rezervarea unui loc de parcare
* Modificarea/ștergerea rezervărilor
* Raportarea utilizatorilor

Acest nivel de folosește de partea de *Model* pentru a transmite datele extrase din baza de date. În acest modul, s-a folosit librăria oferită de *Google*, *Guava*[[12](#_Bibliografie)]*.* Scopul este a defini clase *immutable* respectiv *modiafible* care să implementeze interfețe definite de utilizator. Avantajul este acela că efortul depus de programator pentru a crea clasele model este redus, el fiind nevoit doar să menționeze o interfață cu metode tip *getNumeAtribut***()** pe care să folosească adnotările predefinite de acestă librărie (*@Immutable, @Modifiable*) pentru a obține clase imutabile sau modificabile cu variabilele de clasă definite de *numeAtribut. Guava* generează automat un folder ”*generated”* în cadrul proiectului în care programatorul poate să regăsească toate clasele create de această librărie. Utilizarea acestor clase implică importarea lor prin specificarea pachetul în care a fost definită interfața.

**Crearea/modificarea/ștergerea conturilor**

Aceste operații sunt efectuate atât de către clienți cât și de către administrator. Ele au efect direct asupra bazei de date, de aceea este important ca server-ul să se asigure că după efectuarea lor, baza de date este rămâne într-o stare consistentă. Problemele pot apărea din cauza accesului neautorizat sau al accesului concurent.

**Logarea/delogarea**

Sunt două operații importante în cadrul sistemului care se reduc la o simplă interogare efectuată la nivel de bază de date asupra numelui de cont, respectiv a parolei unei utilizator.

**Crearea/ștergerea parcărilor**

Astfel de operații sunt disponibile și pot fi efectuate doar de către utilizatorii cu drepturi de administrator. O parcare creată trebuie să respecte structura elementului real pe care îl reprezintă (număr de locuri, număr de etaje, locație etc.).

**Modificarea locurilor de parcare**

Operație efectuată de către un administrator, el are dreptul de a marca un loc de parcare ca fiind loc de parcare cu abonament. În același timp tot administratorul are responsabilitatea de a marca locurile rezervabile dacă nu mai sunt folosite ca locuri cu abonament.

**Crearea/modificarea/ștergerea abonamentelor**

Atât administratorii cât și utilizatorii pot efectua astfel de operații.

**Alegerea/modificarea unei modalități de plată**

Este o funcționalitate pe care doar utilizatori o pot face. Deoarece astfel de aspecte sunt legate doar de conturile lor.

**Rezervarea unui loc de parcare**

Funcție dedicată clienților. Există numeroase erori care pot apărea la efectuarea unei rezervări. Utilizatorii trebuie sa să selecteze perioada în care își doresc rezervarea și au la dispoziție diferite filtre pentru a-și alege locul de parcare dorit. Server-ul se asigură că dacă o rezervare nu este marcată ca și CLAIMED locul este eliberat pentru a putea fi rezervat de o altă persoană. Acest lucru se întâmplă doar în cazul în care utilizatorul nu ajunge la timp pentru a-și ocupa locul. De asemenea, rezervările trebuie șterse la încheierea acestora astfel încât locurile care au fost ocupate să poată fi folosite din nou.

**Modificarea/anularea unei rezervări**

Clienții pot modifica sau anula rezervări în orice moment înainte de începerea acestora.

**Raportarea utilizatorilor**

Gestiunea neeliberării unui loc de parcare este o problemă majoră pentru un astfel de sistem. Server-ul trebuie să ofere o soluție pentru această situație.

### Efectuarea plăților în siguranță

Responsabilitatea pentru gestiunea cărților de credit ale clienților este a platformei software de procesare a plăților *Stripe* [[13](#_Bibliografie)]. *Stripe* oferă un API (application programming interface) care este ușor de integrat în proiectele software. Stripe garantează siguranța informațiilor legate de cărțile de credit ale clienților.

S-a ales această platformă dat fiind faptul de ușurința de integrare a acesteia cu restul sistemului. Obiectele *Stripe* pot fi extrase cu ușurință, ele fiind reprezentate la nivelul proiectului sub formatul unui obiect *Map* cu parametrii *String, String*.

### Gestionarea erorilor

Server-ul trebuie să ofere posibilitatea de a rezolva problemele ce pot apărea din cauza erorilor sau a utilizări malițioase. Sistemul propus poate întâmpina numeroase probleme. Mascarea erorilor pentru utilizatori este una dintre responsabilitățile server-ului.

Constrângerile impuse sunt legate de numărul de etaje al parcărilor (cel puțin un nivel), numărul de zone sau de locuri.

### Oferirea unui punct de acces global

Clienții care comunică cu server-ul au nevoie de un punct de acces pentru a putea trimite și primi date de la acesta. Serviciile care țin de nivelul de business al sistemului trebuie expuse. Pentru a putea realiza acest lucru, server-ul se folosește de *Spring Boot Starter Web* care oferă pachetele conținând adnotările necesare pentru a crea aplicații Restfull(Representational State Transfer) bazat pe transferul de resurse.

Comunicarea cu server-ul se face utilizând protocolul *HTTP (*hypertext transfer protocol).

#### HTTP

Cel mai folosit protocol de transfer al datelor, fiind protocolul implicit pentru www (*world wide web).* O cerere de acces la o resursă presupune o operație de translatare a locației folosind un server de DNS pentru transformarea adresei în adresa de IP la care se află resursa solicitată [[14](#_Bibliografie)].

Resursele pot include parametrii adiționali precum variabilele de cale (PathVariable) care pot fi specificați folosind simbolul ”?”urmat de numele parametrului și de simbolul ”**=**” urmat de valoarea acestuia (de exemplu: ?nume=Andrada). Variabilele de cale sunt utilizate pentru a specifica structuri de tipul cheie-valoare sau pentru trimiterea filtre.

O alta modalitate de includere a parametrilor adiționali în calea spre o resursă este de a folosi parametrii de cerere (RequestParameter). Ca și structură, aceștia sunt reprezentați folosind simbolul / urmat de valoarea parametrului (de exemplu: /Andrada).

Funcțiile integrate de protocol sunt: *GET, POST, DELETE*și *PUT***.**

Metoda *GET* reprezintă o cerere pentru a primi o anumită resursă. Această resursă va fi translatată la un anumit URL (uniform resource location). Folosind variabilele de cale metoda *GET* poate filtra rezultatele dintr-o listă în funcție de valoarea cheii parametrului adițional. Parametrii de cerere pot fi utilizați pentru a extrage o anumită înregistrare dintr-o listă în funcție de  valoarea specificată (de exemplu: pentru folosi un parametru de cerere pentru a extrage un utilizator care are numele specificat în valoarea parametrului).

*POST* folosit pentru a transmite informații care trebuie salvate persistent în baza de date sau care implică modificări asupra unor resurse incluse la nivelul server-ului care primește apelul. În structură, un apel al metodei *POST* poate include un corp (RequestBody) în care se află obiectul transmis.

*DELETE*se folosește pentru a-i solicita calculatorului gazdă ștergerea unei anumite resurse.

*PUT*este metoda utilizată pentru a marca operațiile de actualizare care se efectuează. *PUT*și*POST* prezintă numeroase asemănări. Funcționalitățile pe care metoda *PUT*le realizează pot fi înlocuite cu *POST*fără a produce efecte nedorite în resurselor modificate.

*Spring Boot* oferă o modalitate prin care aceste metode pot fi translatate la adresele specificate. Pentru a folosi protocolul *HTTP* în cadrul unei aplicații *Spring Boot,*trebuie parcurse următoarele etape.

* O clasă adnotată cu *@RestController* în care să fie specificată calea spre această resursă trebuie creată.
* Să definească metodele din cadrul acestei clase care sunt accesibile și punctele de acces la care pot fi regăsite.
* În funcție de tipul metodelor, se folosesc adnotările de *@DeleteMapping @GetMapping, @PostMapping* sau *@PutMapping.*

Framework-ul *Spring Boot* oferă de asemenea posibilitatea de a folosi parametrii de cerere (*@RequestParam*) și variabilele de cale (*@PathVariable)*.

Pentru variabilele de cale se folosesc acolade pentru a delimita parametrul de restul căii, spre exemplu, pentru a marca un parametru *id* se va folosi o cale *“nume-cale/{id}”* la nivelul adnotării corespunzătoare metodei iar, pentru extragerea valorii acestui parametru, în corpul metodei se va folosi *@PathVariable(“id”) tipVariabilă numeVariabilă* ceea ce permite regăsirea valorii din *id* în cadrul variabilei *numeVariabilă.*

Pentru a specifica extragerea parametrilor de cerere, *@RequestParam(“numeParam”) tipParam numeParam* trebuie inclus în lista de parametrii ai metodei apelate*.* Trebuie menționat faptul ca parametrii de cerere nu trebuie specificați cu adnotări la nivelul metodei.

Metodele *POST* și *PUT* au nevoie de adnotarea *@RequestBody* oferită de *Spring Boot Starter WEB* la nivelul listei de parametri din corpul metodei din controller.

Răspunsul server-ului la o solicitare a unei resurse se numește Response și prezentă următoarea structură: corp, HTTP status și tipul media.

Corpurile resurselor returnate de server trebuie să poată fi preluate atât de clientul WEB cât și de cel Android. Corpul răspunsurilor care vin de la server sunt obiecte *JSON* (JavaScript Object Notation). Serializarea respectiv deserializarea acestor date se face folosind librăria *fasterxml* și clasele imutabile generate folosind *Guava.* Un obiect de acest tip respectă formatul cheie-valoare.

Status-ul unei apel HTTP este un cod care reflectă dacă solicitarea făcută spre client a fost făcută cu succes sau nu: succesul este reprezentat de codul 200 și erorile cu alte coduri (404 – Resursa nu a fost găsită, 405 – metodă interzisă etc.).

Tipul media specifică tipul conținutul de răspunsul server-ului. Poate fi: text, image, audio, video sau application.

### Securizarea datelor

Integritatea datelor este esențială pentru funcționarea sistemului. Server-ul trebuie să blocheze accesul neautorizat. *Spring Security* este un modul care vine în sprijinul programatorilor pentru a gestiona aspecte legate de securitate.

S-a ales o abordare relativ simplă pentru a gestiona securizarea resurselor și anume cea a interceptorilor în combinație cu header-ele apelurilor *HTTP.*

 Astfel cererile care nu conțin un header HTTP predefinit cu o anumită valoare atribuită nu vor ajunge să acceseze resursele serverului, cererile lor fiind interceptate și blocate. Răspunsul server-ului va fi acces neautorizat incluzând un header cu codul HTTP – 500 (Unauthorized).

## Clientul WEB

Pentru a-i permite administratorului să utilizeze platforma cu ușurință partea de WEB a sistemului este dedicată funcționalităților specifice acestuia. Este mai ușor de configurat o parcare folosind un ecran mai mare decât ecranul telefonului. Importanța creări unei parcări care să reflecte realitatea este de asemenea un motiv pentru care s-a ales proiectarea și dezvoltarea unei aplicații WEB.

Clientul WEB a fost creat folosind *AngularJS*[[15](#_Bibliografie)]. Acesta fiind o tehnologie modernă, ajunsă deja la cea de-a opta versiune care se folosește de un limbaj derivat din *Javascript* și anume *TypeScript*[[16](#_Bibliografie)]*.*

Aplicațiile *Angular* sunt aplicații modulare, ale căror module sunt compuse din componente. Ca și structură acestea au un format asemănător cu clasele din limbajele de programare orientate pe obiect. Componentele sunt formate din 4 fișiere, fiecare dintre acestea având o extensie diferită și bineînțeles cu roluri diferite: .*html* (hypertext markup language) cu elementele vizuale ce pot fi observate pe ecran, *.ts* (typescript) cu logica și obiectele *Angular* care definește componenta și ce fișiere mai sunt folosite pentru această componentă, *.css* (cascading style sheet) pentru a schimba felul în care sunt reprezentate pe ecran diverse elemente conținute în fișierul *.html* și .*ts* pentru a testa componenta. Dintre aceste fișiere singurul obligatoriu fiind *.ts* cu componenta. Aceasta este adnotată cu @Component.

*Angular* se folosește de servicii pentru a putea realiza diferite funcționalități. Un serviciu este definit într-un fișier de tipul *TypeScript* și este adnotat cu @Injectable() pentru a putea fi inclus și folosit în cadrul altor componente. Un serviciu este menționat în secțiunea *providers* a unui modul, sau a unei componente pentru a putea fi folosit. Acesta trebuie inclus în constructorii componentelor folosind cuvântul predefinit *public.*

*Angular 8* este ușor de utilizat și integrează module care pot gestiona apelurile către server (modulul http). Funcționarea clientului se bazează pe această operație. Ușurința cu care se fac apelurile către server fac acest framework de dezvoltare al clienților WEB o alegere potrivită pentru sistemul propus.

Modulul *http* conține clientul care poate fi injectat în interiorul unui serviciu sau al unei componente. Ca o bună practică, se recomandă crearea unui serviciu separat care să includă acest client și utilizarea sa în componentele aplicației. Apelurile către server se pot realiza utilizând acest client iar răspunsurile pot fi extrase în clase model definite la nivelul aplicației *Angular*.

Acest framework prezintă numeroase alte avantaje printre care se numără și următoarele: ușurința de navigare între diferite pagini web, posibilitatea de a definirii clase, enumerații, interfețe sau servicii, diferite metode de conversie a datelor sau diferite modalități de a opera asupra structurilor de obiecte complexe.

*Angular* se poate folosi de biblioteci adiționale pentru a face aplicația să arate și să funcționeze cât mai bine. Dintre acestea, clientul WEB se folosește de următoarele: *Bootstrap, Angular Material, MDBoostrap* și *FontAwesome.* Toate aceste tehnologii au rolul de a ajuta la reprezentarea grafică a datelor și de a face interfața utilizator mai sugestivă și ușor de folosit.

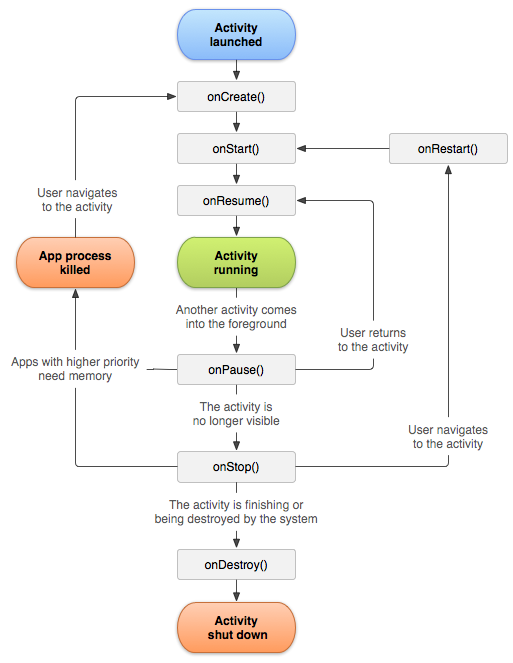
O astfel de aplicație este o resursă statică. Pentru a putea folosi acest detaliu, clientul WEB a fost împachetat și încărcat pentru a putea fi accesat fără a fi nevoia rulării aplicației folosind *Angular CLI.* Acest lucru este posibil folosind *Firebase*[[17](#_Bibliografie)].

## Clientul Android

Partea de Android a platformei este dedicată clienților. Partea corespunzătoare clienților ar fi putut fi dezvoltată folosind încă un client WEB însă s-a considerat că un client Android este mai ușor de folosit de către clienți. Mai mult, funcțiile pe care acest element le îndeplinește în cadrul sistemului sunt făcute de pe platforme mobile.

Clientul a fost creat în *Android Studio.* Programul acesta permite crearea, modificarea precum și testarea de aplicații care rulează sub sistemul de operare *Android.*

Din puncte de vedere structural, o aplicație Android este un program format din mai multe *Activități.* O activitate este o acțiune unică pe care un utilizator poate să o realizeze[[18](#_Bibliografie)]. Ciclul de viață al unei activități este predefinit: *activity launched, activity running, activity process kill* și *activity shut down.* Figura 4.3.1 ilustrează acest ciclu de viață.



**Figura 4.3.1 Ciclul de viață al activităților Android**

O activitate nu este distrusă în cazul în care trebuie să se facă o navigare spre o altă activitate ci doar va trece în starea de *onPause()* după cum se poate observa și în figura de mai sus. Aceasta este o caracteristică importantă deoarece o dată ce s-au încărcat anumite resurse pentru o activitate, nu mai este necesară reîncărcarea acestora la revenirea în activitatea respectivă.

Apelurilor către server, se fac folosind API-ului *Retrofit* [[19](#_Bibliografie)]*,* un client care poate fi utilizat în aplicațiile Android sau Java. Retrofit este o bibliotecă ce se evidențiază prin simplitatea cu care pot fi definite metodele specifice HTTP.

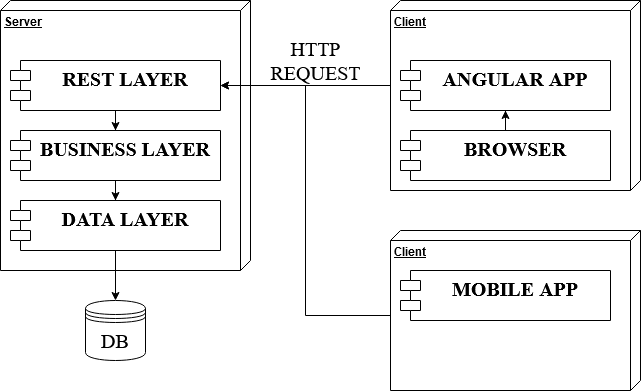
Un serviciu HTTP este o interfață care definește metode adnotate cu numele tipului de apel HTTP pe îl reprezintă: *@POST(adresaResursei), @GET, @UPDATE* sau @*DELETE*. Metodele specificate de această interfață trebuie să conțină adnotarea de @*Headers* cu definiția header-elor folosite în apel în format *JSON* iar, răspunsurile trebuie să fie din clasa *Call*<*TipObiectReturnat*>. Pentru a specifica variabilele de cale metodele au un parametru adițional adnotat cu *@Path (numeParametru) TipParametru numeParametru.* Corpurile metodelor POST și PUT necesită adăugarea unui parametru *@Body TipParametru numeParametru*  în lista de parametrii ai metodei. Orice clasă poate beneficia de interfețele definite, apelurile HTTP realizându-se prin apelarea metodelor definite în interiorul acestora.

*Gradle* este folosit pentru a crea și construi aplicați *Android*

O alternativă pentru gestionarea părții dedicate clienților poate fi înlocuirea aplicației *Android* cu o altă aplicație WEB. Dezavantajul principal al unei astfel de abordării ar fi însă acela că introduce necesitatea folosirii unui browser WEB. Problema majoră a folosirii browserelor de pe telefoanele mobile este că se pot pierde foarte ușor date relevante dacă utilizatorul nu este atent (de exemplu: apasă butonul de *Back*). Din aceste motive, pentru partea care gestionează clienții s-a ales o abordare mai sigură dar, mai puțin flexibilă în care partea care dedicată utilizatorilor este o altă aplicație *StandAlone* care trebuie instalată separat.

# Proiectare de Detaliu si Implementare

Schema generală a aplicației este prezentată în următoarea figură.



**Figura 5.0.1. Structura generală a sistemului**

În continuare vom prezenta în amănunt fiecare dintre componentele descrise în schema de mai sus.

## Baza de date

Baza de date este fundația aplicației. Toate operațiile efectuate de sistem sunt legate de acestă componentă. Schema acesteia este prezentată în următoarea figură.

Toate tabelele conțin coloanele *id, code, createdAt* și *updatedAt.* Cheia primară denumită id, o coloană de cod, momentul creării și ultima actualizare asupra obiectului.

*Memberships*

Entitatea reflectă detaliile legate de abonamentele clienților. Conține un tip de abonament și o legături spre utilizator și locul de parcare. Acestă entitate nu poate exista fără un utilizator și fără un loc de parcare asociat.

*Parkings*

Reprezintă parcarea și conține detalii precum: numele acesteia, locația, ora deschiderii și cea a închiderii dar, și costul pe care îl impune pentru o ora de staționare.

*ParkingLevels*

Etajele unei parcări, conțin o cheie străină spre *Parking* dar, și o referență spre zonele care aparțin unei astfel de entități. Un etaj nu poate exista fără o parcare.

*ParkingZones*

Următoarea subunitate din cadrul unei parcări, zonele. O zonă are o literă, o cheie străină spre nivelul de care aparține și o listă cu locurile de parcare pe care le include.

*ParkingSpots*

Cea mai mică unitate din cadrul unei parcări. Un loc de parcare are un număr precum și diferite flag-uri utilizate pentru a indica statusul pe care îl poartă un loc. Disponibilitatea este exprimată prin coloana *availability* și este setată la *true* dacă locul de parcare nu este ocupat de nicio mașină și nu este folosit ca loc de parcare cu abonament. *Rentable* este folosit pentru a marca un loc ce poate fi închiriat iar, *rented* pentru a știi care locuri de parcare sunt închiriate și care nu. Și locul de parcare păstrează o referință spre zona de parcare de care aparține.

*PaymentOptions*

Detaliile legate de plata abonamentelor. Conține un status, o dată de început și una de final.

*Reservations*

Tabelul care păstrează detaliile legate de rezervări. O rezervare este formată din următoarele câmpuri: *startTime, endTime, status, notes, vehicleLicencePlate, cost, user* și *parkingSpot.* Timpii reprezentând marginile intervalului în care va avea loc rezervarea sunt *startTime* respectiv *endTime*. Detalii legate de desfășurarea rezervării pot fi regăsite în coloanele *notes, vehicleLicencePlate* și *status.* O rezervare nu poate exista fără un utilizator și nici fără un loc de parcare.

*User*

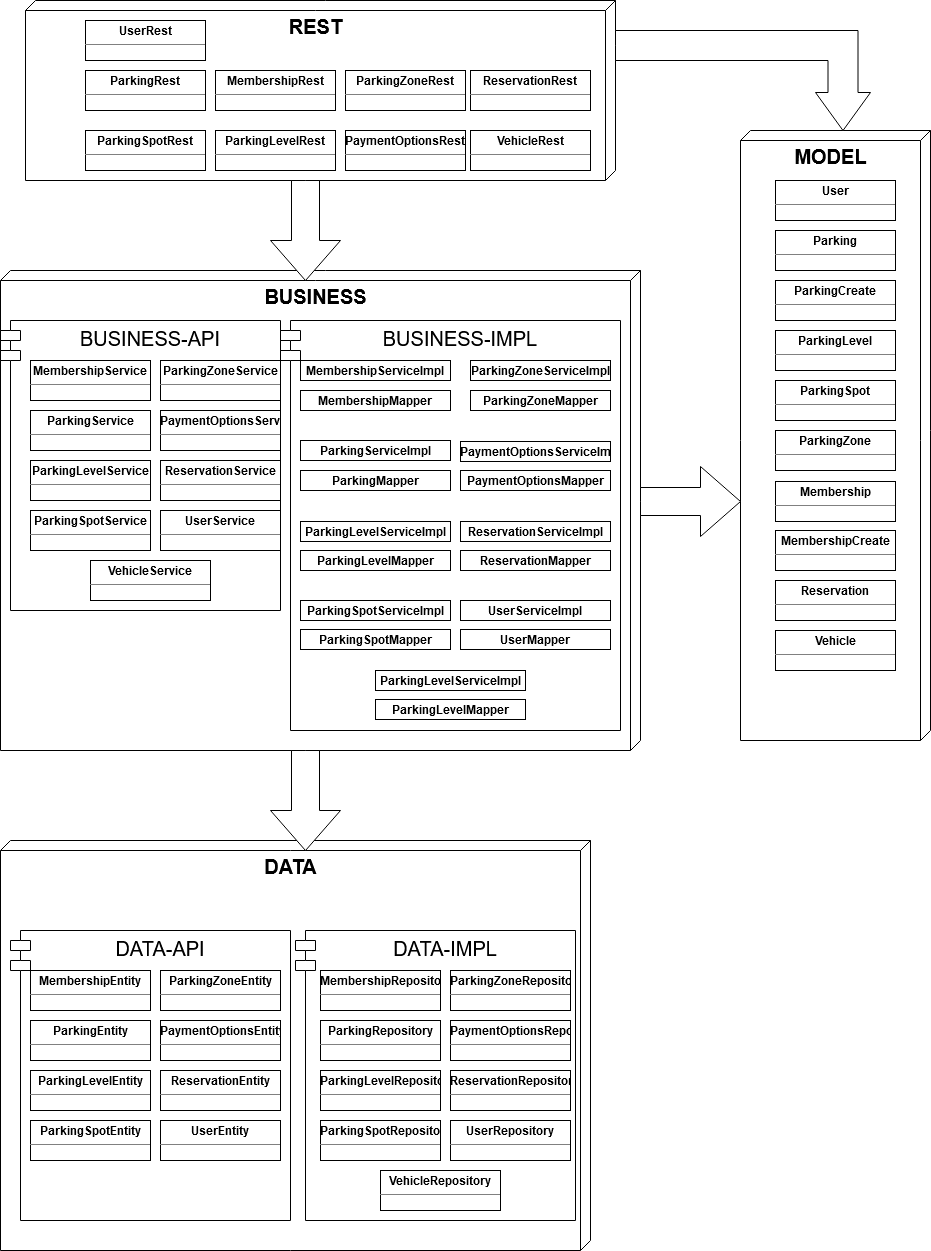
Tabela care stochează date despre utilizatori și despre administratori. Astfel de detalii sunt reprezentate de adresa de email, de nume, de numărul de telefon. Fiecare utilizator are un tip, aplicația recunoaște administratorii după tipul de utilizator *ADMIN* și clienții ca fiind *CUSTOMER.* Toți utilizatorii au nevoie de un nume de utilizator și de o parolă.

*Vehicle*

Acest tabel a fost creat pentru a oferi suport clienților de stocare a detaliilor legate de mașinile pe care le dețin. Cea mai importantă coloană a acestui tabel fiind licencePlate în care sunt salvate persistent numere de înmatriculare ale mașinilor. S-a ales separarea acestui detaliu într-o tabelă diferită pentru a le permite utilizatorilor să aibă mai multe mașini înregistrate și pentru a putea refolosi tabela *User* și pentru utilizatorii de tip *ADMIN* care nu dețin mașini.

## Serverul

Această parte din aplicație prezintă o structură precum cea prezentată în următoarea figură:



**Figura 5.2.1. Structura internă a server-ului**

Server-ul respectă conceptele arhitecturale J2EE cu server-ul partiționat în trei nivele:

* Model
* Data
* Business
* Rest

### Model

Acest proiect conține clase care sunt utilizate în cadrul aplicației la toate nivelele. Aici sunt definite modelele obiectelor *JSON* care vor fi trimise ca răspuns la cererile *HTTP* care ajung la server.

Proiectul nu conține dependențe funcționale față de niciunul din nivelele superioare. Modele conțin informațiile relevante pentru clienți server-ului. Acestea sunt definite ca interfețe și sunt create folosind *Guava* ca și clase imutabile.

Aceste clase sunt serializate și deserializate pentru a putea fi trimise respectiv primite ca și corpuri ale răspunsurilor *HTTP.*

### Data

Acest layer al aplicației este format din două proiecte: unul proiect ce conține entitățile conținute în baza de date și unul cu repositoriile ce efectuează operații asupra acestor entități.

#### Data - API

Proiectul *Java* definește clasele care reflectă structura tabelelor din baza de date. Fiecărei tabele i-a fost dedicat un pachet diferit asfel încât delimitarea dintre acestea să fie mai evidentă. *Hibernate* este responsabil cu maparea acestor clase pentru persistență.

#### Data – IMPL

Ca și dependență funcțională, acest proiect cunoaște doar proiectul data-api și se folosește de entitățile definite în acest proiect pentru a crea *DAO*-urile aplicației. S-a folosit *moștenirea* pentru a obține implementarea operațiilor CRUD astfel a fost extinsă clasa JpaRepository interfață care primește ca și parametrii clasa entității și tipul cheii primare a tabelei pe care o accesează.

Următoarea secvență de cod are rolul de a expune modul în care în care pot fi adăugate alte metode de acces la baza de date pe lângă cele de bază (findAll, findById, findOne, getOne, save, etc.).

@Repository

public interface UserRepository extends JpaRepository<UserEntity, Long> {

public Optional<UserEntity> findByUsername(String username);

public Optional<UserEntity> findByEmail(String email);

public Optional<UserEntity> findByEmailAndPassword(String email, String password);

public Optional<UserEntity> findByUsernameAndPassword(String username, String password);

}

Metodele adiționale din acestă clasă, extrag un utilizator folosind username-ul sau adresa de email.

### Business

Similar cu nivelul inferior, acest nivel este compus din două proiecte care respectă un principiu asemănător. Cele două proiecte sunt sufixate cu *api* și respectiv *impl*.

#### Business – API

Acestă parte din logica de business conține interfețele ce definesc serviciile server-ului. Dependența funcțională fiind acea la nivelul inferior (*Data)*.

Fiecare serviciu conține următoarele metode: *getById(Long id), getByCode (String code), List getList(), add(Model model), removeById (Long id), removeByCode(String code).* Bineînțeles pe lângă aceste metode mai există și metode suplimentare în serviciile care au nevoie de acestea.

Acest proiect definește de asemenea clasa de eroare ce este aruncată în cazul în care apar încălcări are constrângerilor impuse de sistem.

#### Business - IMPL

Acesta este probabil cel mai important proiect din cadrul sistemului. Felul în care sunt tratate erorile și gestionate operațiile solicitate de către utilizator dau performanța și eficiența sistemului.

Pe lângă implementarea servicilor definite în proiectul *API*, fiecare entitate are definit un *Mapper* cu scopul de a transforma un obiect de tipul *Entity* din baza de date într-un obiect de tipul *Model* care este serializat și deserializat la nivelul superior.

Fiecare *mapper* implementează metodele *toEntity, toModel* și *toModelList.*

### Rest

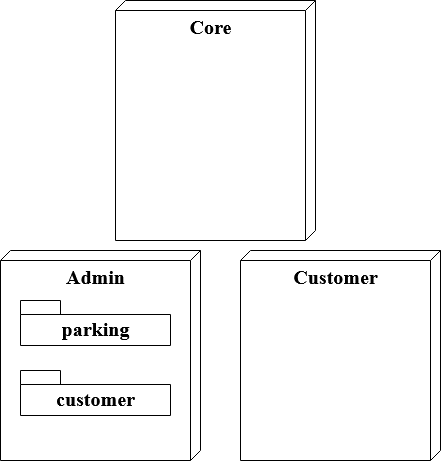
Nivelul care comunică cu clienții. Ca ți structură, acest nivel este structurat în mai multe pachete, fiecare corespunzând entității pe care o expune ca resursă. Clasele incluse la acest nivel sunt controllerele aplicației. Pe lângă controller-ele *REST* proiectul conține o clasă de configurări în care sunt definite *Bean*-urile *Spring* care vor fi injectate în controllere.

O altă caracteristică specifică este dependența spre *Spring Starter Web* pentru a obține adnotările și server-ul *TomCat* pe care se lansează aplicația. Metoda *main* a serverlui este inclusă în acest proiect.

Fișierul de configurări al aplicației este definit în cadrul acestui layer. Pe lângă caracteristicile specifice *Spring* (conexiunea la baza de date, port-ul, configurările pentru *tomcat)* acest fișier leagă aplicația de *Stripe* și are definită o proprietatea care dictează timpul de așteptare până la expirarea unei rezervări în cazul în care locul nu a fost revendicat.

## Clientul WEB

Clientul WEB este organizat în trei module principale. Acestea sunt core, admin și customer după cum prezintă și figura 5.3.1.



**Figura 5.3.1 Structura clientului WEB**

### CORE

Acest modul a fost creat pentru a încorpora toate dependențele de care au nevoie celălalte module din cadrul aplicației. Aici sunt localizate inițializările pentru librăriile adiționale utilizate în reprezentarea grafică a componentelor sistemului. Modulul integrează componenta header care conține meniul aplicației.

Alte resurse importante definite la acest nivel sunt serviciul de autentificare precum și serviciul *REST* responsabil cu trimiterea cererilor *HTTP* spre server.

Trebuie menționat fișierul model în care au fost definite clasele obiectelor *JavaScript* care să reflecte caracteristicile claselor din server.

### ADMIN

Modulul admin este format din două părți *parking* și *customer.*

#### Parking

Această secțiune este dedicată operațiilor pe care administratorii le pot efectua asupra parcărilor. Astfel de funcționalități includ: vizualizare, editare și creare. Fiecare dintre aceste acțiuni are dedicată câte o componentă *AngularJS* diferită.

#### Customer

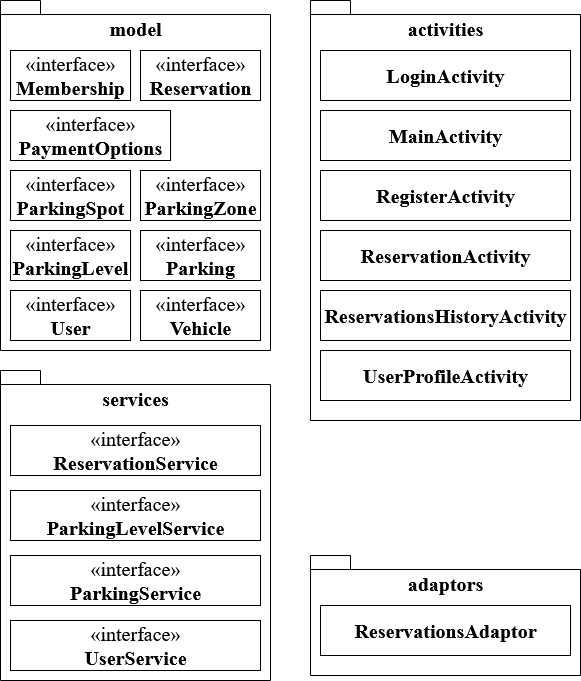
Partea aceasta a modulului le permite utilizatorilor de tip ADMIN să efectueze operații asupra utilizatorilor aplicației. Astfel de operații sunt: vizualizarea, ștergerea/modificarea de abonamente, crearea de utilizatori. Funcționalitățile menționate anterior au fost integrate în cadrul sistemului folosind componente diferite *AngularJS.*

### CUSTOMER

Parte dedicată acțiunilor ce pot fi efectuate de către utilizatori. Acest modul are scopul de a constitui mai degrabă un schelet pentru dezvoltările ulterioare, momentan incluzând doar posibilitatea de a adăuga utilizatori.

## Clientul Android

Partea de mobile este organizată în patru pachete de bază, după cum se poate observa și în figura 5.4.1.



**Figura 5.4.1**

### Model

Clasele din acest pachet sunt identice cu cele din proiectul model de la nivelul serverului. Mai mult, s-a folosit *Guava* și adnotările aferente pachetului *Immutables* pentru a obține obiecte imutabile.

### Services

Acest pachet conține serviciile utilizate pentru a primi date de la server. Pentru a obține aceste resurse s-a folosit biblioteca *Retrofit*. Toate serviciile incluse în acest pachet extind un serviciu de bază.

public interface BaseService<T> {

static final String CONTENT\_TYPE = "Content-Type:application/json";

static final String AUTHORIZATION = "X-Authorization: user";

static final String ID\_PATH = "/id/{id}";

static final String CODE\_PATH = "/code/{code}";

@GET(ID\_PATH)

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> getById(@Path("id") final Long id);

@GET(CODE\_PATH)

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> getByCode(@Path("code") final String code);

@GET

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<List<T>> getAll();

@POST

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> create(@Body final T body);

@PUT

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> update(@Body final T body);

@DELETE(ID\_PATH)

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> deleteById(@Path ("id") final Long id);

@DELETE(CODE\_PATH)

@Headers({CONTENT\_TYPE, AUTHORIZATION})

Call<T> deleteByCode(@Path ("code") final String code);

}

Folosind parametrii generici s-a obținut un serviciu ce poate fi utilizat prin extinderea acestei clase și adăugarea metodelor specifice. Metodele predefinite în toate serviciile sunt: *getById, getByCode, getAll, create, update, deleteById* sau *deleteByCode*.

### Adaptors

În acest pachete sunt conținute clasele ce definesc obiecte afișate pe ecran sub formă de liste. Pentru a obține astfel de obiecte trebuie extinsă clasa ArrayAdaptor din pachetul android widgets. Metoda *getView* trebuie suprascrisă astfel încât elementele incluse în listă să respecte forma claselor *Java*.

### Activities

Acest pachet include toate activitățile aplicației mobile. Fiecare clasă reprezintă o funcționalitate diferită a sistemului.

Spre exemplu *LoginActivity* este activitatea dedicată login-ului unui client. Sistemul primește numele de utilizator și parola, returnând un utilizator null în cazul în care acestea nu sunt găsite în baza de date.

*MainActivity* este activitatea folosită pentru a–l anunța pe utilizator când are loc următoarea rezervare pe care o are programată. Clienții pot urmări un contor care îi anunță exact cât timp mai au până la începerea rezervării. Atâta timp cât minutele acestui contor nu au ajuns la 15, utilizator poate să–și anuleze rezervarea. Ulterior butonul pentru revendicarea acesteia va putea fi apăsat. Aplicația continuă cu un alt cronometru, de dat acesta pentru a-l ține la curent pe utilizator cu momentul în care îi expiră rezervarea, momentul în care acesta ajunge la 15 minute, utilizator poate elibera locul de parcare apăsând butonul care apare pe ecran ceea ce va elibera locul de parcare.

*RegisterActivity* conține ecranul responsabil cu înregistrarea unui nou utilizator.

*ReservationActivity,* clasa care este folosită pentru a crea o nouă rezervare.

*ReservationHistoryActivity* dispune de o listă și de un *ArrayAdaptor* pentru a afișa istoricul rezervărilor unui utilizator. Aici clienții pot vedea detalii legate de rezervările precedente.

*UserProfileActivity* este dedicată schimbărilor de pe care utilizatorul le poate efectua asupra detaliilor din profilul său precum: modalitatea de plată, email-ul, numărul de telefon.

# Testare şi Validare

Funcționalitățile sistemului au fost testate manual.

Având în vedere importanța funcționalității corecte a serverului s-a folosit librăria de testare Junit 4[[20](#_Bibliografie)] pentru a efectua teste unitare la nivelul bazei de date precum și la cel al logicii de business.

Automat sunt testate toate funționalitățiile sistemului.

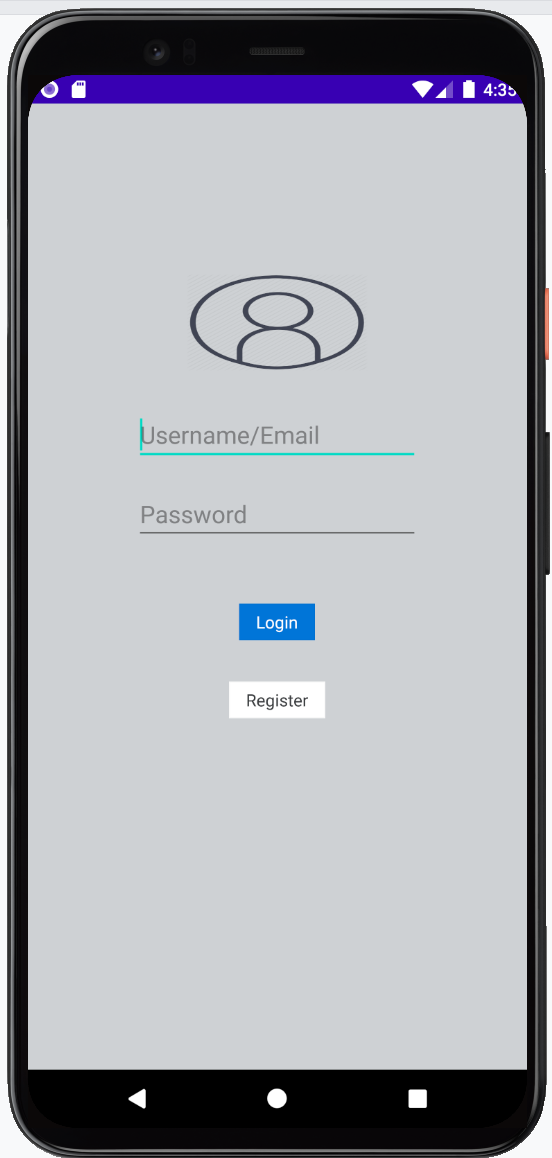
Baza de date asupra căreia se efectuează testele este o bază de date *H2*[[21](#_Bibliografie)] *inMemory*.

Testele se află într-un proiect diferit, unde pachetele conținând codul sunt denumite după proiectele asupra cărora acționează.

# Ghid de utilizare

## Partea de client

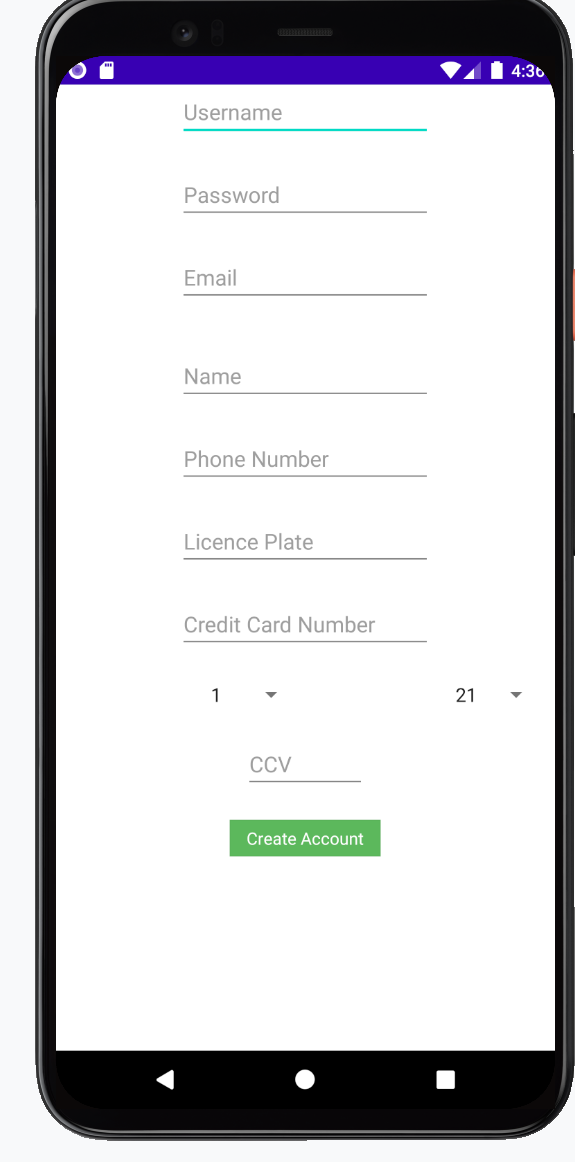
Pagina de login este pagina de deschidere a aplicației, bineînțeles login-ul este necesar doar dacă aplicația a fost omorâtă (conform ciclului de viață al unei aplicații Android). Acestă pagină este prezentată în următoarea imagine.



**Figura 7.1.1 Pagina de login**

Sistemul conține un mecanism de specificare al erorilor care pot apărea în această etapă (nume de utilizator sau parolă invalide). Apăsarea butonului de login face o cerere spre *server* pentru a valida existența utilizatorilor în baza de date. Acestă cerere de tip *POST* este plasată la nivelul *Controller*-ui pentru *User* din sistem. Acesta este cel mai înalt nivel al aplicației și poate accesa nivelul de servicii unde se afla clasa responsabilă cu verificarea validității user-ului introdus. Serviciul apelează *Repository-ul* care accesează tabela din *User* din baza de date cu informațiile utilizator.

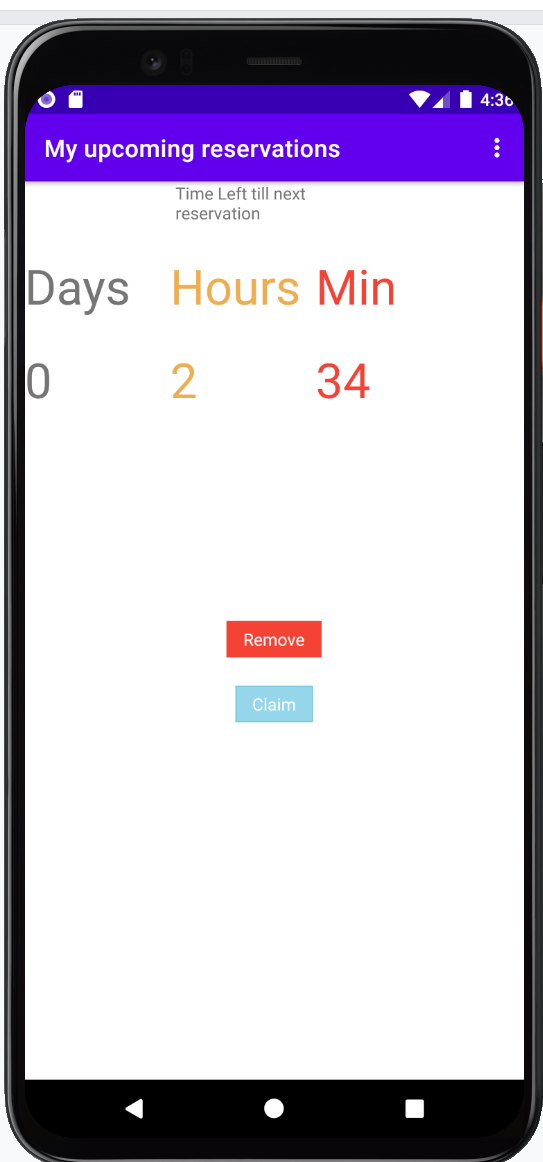
O altă acțiune valabilă din acest ecran este aceea de a crea un utilizator. Butonul de *Register* va deschide o altă activitate. Acest ecran va permite introducerea de date utilizator precum nume, prenume, email, date legate de cardul de credit,etc.



**Figura 7.1.2 Register**

Completarea formularului poate fi încheiată prin apăsarea butonului *Create Account*. Clientul *Android* va trimite un *request* de tip *POST* pentru *Controller-ul* de utilizatori cu datele înscrise în formularul de mai sus. Controller-ul va apela nivelul de business unde se află serviciul pentru utilizatori. Se verifică unicitatea numelui de utilizator precum și cea a email-ului iar, dacă acestea nu-i sunt alocate unui alt client, serviciul va salva datele înscrise în baza de date. Parola va fi codificată folosind algoritmul *PBKDF2WithHmacSHA1* [[22](#_Bibliografie)], pentru a asigura imposibilitatea de a sparge baza de date de către utilizatorii malițioși. Acest algoritm este considerat a fi sigur. Utilizatorii vor fi întorși la pagina de login unde își pot folosi contul nou creat.

După un login cu succes, aplicația se prezintă în felul descris de imagina de mai jos.

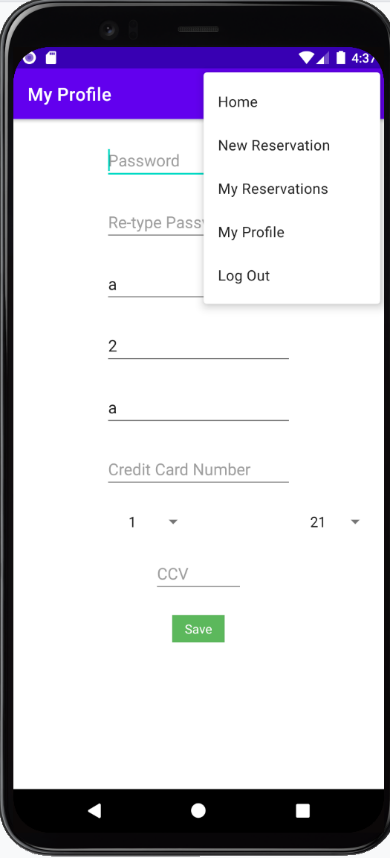


**Figura 7.1.3 Rezervările mele**

Ecranul prezintă următoarea rezervare pe care o are utilizatorul. Elementele dispuse pe ecran sunt actualizate în funcție cu trecerea timpului. De exemplu dacă mai sunt doar 15 minute până la începerea rezervării contorul pentru *Days* va fi setat la 0, la fel și cel pentru *Hours* iar, cel pentru *Min* va indica 15. Este de menționat că timpul de la care o rezervare poate fi confirmată poate fi configurat. De exemplu, aceasta configurație este setată default la 15 minute dar, administratorii au posibilitatea de a schimba asta. Așadar, butonul de *Remove* este activ până ce timpul rămas este 15 minute. Dacă acest buton este apăsat clientul va solicita o cerere de *DELETE* pentru rezervarea curentă. Această cerere va ajunge pe *Controller* pentru *Reservation* unde va fi gestionată de către serviciul responsabil cu această entitate. Dacă mai sunt doar 15 minute până ce rezervarea trebuie să înceapă, butonul *Claim* va deveni disponibil. Apăsarea acestui buton constă în trimiterea unei cereri de schimbare al statusului rezervării, metodă *PUT* la nivelul *Controller-ui* pentru rezervării. Tot acestă acțiune va face ce statusul locului de parcare care a fost alocat pentru această rezervare își va schimba *flag-ul* de free în valoarea *booleană* de *false.*

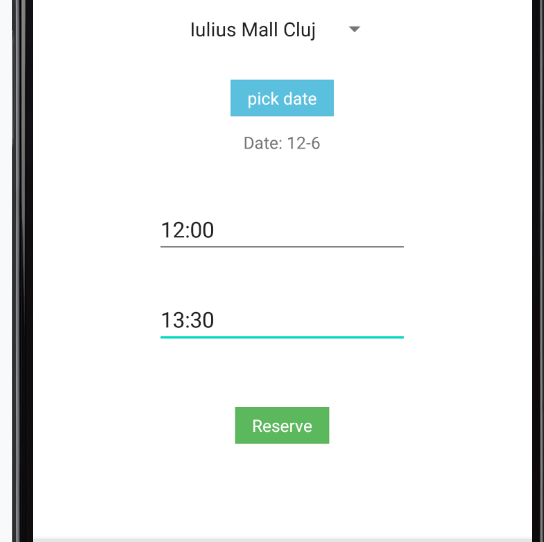
La expirarea rezervării, butonul de eliberare al locului de parcare va apărea pe ecran. Utilizatorii trebuie să se ocupe de acestă etapă pentru ca sistemul să se poată asigura că locul de parcare este liber. Apăsarea acestui buton constă în schimbarea flag-ului *free* al locului de parcare în *true* iar, la nivelul rezervării pentru setarea statusului în *FINISHED.*

Meniul din dreapta sus le permite clienților să acceseze celălalte parți ale aplicației,



**Figura 7.1.4 Meniul aplicației**

O altă activitate importantă a clientului este de a crea o rezervare. Acesta corespunde următorului ecran.



**Figura 7 .1.5. Crearea unei rezervări**

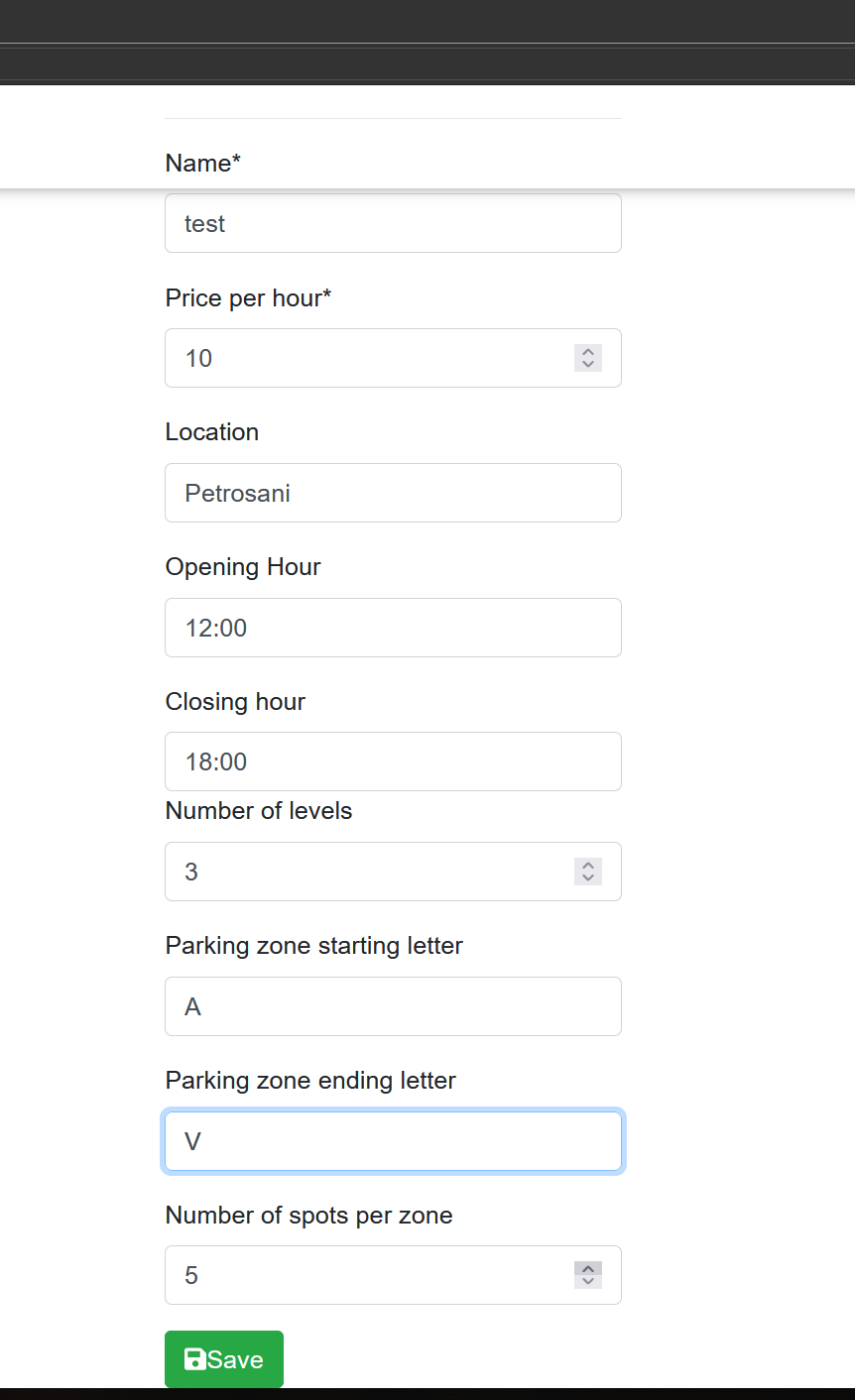
Din *dropdown* clienții vor selecta parcarea . Apoi pagina conține un *datepicker* pentru data la care trebuie făcută rezervarea și orele între care se va petrece aceasta. Butonul de *Reserve* va trimite o cerere de tip *POST* spre *endpoint-ul* de creare a unei rezervării din controller-ul serverlui. Găsirea locului de parcare, dacă există un loc de parcare liber în acel interval de timp îi revine serviciului de rezervare. Acest va căuta un loc de parcare liber în acel interval de timp și va verifica dacă există vreo rezervare făcută pentru acel loc al cărui timp sa se suprapună cu orele introduse de client. Va salva datele în baza de date și utilizatorul va primi confirmarea rezervării. Nou creata rezervare poate fi văzută în pagina cu istoricul/rezervările pe care le are un client.

Clienții își pot modifica datele din *My Profile*. Acest ecran este similar cu cel pentru înregistrarea unui cont de utilizator nou.

## Partea de administrator

Administratorii au o pagina de login diferit față de cea de utilizatori iar, sistemul nu permite adăugarea de utilizatori de acest tip. Un administrator poate fi adăugat doar de către un alt administrator.

Una dintre cele mai importate acțiuni ale un astfel de utilizator este acea de a configura o parcare. Sistemul pune la dispoziția administratorilor următorul ecran.

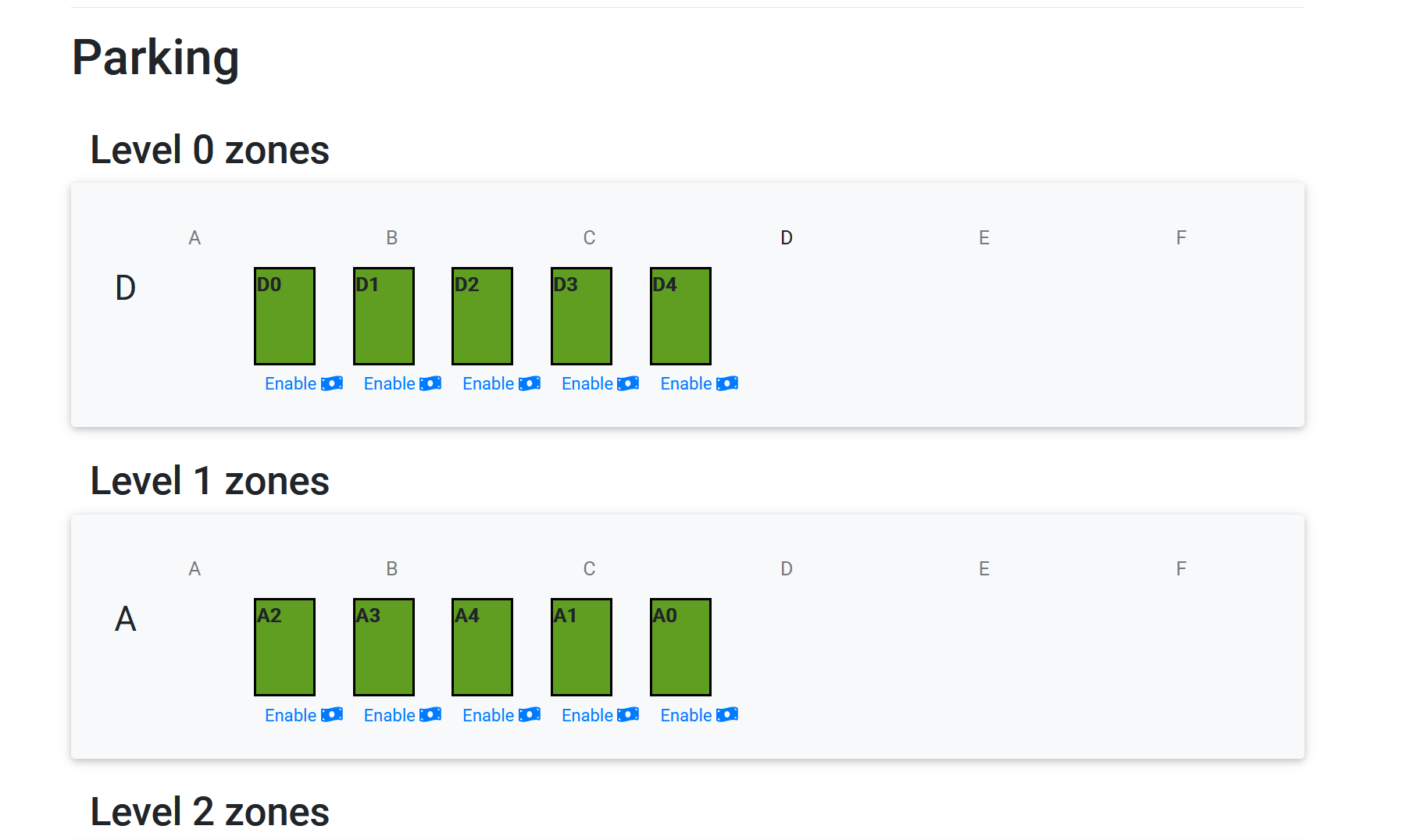


**Figura 7.2.1 Adăugarea unei parcări**

După cum se poate observa din imagine există numeroase configurări pe care administratorii le pot face. Aceștia aleg numele parcării, îi adaugă locația, selectează ora de deschidere precum și cea de închidere și poate cele mai importante setări, selectarea numărului de nivele și cea a numărului de zone. Zonele parcărilor sunt marcare cu litere astfel, administratorii selectează litera de început, care ar trebui de ce mai multe ori să fie *A* și litera finală, numărul de zone este egal cu numărul de litere de la litera de început până la cea de final. Setarea numărului de locuri per zonă va determina numărul de locuri disponibile.

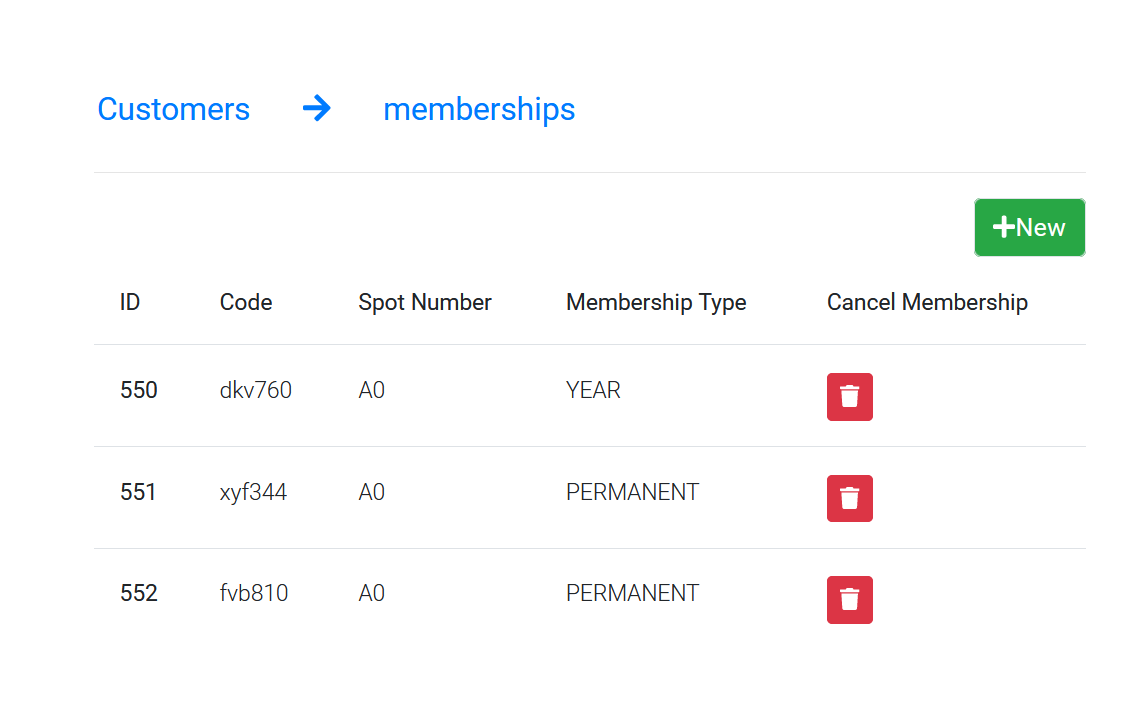
Butonul *Save* va trimite un *request* de tip *POST* spre server cu *body-ul* configurărilor alese. Această cerere este gestionată de *controller*-ul destinat parcărilor. La nivelul serviciului, se va petrece o operație de inserare în baza de date. Acestă operație va salva o intrare în tabela *Parking,* în tabela *ParkingLevel* se vor salva numărul aferent de nivele din corpul cererii, în tabela *ParkingZone* se vor salva toate zonele selectate iar, în *ParkingSpot* se vor salva  *z \* p* intrări unde *z* – *numărul de zone* și *p – numărul de nivele*. Locurile de parcare vor avea un nume format din litera zonei și a numărului locului în acea zona, de exemplu *A0, B5, C4* etc.

Parcările pot fi vizualizate și administratorii pot observa disponibilitatea locurilor existente.



**Figura 7.2.2 Vizualizarea unei parcări**

Administratorii pot apăsa pe litera unei zone pentru a cerceta statusul locurilor de parcare. În figura de mai sus, un administrator privește zona *D* a nivelului 0 și zona *A* al primului nivel. Butoanele de *Enable* trimit o cerere de tip *PUT* pentru locul de parcare și setează *flag-ul* *Rentable* la valoarea pozitivă astfel încât aceste locuri să poată fi închiriate de utilizatori.



**Figura 7.2.3 Abonamentele utilizatorilor**

Administratorii pot șterge sau adăuga abonamente pentru utilizatori. Această operație constă în selectarea tipului de abonament pentru *un an, permament, lunar* etc. și cea a nivelului unde se dorește un loc de parcare. *Controller*-ul pentru *Membership* va primi un *request* de tip *POST* pentru a crea abonamentul. Salvarea datelor îi revine serviciului *Membership* care va crea o intrare în tabela *Membership* și va actualiza statusul locului de parcare.

# Concluzii

Sistemul propus vine în sprijinul conducătorilor auto și al locuitorilor orașelor mari (în special) și introduce o modalitate inovativă de gestionare al uneia dintre problemele de actualitate. Aplicația este ușor de folosit și necesită resurse minime.

Dezvoltări ulterioare:

* Introducerea unei activității de crearea a abonamentelor pentru partea de client.
* Crearea unei interfețe WEB pentru clienți și adăugarea unei modalități de vizualizare al locurilor disponibile într-o parcare
* O mai bună gestiune în cazul în care apar probleme legate de reținerea locurilor de parcare
* Îmbunătățirea interfețelor grafice

# Bibliografie

[1] Opinia Timișoarei <https://www.opiniatimisoarei.ro/cele-7-probleme-uriase-din-centrul-timisoarei-cauzate-de-lipsa-locurilor-de-parcare/16/04/2018>

[2] Actual De Cluj, <https://actualdecluj.ro/un-startup-din-cluj-face-studiu-pe-locurile-de-parcare-private-concluzia-8-din-10-stau-goale-noaptea-ar-putea-aduce-profit-de-mii-de-euro>

[3] Piața auto, <https://www.superspeed.tv/piata-auto-romania-crestere-de-20-fata-de-2018-volkswagen-pierde-teren>

[4] ZF, [https://www.zf.ro/eveniment/bucurestiul-a-ajuns-al-treilea-cel-mai-aglomerat-oras-din-lume-din-punctul-de-vedere-al-traficului-rutier-seara-capitala-romaniei-domina-detasat-topul-european-17638222 /](http://www.tutorialspoint.com/ajax/).

[5] Gradle, <https://gradle.org> .

[6] Spring Boot ,<https://spring.io/projects/spring-boot> .

[7] Tomcat, <http://tomcat.apache.org> .

[8] Oracle J2EE Platforms, <https://www.oracle.com/java/technologies/appmodel.html>.

[9] Postgres, <https://www.postgresql.org> .

[10] Hibernate, <https://hibernate.org> .

[11] Strping JPA. <https://spring.io/projects/spring-data-jpa> .

[12] Google GuavaTutorial, <https://www.tutorialspoint.com/guava/index.htm> .

[13] Stripe, <https://stripe.com/en-ee> .

[14] Wikipedia, protocolul de transfer HTTP , <https://ro.wikipedia.org/wiki/Hypertext_Transfer_Protocol> .

[15] AngularJS tutorial, <https://angularjs.org> .

[16] TypeScript, <https://www.typescriptlang.org> .

[17] Firebase, <https://firebase.google.com>.

[18] Android, <https://developer.android.com/reference/android/app/Activity> .

[19] Retrofit Tutorial, <https://www.vogella.com/tutorials/Retrofit/article.html> .

[20] Junit 4 Tutorial, <https://www.vogella.com/tutorials/JUnit4/article.html>

[21] H2database, <https://www.h2database.com/html/main.html>

[22] Wikipedia PBKDF2, <https://en.wikipedia.org/wiki/PBKDF2>

# Anexa 1

1. **SERVER-UL**
   1. Nivelul ***DATA***
      1. Modelul unei entități

@Data  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
@Entity(name = "reservations")  
public class ReservationEntity implements Serializable{  
 */\*\*  
 \*   
 \*/* private static final long *serialVersionUID* = 1L;  
   
 */\*\*  
 \* identifier for the entity  
 \*/* @Id  
 @GeneratedValue   
 private Long id;  
   
 */\*\*  
 \* common fields  
 \*/* @Embedded  
 private BaseEntity base;  
   
 */\*\*  
 \* start time   
 \*/* private Instant startTime;  
   
  
 */\*\*  
 \* end time   
 \*/* private Instant endTime;  
  
 */\*\*  
 \* additional info for the reservation  
 \*/* private String notes;  
  
 */\*\*  
 \* status for the reservation   
 \*/* private ReservationStatus reservationStatus;  
   
  
 */\*\*  
 \* registration number of the vehicle that will be parked   
 \*/* private String vehicleLicencePlate;  
   
 */\*\*  
 \* Cost of staying  
 \*/* private double cost;  
   
 */\*\*  
 \* the user  
 \*/* @ManyToOne (fetch = FetchType.*LAZY*)  
 private UserEntity user;  
  
   
 */\*\*  
 \* parking spot  
 \*/* @ManyToOne(fetch = FetchType.*LAZY*)  
 private ParkingSpotEntity parkingSpot;  
  
 @Override  
 public String toString() {  
 return "ReservationEntity{" +  
 "id=" + id +  
 ", startTime=" + startTime +  
 ", endTime=" + endTime +  
 ", reservationStatus=" + reservationStatus +  
 ", cost=" + cost +  
 ", user=" + user +  
 '}';  
 }  
}

* + 1. Modelul unui repository

@Repository  
public interface UserRepository extends JpaRepository<UserEntity, Long> {  
  
 Optional<UserEntity> findByUsername(String username);  
  
 Optional<UserEntity> findByEmail(String email);  
  
 Optional<UserEntity> findByEmailAndPassword(String email, String password);  
  
 Optional<UserEntity> findByUsernameAndPassword(String username, String password);  
  
}

* 1. Nivelul ***BUSINESS***
     1. Modelul unei interfețe

*/\*\*  
 \** ***@author*** *Andrada  
 \* Service to handle reservation entities  
 \*/*public interface ReservationService extends BaseService<Reservation> {  
   
 */\*\*  
 \** ***@param*** *username the username of the user  
 \** ***@return*** *the list of all the reservations he had  
 \*   
 \*/* List<Reservation> findAllForUserByUsername(final String username);  
  
  
 */\*\*  
 \** ***@param*** *reservationCreate the details for the reservation to be created  
 \** ***@return*** *the created reservation  
 \*/* Reservation createReservation(final ReservationCreate reservationCreate);  
  
 */\*\*  
 \** ***@param*** *username the user for which we search for the next reservation  
 \** ***@return*** *the next reservation of the user with the requested username  
 \*/* ReservationNext getReservationNext(final String username);  
  
 */\*\*  
 \** ***@param*** *reservationId the reservation that needs to be claimed  
 \** ***@return*** *the updated value  
 \*/* Reservation claim(final Long reservationId);  
  
  
 */\*\*  
 \** ***@param*** *reservationId the id of the reservation  
 \** ***@return*** *the completed reservation  
 \*/* Reservation complete(final Long reservationId);  
  
  
 */\*\*  
 \** ***@param*** *reservationId the id of the reservation  
 \** ***@return*** *the started reservation  
 \*/* Reservation start(Long reservationId);  
}

* + 1. Implementarea unui serviciu

*/\*\*  
 \** ***@author*** *Andrada  
 \* Business level logic implementation for reservations  
 \*/*@Service  
@Slf4j  
public class ReservationServiceImpl implements ReservationService {  
  
 @Value("${reservation.claim.allocated.time}")  
 private Integer RESERVATION\_EXPIRATION\_TIME; // 15 minutes  
  
 private final ReservationValidator reservationValidator;  
  
 private final ReservationRepository reservationRepo;  
  
 private final UserRepository userRepo;  
  
 private final ParkingSpotRepository parkingSpotRepo;  
  
 public ReservationServiceImpl(ReservationValidator reservationValidator, ReservationRepository reservationRepo, UserRepository userRepo, ParkingSpotRepository parkingSpotRepo) {  
 this.reservationValidator = reservationValidator;  
 this.reservationRepo = reservationRepo;  
 this.userRepo = userRepo;  
 this.parkingSpotRepo = parkingSpotRepo;  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation getById(final Long reservationId) {  
 return ReservationMapper.*toReservation*(reservationRepo.getOne(reservationId));  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation getByCode(final String reservationCode) {  
 return null;  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public List<Reservation> getList() {  
 return ReservationMapper.*toReservationList*(reservationRepo.findAll());  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation add(final Reservation reservation) {  
 final ReservationEntity entity = ReservationMapper.*toReservationEntity*(reservation);  
 final ReservationEntity savedEntity = reservationRepo.save(entity);  
 return ReservationMapper.*toReservation*(savedEntity);  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation update(final Reservation reservation) {  
 final ReservationEntity entity = ReservationMapper.*toReservationEntity*(reservation);  
 final ReservationEntity savedEntity = reservationRepo.save(entity);  
 return ReservationMapper.*toReservation*(savedEntity);  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation removeById(final Long reservationId) throws BusinessException {  
 final ReservationEntity entity = reservationRepo.getOne(reservationId);  
 if (entity == null) {  
 throw new BusinessException("Reservation does not exist");  
 }  
 reservationRepo.deleteById(reservationId);  
 return ReservationMapper.*toReservation*(entity);  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation removeByCode(final String reservationCode) {  
 // *TODO Auto-generated method stub* return null;  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public List<Reservation> findAllForUserByUsername(final String username) {  
 return ReservationMapper.*toReservationList*(reservationRepo.findAllByUserUsername(username));  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public Reservation createReservation(final ReservationCreate reservationCreate) {  
 // find the user for the reservation if none throw exception  
 final UserEntity ue = userRepo.findByUsername(reservationCreate.getUsername())  
 .orElseThrow(BusinessException::new);  
  
 // find all available parking spots by parking name  
 final List<ParkingSpotEntity> parkingSpots = parkingSpotRepo.findAllAvailableByParkingName(reservationCreate.getParkingName());  
 final ReservationEntity re = ReservationEntity.*builder*()  
 .user(ue)  
 .startTime(Instant.*parse*(reservationCreate.getStartTime()))  
 .endTime(Instant.*parse*(reservationCreate.getEndTime()))  
 .reservationStatus(ReservationStatus.*PENDING*)  
 .build();  
  
  
 // validate parking spot availability during the selected time  
 re.setParkingSpot(parkingSpots.stream()  
 .filter(x -> reservationValidator.isValidReservation(x.getId(), re.getStartTime(), re.getEndTime()))  
 .findFirst().orElseThrow(BusinessException::new));  
  
 //save reservation  
 // set cost as price per hour \* reservation duration  
 re.setCost(*computeCost*(re.getEndTime(), re.getStartTime(), re.getParkingSpot().getParkingZone().getParkingLevel().getParking().getPricePerHour()));  
 final ReservationEntity savedEntity = reservationRepo.save(re);  
 startCountdownTimer(savedEntity);  
 return ReservationMapper.*toReservation*(savedEntity);  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public ReservationNext getReservationNext(final String username) {  
  
 // all upcoming reservations have the pending status  
 final List<ReservationEntity> reservations = reservationRepo.findAllByUserUsernameAndReservationStatus(username, ReservationStatus.*PENDING*);  
  
 // find the closest upcoming reservation  
 final Optional<ReservationEntity> re = reservations.stream().sorted((r1, r2) -> r1.getStartTime().isBefore(r2.getStartTime()) ? 1 : -1)  
 .findFirst();  
 ReservationNext reservationNext = null;  
 if (re.isPresent()) {  
 reservationNext = *getReservationNext*(re.get());  
 }  
  
 return reservationNext;  
 }  
  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 @Transactional  
 public Reservation claim(final Long reservationId) {  
 return updateReservation(reservationId, ReservationStatus.*CLAIMED*, false);  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 @Transactional  
 public Reservation start(final Long reservationId) {  
 return updateReservation(reservationId, ReservationStatus.*ONGOING*, false);  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 @Transactional  
 public Reservation complete(final Long reservationId) {  
 return updateReservation(reservationId, ReservationStatus.*COMPLETED*, true);  
 }  
  
  
 */\*\*  
 \** ***@param*** *reservationEntity the reservation  
 \*/* private void startCountdownTimer(final ReservationEntity reservationEntity) {  
 TimerTask timerTask = new TimerTask() {  
 @Override  
 public void run() {  
 try {  
 ReservationEntity re = reservationRepo.findById(reservationEntity.getId())  
 .orElseThrow(BusinessException::new);  
 re.setReservationStatus(ReservationStatus.*UNCLAIMED*);  
 re.setParkingSpot(reservationEntity.getParkingSpot());  
 reservationRepo.save(re);  
 claimReservation(re);  
 } catch (Exception e) {  
 *log*.info("Failed to make reservation in unclaimed {}", e);  
 }  
 }  
 };  
  
 // schedule the status to change to unclaimed when the reservation is starting  
 // note Instant.now() gives the current time - 3h  
 long duration = Math.*abs*(Duration.*between*(reservationEntity.getStartTime(),  
 Instant.*now*()).toMinutes()) - 180;  
 *log*.info("Reservation status changing to UNCLAIMED in : {} minutes", duration);  
 new Timer().schedule(timerTask, duration \* 60000);  
 }  
  
  
 */\*\*  
 \** ***@param*** *reservationEntity unclaimed entity. If by the end of the timer expiration this is not claimed than remove the reservation  
 \*/* private void claimReservation(final ReservationEntity reservationEntity) {  
 TimerTask timerTask = new TimerTask() {  
 @Override  
 public void run() {  
 try {  
 if (reservationEntity.getReservationStatus().equals(ReservationStatus.*UNCLAIMED*)) {  
 reservationRepo.delete(reservationEntity);  
 }  
 } catch (Exception e) {  
 *log*.error("Failed to remove unclaimed reservation: {}", reservationEntity.getId(), e);  
 }  
 }  
 };  
  
 // schedule a timer to check if the status is changed to claimed if not just remove the reservation  
 long duration = Math.*abs*(Duration.*between*(reservationEntity.getStartTime().plusSeconds(RESERVATION\_EXPIRATION\_TIME),  
 reservationEntity.getStartTime()).toMinutes());  
 *log*.info("Reservation will be deleted in : {} minutes", duration);  
 new Timer().schedule(timerTask, duration \* 60000);  
 }  
  
 private Reservation updateReservation(final Long reservationId, final ReservationStatus status, final boolean available) {  
 ReservationEntity re = reservationRepo.findById(reservationId).orElseThrow(BusinessException::new);  
 re.setReservationStatus(status);  
 reservationRepo.save(re);  
  
 //make spot occupied  
 ParkingSpotEntity pse = parkingSpotRepo.findById(re.getParkingSpot().getId())  
 .orElseThrow(BusinessException::new);  
 pse.setAvailable(available);  
 parkingSpotRepo.save(pse);  
 return ReservationMapper.*toReservation*(re);  
 }  
  
 private static Double computeCost(final Instant startTime, final Instant endTime, final double pricePerHour) {  
 final long duration = Duration.*between*(endTime, startTime).toMinutes();  
 return pricePerHour \* duration / 60;  
 }  
  
 private static ReservationNext getReservationNext(final ReservationEntity entity) {  
 // compute times for count down timers  
 LocalDateTime startTimeLDT = LocalDateTime.*ofInstant*(entity.getStartTime(), ZoneOffset.*UTC*);  
 LocalDateTime currentTimeLDT = LocalDateTime.*ofInstant*(Instant.*now*(), ZoneOffset.*UTC*);  
 LocalDateTime endTimeLDT = LocalDateTime.*ofInstant*(entity.getEndTime(), ZoneOffset.*UTC*);  
  
  
 final Integer daysTillReservation = startTimeLDT.getDayOfYear() - currentTimeLDT.getDayOfYear() - currentTimeLDT.getHour() > startTimeLDT.getHour() ? 1 : 0;  
 final Integer hoursTillReservation = startTimeLDT.getHour() - currentTimeLDT.getHour() - 3;  
 final Integer minutesTillReservation = startTimeLDT.getMinute() - currentTimeLDT.getMinute();  
  
  
 final Integer reservationDurationHours = Math.*abs*(endTimeLDT.getHour() - startTimeLDT.getHour());  
 final Integer reservationDurationMinutes = Math.*abs*(endTimeLDT.getMinute() - startTimeLDT.getMinute());  
  
  
 return ReservationNext.*builder*()  
 .days(daysTillReservation)  
 .hours(*correctNegativeTimes*(hoursTillReservation, 24))  
 .minutes(*correctNegativeTimes*(minutesTillReservation, 60))  
 .reservationId(entity.getId())  
 .durationHours(reservationDurationHours)  
 .durationMinutes(reservationDurationMinutes)  
 .reservationStatus(entity.getReservationStatus())  
 .build();  
 }  
  
 private static int correctNegativeTimes(final int time, final int correction) {  
 if (time < 0) {  
 return time + correction;  
 } else {  
 return time;  
 }  
 }  
  
  
}

* + 1. Modelul unui validator

*/\*\*  
 \** ***@author*** *Andrada  
 \*/*@Service  
public class ReservationValidatorImpl implements ReservationValidator {  
  
 private final ReservationRepository reservationRepo;  
  
 public ReservationValidatorImpl(ReservationRepository reservationRepo) {  
 this.reservationRepo = reservationRepo;  
 }  
  
 */\*\*  
 \* {****@inheritDoc****}  
 \*/* @Override  
 public boolean isValidReservation(final Long parkingSpotId, final Instant startTime, final Instant endTime) {  
 boolean valid = true;  
 final List<ReservationEntity> reservationsOnSpot =   
 reservationRepo.findAllByParkingSpotId(parkingSpotId);  
 for (ReservationEntity oldReservation : reservationsOnSpot) {  
  
 if (oldReservation.getReservationStatus().equals(ReservationStatus.*COMPLETED*)) {  
 break;  
 }  
   
 /\*   
 \* Reservation 1: |||||||||||||||||||||  
 \* Reservation 2: ||||||||||||||||||||  
 \*/  
 if (oldReservation.getStartTime().isBefore(startTime) &&  
 oldReservation.getEndTime().isBefore(endTime))  
 {  
 valid = false;  
 break;  
 }  
  
 /\*   
 \* Reservation 1: |||||||||||||||||||||||||||||||||||||||  
 \* Reservation 2: ||||||||||||||||||||  
 \*/  
 if (oldReservation.getStartTime().isBefore(startTime) &&  
 oldReservation.getEndTime().isAfter(endTime)) {  
 valid = false;  
 break;  
 }  
   
  
 /\*   
 \* Reservation 1: |||||  
 \* Reservation 2: ||||||||||||||||||||  
 \*/  
 if (oldReservation.getStartTime().isAfter(startTime) &&  
 oldReservation.getEndTime().isBefore(endTime)) {  
 valid = false;  
 break;  
 }  
   
  
 /\*   
 \* Reservation 1: |||||||||||||||||||||  
 \* Reservation 2: ||||||||||||||||||||  
 \*/  
 if(oldReservation.getStartTime().isAfter(startTime) &&  
 oldReservation.getStartTime().isBefore(endTime)) {  
 valid = false;  
 break;  
 }  
   
 }  
   
 return valid;  
 }  
  
}

* 1. Nivelul ***REST***
     1. Controller-ul generic

@Slf4j  
public class BaseController<T> {  
  
 private static final String *CODE\_PATH* = "/code/{code}";  
 private static final String *ID\_PATH* = "/id/{id}";  
 private static final String *LIST\_PATH* = "/list";  
  
 protected BaseService<T> service;  
 protected static final String *USER\_USERNAME\_PATH* = "/user/{username}";  
  
  
 public BaseService<T> getService() {  
 return service;  
 }  
  
 public void setService(BaseService<T> service) {  
 this.service = service;  
 }  
  
 */\*\*  
 \** ***@param*** *code unique for each entity  
 \** ***@return*** *the entity with the corresponding code  
 \*/* @GetMapping(path = *CODE\_PATH*)  
 public ResponseEntity<T> get(@PathVariable final String code) {  
 *log*.info("REST request to GET entity by code: {}", code);  
 final T entity = service.getByCode(code);  
 if (entity == null) {  
 *log*.info("Entity with code: {} does not exist", code);  
 return ResponseEntity.*notFound*().build();  
 } else {  
 return ResponseEntity.*ok*(entity);  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *id of the entity  
 \** ***@return*** *the entity with the corresponding id  
 \*/* @GetMapping(path = *ID\_PATH*)  
 public ResponseEntity<T> get(@PathVariable final Long id) {  
 *log*.info("REST request to GET entity by id: {}", id);  
 final T entity = service.getById(id);  
 if (entity == null) {  
 *log*.info("entity with id: {} does not exist", id);  
 return ResponseEntity.*notFound*().build();  
 } else {  
 return ResponseEntity.*ok*(entity);  
 }  
 }  
  
  
 */\*\*  
 \** ***@return*** *the list with all the entities  
 \*/* @GetMapping(*LIST\_PATH*)  
 public ResponseEntity<List<T>> gets() {  
 *log*.info("REST request to GET all entities");  
 final List<T> entityList = service.getList();  
 if (entityList == null) {  
 *log*.info("No entities found");  
 return ResponseEntity.*notFound*().build();  
 } else {  
 *log*.info("Entities found : {}", entityList.toString());  
 return ResponseEntity.*ok*(entityList);  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *entity the entity to be added  
 \** ***@return*** *the created entity  
 \*/* @PostMapping  
 @Transactional  
 public ResponseEntity<T> post(@RequestBody final T entity) {  
 *log*.info("REST request to POST entity : {}", entity);  
 final T created;  
 try {  
 created = service.add(entity);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong creating the entity : {}", entity);  
 return null;  
 }  
 return ResponseEntity.*ok*(created);  
 }  
  
  
 */\*\*  
 \** ***@param*** *entity the entity to be added  
 \** ***@return*** *the created entity  
 \*/* @PutMapping  
 @Transactional  
 public ResponseEntity<T> put(@RequestBody final T entity) {  
 *log*.info("REST request to PUT entity : {}", entity);  
 final T updated;  
 try {  
 updated = service.update(entity);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong updating the entity: {}", entity);  
 return null;  
 }  
 return ResponseEntity.*ok*(updated);  
 }  
  
  
 */\*\*  
 \** ***@param*** *id the identifier for the entity to be deleted  
 \** ***@return*** *the deleted entity  
 \*/* @DeleteMapping(path = *ID\_PATH*)  
 @Transactional  
 public ResponseEntity<T> delete(@PathVariable final Long id) {  
 *log*.info("REST request to DELETE entity with id : {}", id);  
 final T deleted;  
 try {  
 deleted = service.removeById(id);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong deleting the entity with the id: {}", id);  
 return null;  
 }  
 return ResponseEntity.*ok*(deleted);  
 }  
  
 */\*\*  
 \** ***@param*** *code the unique code for the entity to be deleted  
 \** ***@return*** *the deleted entity  
 \*/* @DeleteMapping(path = *CODE\_PATH*)  
 @Transactional  
 public ResponseEntity<T> delete(@PathVariable final String code) {  
 *log*.info("REST request to DELETE entity with code : {}", code);  
 final T deleted;  
 try {  
 deleted = service.removeByCode(code);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong deleting the entity with the code: {}", code);  
 return null;  
 }  
 return ResponseEntity.*ok*(deleted);  
 }  
}

* + 1. Controller specific

RestController  
@RequestMapping(value = "/v1/reservations")  
@CrossOrigin(maxAge = 3600)  
@Slf4j  
public class ReservationController extends BaseController<Reservation> {  
  
 private static final String *USER\_USERNAME\_PATH* = "/user/{username}";  
 private static final String *CREATE\_PATH* = "/create";  
 private static final String *RESERVATION\_NEXT\_USERNAME\_PATH* = "/reservation-next/{username}";  
 private static final String *RESERVATION\_CLAIM\_PATH* = "/claim/{reservationId}";  
 private static final String *RESERVATION\_START\_PATH* = "/start/{reservationId}";  
 private static final String *RESERVATION\_COMPLETE\_PATH* = "/complete/{reservationId}";  
  
  
 @Inject  
 private ReservationService service;  
  
  
 @Override  
 @Inject  
 public void setService(BaseService<Reservation> service) {  
 super.setService(this.service);  
 }  
  
 */\*\*  
 \** ***@param*** *username of the user  
 \** ***@return*** *the list with all the reservations of a user  
 \*/* @GetMapping(*USER\_USERNAME\_PATH*)  
 public ResponseEntity<List<Reservation>> getReservationsForUser(@PathVariable final String username) {  
 *log*.info("REST request to GET reservations by username : {}", username);  
 final List<Reservation> reservationList = service.findAllForUserByUsername(username);  
 if (reservationList == null) {  
 *log*.info("No reservations found for the user with username: {}", username);  
 return ResponseEntity.*notFound*().build();  
 } else {  
 *log*.info("Reservations found for the user: {}", reservationList);  
 return ResponseEntity.*ok*(reservationList);  
 }  
 }  
  
  
 */\*\*  
 \** ***@param*** *reservationCreate the entity to be added  
 \** ***@return*** *the created entity  
 \*/* @PostMapping(*CREATE\_PATH*)  
 public ResponseEntity<Reservation> createReservation(@RequestBody final ReservationCreate reservationCreate) {  
 *log*.info("REST request to CREATE RESERVATION : {}", reservationCreate);  
 final Reservation created;  
 try {  
 created = service.createReservation(reservationCreate);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong creating the entity : {}", reservationCreate, e);  
 return null;  
 }  
 return ResponseEntity.*ok*(created);  
 }  
  
  
 */\*\*  
 \** ***@param*** *username of the user  
 \** ***@return*** *the list with all the reservations of a user  
 \*/* @GetMapping(*RESERVATION\_NEXT\_USERNAME\_PATH*)  
 public ResponseEntity<ReservationNext> getReservationNextForUser(@PathVariable final String username) {  
 *log*.info("REST request to GET reservationsNEXT by username : {}", username);  
 final ReservationNext reservationNext = service.getReservationNext(username);  
 if (reservationNext == null) {  
 *log*.info("No reservations found for the user with username: {}", username);  
 return ResponseEntity.*notFound*().build();  
 } else {  
 *log*.info("Reservations found for the user: {}", reservationNext);  
 return ResponseEntity.*ok*(reservationNext);  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *reservationId id of the resevation to be claimed  
 \** ***@return*** *the created entity  
 \*/* @PutMapping(*RESERVATION\_CLAIM\_PATH*)  
 @Transactional  
 public ResponseEntity<Reservation> claim(@PathVariable final Long reservationId) {  
 *log*.info("REST request to CLAIM RESERVATION : {}", reservationId);  
 final Reservation updated;  
 try {  
 updated = service.claim(reservationId);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong UPDATING the entity : {}", reservationId);  
 return null;  
 }  
 return ResponseEntity.*ok*(updated);  
 }  
  
 */\*\*  
 \** ***@param*** *reservationId the id of the reservation to be started  
 \** ***@return*** *the created entity  
 \*/* @PutMapping(*RESERVATION\_START\_PATH*)  
 @Transactional  
 public ResponseEntity<Reservation> start(@PathVariable final Long reservationId) {  
 *log*.info("REST request to CLAIM RESERVATION : {}", reservationId);  
 final Reservation updated;  
 try {  
 updated = service.start(reservationId);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong UPDATING the entity : {}", reservationId);  
 return null;  
 }  
 return ResponseEntity.*ok*(updated);  
 }  
  
  
 */\*\*  
 \** ***@param*** *reservationId identifier of the completed reservation  
 \** ***@return*** *the created entity  
 \*/* @PutMapping(*RESERVATION\_COMPLETE\_PATH*)  
 @Transactional  
 public ResponseEntity<Reservation> complete(@PathVariable final Long reservationId) {  
 *log*.info("REST request to CLAIM RESERVATION : {}", reservationId);  
 final Reservation updated;  
 try {  
 updated = service.complete(reservationId);  
 } catch (final Exception e) {  
 *log*.info("Something went wrong UPDATING the entity : {}", reservationId);  
 return null;  
 }  
 return ResponseEntity.*ok*(updated);  
 }  
  
}

* + 1. Fisierul de configurări

server.port: 8000  
reservation.claim.allocated.time: 900  
  
spring:  
 tomcat:  
 max-wait: 10000  
 max-active: 5  
 test-on-borrow: true  
 main:  
 banner-mode: "off"  
 jpa:  
 generate-ddl: true  
 hibernate:  
 ddl-auto: update  
 datasource:  
 initialization-mode: always  
 platform: postgres  
 url: jdbc:postgresql://localhost:5432/parking-management-system  
 username: postgres  
 password: warlord  
  
  
stripe:  
 secret:  
 key: sk\_test\_51HskSzKjGxGlZ0lyFGqgqRioYYAOyxfjkJf1LPNvhQ1WPgbyRJ8P3rbpGpHJG0cJproPUd3jgpuT4JBlBb8nowwU00i38BUUCn  
 publishable:  
 key: pk\_test\_51HskSzKjGxGlZ0ly9rb5ZXdHsozaY8DPqliYzVgOXhNvN6rtKUrRPM9lSH5WhzZUO79TNNIll80m7FjsYw0ZDY8D00lhUZjT52

* + 1. Serviciul Stripe

@Slf4j  
@AllArgsConstructor  
@Service  
public class UserCreateStripeService {  
  
 private final StripeConfig config;  
  
 private final UserService userService;  
  
 public Customer createCustomer(final String email, final String name) {  
 Stripe.*apiKey* = config.getSecret().getKey();  
 Map<String, Object> params =Maps.*newHashMap*();  
 params.put("description", "My First Test Customer (created for API docs)");  
 params.put("name", name);  
 params.put("email", email);  
  
 try {  
 Customer customer = Customer.*create*(params);  
 userService.updateStripe(email, customer.getId());  
 *log*.info("Successfully created customer: {}", customer);  
 return customer;  
 } catch (StripeException e) {  
 *log*.info("Couldn't create Stripe customer for stripe email {}", email, e);  
 }  
 return null;  
 }  
  
 public Card createCard(final String customerId, final UserCreate userCreate) {  
 Map<String, Object> retrieveParams = Maps.*newHashMap*();  
 List<String> expandList = Lists.*newArrayList*();  
 expandList.add("sources");  
 retrieveParams.put("expand", expandList);  
 Customer customer;  
 try {  
 customer = Customer.*retrieve*(  
 customerId,  
 retrieveParams,  
 null  
 );  
 Map<String, Object> sourceParams = Maps.*newHashMap*();  
 sourceParams.put("object", "card");  
 sourceParams.put("number", userCreate.getCreditCardNumber());  
 sourceParams.put("exp\_month", userCreate.getCreditCardExpMonth());  
 sourceParams.put("exp\_year", userCreate.getCreditCardExpYear());  
 sourceParams.put("cvc", userCreate.getCreditCardCCV());  
 sourceParams.put("name", customer.getName());  
 Map<String, Object> params = new HashMap<>();  
 // params.put("source", sourceParams);  
 params.put("source", "tok\_amex");  
 Card card =  
 (Card) customer.getSources().create(params);  
 *log*.info("Successfully created Stripe card: {}", card);  
 return card;  
 } catch (StripeException e) {  
 *log*.info("Couldn't find customer for stripe id {}", customerId, e);  
 }  
  
 return null;  
 }  
}

1. **CLIENTUL WEB**
   1. **Fișier TYPSECRIPT**
2. import { Component, OnInit } from '@angular/core';
3. import { ParkingCreate } from 'src/app/core/models';
4. import { ActivatedRoute, Router } from '@angular/router';
5. import { RestService } from 'src/app/core/RestService.service';
6. import { faSave, faArrowRight } from '@fortawesome/free-solid-svg-icons';
7. import { FaIconLibrary } from '@fortawesome/angular-fontawesome';
8. @Component({
9. selector: 'app-parking-create',
10. templateUrl: './parking-create.component.html',
11. providers: [RestService]
12. })
13. export class ParkingCreateComponent implements OnInit {
14. toAdd: ParkingCreate = {
15. parking: {closesAt: '', location: '', name: '', opensAt: ''},
16. numberOfLevels: '',
17. parkingZoneEndingLetter: '',
18. parkingZoneStartingLetter: '',
19. parkingZoneSpotNumber: 1,
20. };
21. constructor(
22. private route: ActivatedRoute,
23. private parkingService: RestService<ParkingCreate>,
24. private router: Router,
25. private library: FaIconLibrary
26. ) {
27. library.addIcons(faSave, faArrowRight);
28. }
29. ngOnInit(): void {}
30. create() {
31. const added = this.parkingService
32. .create(this.toAdd, '/v1/parkings/configure')
33. .subscribe((added) => {
34. this.router.navigate(['parkings']);
35. });
36. }
37. back() {
38. this.router.navigate(['parkings']);
39. }
40. }
    1. **Fișier HTML**
41. <div class="col-md-12 col-sm-12 col-lg-12">
42. <div class="text-center row">
43. <a class="btn btn-link btn-lg" (click)="back()">Parkings</a>
44. <a class="btn btn-link btn-lg" disabled > <fa-icon [icon]="['fas', 'arrow-right']"></fa-icon> </a>
45. <a class="btn btn-link btn-lg" disabled> create parking </a>
46. </div>
47. <hr/>
48. <div class="form-group float-label-control">
49. <label><span>Name\*</span></label>
50. <input
51. type="text"
52. class="form-control"
53. placeholder="Iulius Mall Parking"
54. title="Name"
55. name="name"
56. [(ngModel)]="toAdd.parking.name"
57. />
58. </div>
59. <div class="form-group float-label-control">
60. <label><span>Price per hour\*</span></label>
61. <input
62. type="number"
63. class="form-control"
64. placeholder="1"
65. title="Price"
66. name="cost"
67. [(ngModel)]="toAdd.parking.pricePerHour"
68. min="1"
69. />
70. </div>
71. <div class="form-group float-label-control">
72. <label><span>Location</span></label>
73. <input
74. type="text"
75. class="form-control"
76. placeholder="Cluj-Napoca, Cluj"
77. title="Location"
78. [(ngModel)]="toAdd.parking.location"
79. name="location"
80. />
81. </div>
82. <div class="form-group float-label-control">
83. <label><span>Opening Hour</span></label>
84. <input
85. type="text"
86. class="form-control"
87. placeholder="hh:mm"
88. title="opening hour"
89. [(ngModel)]="toAdd.parking.opensAt"
90. name="openingHour"
91. />
92. </div>
93. <div class="form-group float-label-control">
94. <label><span>Closing hour</span></label>
95. <input
96. type="text"
97. class="form-control"
98. placeholder="hh:mm"
99. title="closing hour"
100. [(ngModel)]="toAdd.parking.closesAt"
101. name="closingHour"
102. />
103. <div class="form-group float-label-control">
104. <label><span>Number of levels</span></label>
105. <input
106. type="number"
107. class="form-control"
108. placeholder="3"
109. title="number of levels"
110. [(ngModel)]="toAdd.numberOfLevels"
111. name="numberOfLevels"
112. min="1"
113. />
114. </div>
115. <div class="form-group float-label-control">
116. <label><span>Parking zone starting letter</span></label>
117. <input
118. type="text"
119. class="form-control"
120. placeholder="A"
121. title="zone staring letter"
122. [(ngModel)]="toAdd.parkingZoneStartingLetter"
123. maxlength="1"
124. name="parkingZoneStartingLetter"
125. />
126. </div>
127. <div class="form-group float-label-control">
128. <label><span>Parking zone ending letter</span></label>
129. <input
130. type="text"
131. class="form-control"
132. placeholder="H"
133. title="zone ending letter"
134. [(ngModel)]="toAdd.parkingZoneEndingLetter"
135. maxlength="1"
136. name="parkingZoneEndingLetter"
137. />
138. </div>
139. <div class="form-group float-label-control">
140. <label><span>Number of spots per zone</span></label>
141. <input
142. type="number"
143. class="form-control"
144. placeholder="15"
145. title="number of spots per zone"
146. [(ngModel)]="toAdd.parkingZoneSpotNumber"
147. name="parkingZoneSpotNumber"
148. min="1"
149. />
150. </div>
151. </div>
152. <button id="save-btn" class="btn btn-success btn-md" (click)="create()">
153. <fa-icon [icon]="['fas', 'save']"></fa-icon> <span>Save</span>
154. </button>
155. </div>
156. 1. **Serviciul Generic HTTP**
157. import { HttpClient } from '@angular/common/http';
158. import { catchError, tap } from 'rxjs/operators';
159. import { Injectable } from '@angular/core';
160. import { Subject, Observable, of } from 'rxjs';
161. import { environment } from '../../environments/environment';
162. @Injectable()
163. export class RestService<T> {
164. baseUrl = environment.serverPort;
165. constructor(private http: HttpClient) {}
166. getList(url: String): Observable<T[]> {
167. return this.http.get<T[]>(this.baseUrl + url).pipe(
168. catchError(this.handleError<T[]>('getList'))
169. );
170. }
171. getById(id: number, url: String): Observable<T> {
172. const urlTemp = this.baseUrl + url + '/id/' + id;
173. return this.http.get<T>(urlTemp).pipe(
174. catchError(this.handleError<T>('getById'))
175. );
176. }
177. update(updateDetails: T, url: String): Observable<T> {
178. const urlTemp = this.baseUrl + url;
179. return this.http.put<T>(urlTemp, updateDetails).pipe(
180. catchError(this.handleError<T>('update'))
181. );
182. }
183. create(toAdd: T, url: String): Observable<T> {
184. const urlTemp = this.baseUrl + url;
185. return this.http.post<T>(urlTemp, toAdd).pipe(
186. catchError(this.handleError<T>('create'))
187. );
188. }
189. delete(deleteId: number, url: String): Observable<T> {
190. const urlTemp = this.baseUrl + url + '/' + deleteId;
191. return this.http.delete<T>(urlTemp).pipe(
192. catchError(this.handleError<T>('create'))
193. );
194. }
195. handleError<T>(operation = 'operation', result?: T) {
196. return (error: any): Observable<T> => {
197. console.error(error); // log to console instead
198. return of(result as T);
199. };
200. }
201. }

**CLIENTUL ANDROID**

* **Activitate**
* public class ReservationActivity extends MenuHelper implements  
   AdapterView.OnItemSelectedListener {  
    
    
   private ParkingService parkingService;  
    
   private ReservationService reservationService;  
    
   final ReservationCreate reservationCreate = ReservationCreate.*builder*().build();  
   List<String> parkingList = Lists.*newArrayList*();  
    
   static TextView *selectedDateTV*;  
   static String *date*;  
    
    
   @Override  
   protected void onCreate(Bundle savedInstanceState) {  
   super.onCreate(savedInstanceState);  
   setContentView(R.layout.*activity\_reservation*);  
   Toolbar toolbar = findViewById(R.id.*toolbar*);  
   setSupportActionBar(toolbar);  
   final SharedPreferences sharedPref = this.getSharedPreferences(*SHARED\_PREFERENCES*, Context.*MODE\_PRIVATE*);  
   final String username= sharedPref.getString(*USERNAME*, "none");  
   RetrofitServicesInitializer retrofitServicesInitializer = new RetrofitServicesInitializer(sharedPref.getString(*BASE\_URL*, "none"));  
   reservationService = retrofitServicesInitializer.getReservationService();  
   parkingService = retrofitServicesInitializer.getParkingService();  
    
   final EditText startTimeET = (EditText) findViewById(R.id.*reservation\_start\_time\_id*);  
   final EditText endTimeET = (EditText) findViewById(R.id.*reservation\_end\_time\_id*);  
   final TextView invalidTimeTV = (TextView) findViewById(R.id.*reservation\_invalid\_time\_id*);  
   invalidTimeTV.setVisibility(View.*INVISIBLE*);  
    
   *selectedDateTV* = (TextView) findViewById(R.id.*reservation\_selected\_date\_id*);  
    
   final BootstrapButton dateBtn = (BootstrapButton) findViewById(R.id.*reservation\_date\_btn\_id*);  
   final BootstrapButton reserveBtn = (BootstrapButton) findViewById(R.id.*reservation\_reserve\_btn\_id*);  
    
   final Spinner parkingNamesSpinner = (Spinner) findViewById(R.id.*reservation\_parking\_names\_id*);  
    
    
   parkingService.getAll().enqueue(new Callback<List<Parking>>() {  
    
   @Override  
   public void onResponse(Call<List<Parking>> call, Response<List<Parking>> response) {  
   parkingList = response.body().stream().map(Parking::getName).collect(Collectors.*toList*());  
   ArrayAdapter parkingNamesAdaptor = new ArrayAdapter(ReservationActivity.this,  
   android.R.layout.*simple\_spinner\_item*,  
   parkingList  
   );  
   parkingNamesAdaptor.setDropDownViewResource(android.R.layout.*simple\_spinner\_dropdown\_item*);  
   parkingNamesSpinner.setAdapter(parkingNamesAdaptor);  
   parkingNamesSpinner.setOnItemSelectedListener(ReservationActivity.this);  
   }  
    
   @Override  
   public void onFailure(Call<List<Parking>> call, Throwable t) {  
   *//Nothing happens on failure* Toast.*makeText*(ReservationActivity.this, "failed to load parking names", Toast.*LENGTH\_LONG*);  
   }  
   });  
    
   reserveBtn.setOnClickListener(new View.OnClickListener() {  
   @Override  
   public void onClick(View view) {  
   String startTime = startTimeET.getText().toString();  
   String endTime = endTimeET.getText().toString();  
   if (*isValidHourAndMinute*(endTime) && *isValidHourAndMinute*(startTime)) {  
   startTime = *convertTimeToParsingInstantFormat*(*date*, startTime);  
   endTime = *convertTimeToParsingInstantFormat*(*date*, endTime);  
   reservationCreate.setStartTime(startTime);  
   reservationCreate.setEndTime(endTime);  
   reservationCreate.setUsername(username);  
   reservationService.createReservation(reservationCreate).enqueue(new Callback<Reservation>() {  
   @Override  
   public void onResponse(Call<Reservation> call, Response<Reservation> response) {  
   invalidTimeTV.setVisibility(View.*INVISIBLE*);  
   Toast.*makeText*(getApplicationContext(), "Created reservation: " + response.body().toString(), Toast.*LENGTH\_LONG*).show();  
   Intent intent = new Intent(ReservationActivity.this, ReservationHistoryActivity.class);  
   startActivity(intent);  
   }  
    
   @Override  
   public void onFailure(Call<Reservation> call, Throwable t) {  
   *//TODO* Toast.*makeText*(getApplicationContext(), "Failed to create reservation", Toast.*LENGTH\_LONG*).show();  
   }  
   });  
   } else {  
   invalidTimeTV.setVisibility(View.*VISIBLE*);  
   }  
   }  
   });  
    
   dateBtn.setOnClickListener(new View.OnClickListener() {  
   @Override  
   public void onClick(View view) {  
   showDatePickerDialog(view);  
   }  
   });  
   }  
    
   @Override  
   public void onItemSelected(AdapterView<?> adapterView, View view, int i, long l) {  
   reservationCreate.setParkingName((String) adapterView.getItemAtPosition(i));  
   }  
    
   @Override  
   public void onNothingSelected(AdapterView<?> adapterView) {  
   if (adapterView.getChildCount() > 0) {  
   reservationCreate.setParkingName((String) adapterView.getItemAtPosition(0));  
   }  
   }  
    
   public void showDatePickerDialog(View v) {  
   DialogFragment newFragment = new DatePickerFragment();  
   newFragment.show(getSupportFragmentManager(), "datePicker");  
   }  
    
    
    
   public static class DatePickerFragment extends DialogFragment  
   implements DatePickerDialog.OnDateSetListener {  
    
   @Override  
   public Dialog onCreateDialog(Bundle savedInstanceState) {  
   final Calendar c = Calendar.*getInstance*();  
   int year = c.get(Calendar.*YEAR*);  
   int month = c.get(Calendar.*MONTH*);  
   int day = c.get(Calendar.*DAY\_OF\_MONTH*);  
    
   *date* = BaseUtils.*convertDateToParsingInstantFormat*(year, month, day);  
    
   return new DatePickerDialog(getActivity(), this, year, month, day);  
   }  
    
   public void onDateSet(DatePicker view, int year, int month, int day) {  
    
   *selectedDateTV*.setText("Date: " + day + "-" + (++month));  
   *date* = BaseUtils.*convertDateToParsingInstantFormat*(year, month, day);  
   }  
   }  
    
  }
* **Serviciul Generic Retrofit**

public interface BaseService {  
  
 String *CONTENT\_TYPE* = "Content-Type: application/json";  
 String *AUTHORIZATION* = "X-Authorization: user";  
  
 String *ID\_PATH* = "id/{id}";  
 String *CODE\_PATH* = "code/{code}";  
 String *LIST\_PATH* = "list";  
 String *USER\_USERNAME\_PATH* = "user/{username}";  
  
 @GET(*ID\_PATH*)  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> getById(@Path("id") final Long id);  
  
 @GET(*CODE\_PATH*)  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> getByCode(@Path("code") final String code);  
  
  
 @POST  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> create(@Body final Object body);  
  
 @PUT  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> update(@Body final Object body);  
  
 @DELETE(*ID\_PATH*)  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> deleteById(@Path ("id") final Long id);  
  
 @DELETE(*CODE\_PATH*)  
 @Headers({*CONTENT\_TYPE*, *AUTHORIZATION*})  
 Call<Object> deleteByCode(@Path ("code") final String code);  
}