Universitatea „Politehnica“ Timișoara

Facultatea de Automatică și Calculatoare

**Calculatoare și Tehnologia Informației**

**APLICAȚIE MOBILĂ DE RIDESHARING EFICIENT**

Proiect de Diplomă

*Absolvent:*

**Bianca-Elena STOIAN**

*Coordonator științific:*

Ș.l. dr. ing. **Casandra HOLOTESCU**

Timișoara

2019

**CUPRINS**

[1. INTRODUCERE 5](#_Toc11449730)

[1.1. Contextul problemei 5](#_Toc11449731)

[1.2. Descrierea proiectului 6](#_Toc11449732)

[2. SPECIFICAȚII 7](#_Toc11449733)

[2.1. Cerințe 7](#_Toc11449734)

[2.2. Utilizatori 7](#_Toc11449735)

[2.3. Use case-uri 7](#_Toc11449736)

[3. TEHNOLOGII 9](#_Toc11449737)

[3.1. Tehnologii folosite pe partea de Client (client-side) 9](#_Toc11449738)

[**3.1.1.** **Sistemul de operare Android** 9](#_Toc11449739)

[3.2. Tehnologii folosite pe partea de server (server-side) 14](#_Toc11449740)

[**3.2.1.** **JPA** 14](#_Toc11449741)

[**3.2.2.** **Spring Framework** 17](#_Toc11449742)

[**3.2.3.** **Spring Boot** 18](#_Toc11449743)

[4. IMPLEMENTARE 19](#_Toc11449744)

[4.1. Arhitectura 19](#_Toc11449745)

[4.2. Baza de date 20](#_Toc11449746)

[4.3. Clientul Android 22](#_Toc11449747)

[**4.3.1.** **Pornirea aplicației – MainActivity** 22](#_Toc11449748)

[**4.3.2.** **Autentificarea – LogInActivity** 23](#_Toc11449749)

[**4.3.3.** **Înregistrarea unui nou utilizator – SignUpActivity** 25](#_Toc11449750)

[**4.3.4.** **Interfața după momentul autentificării - TabsActivity** 26](#_Toc11449751)

[**4.3.5.** **Pagina de profil a utilizatorului – ProfileFragment** 27](#_Toc11449752)

[**4.3.6.** **Pagina de cereri de carpool - RequestFragment** 28](#_Toc11449753)

[**4.3.7.** **Pagina curselor din ziua respectivă – UpcomingRideFrament** 31](#_Toc11449754)

[**4.3.8.** **Vizualizarea traseului și efectuarea unei curse – MapNavigationActivity** 31](#_Toc11449755)

[**4.3.9.** **Managerul de sesiune SessionManager** 32](#_Toc11449756)

[**4.3.10.** **Efectuarea apelurilor HTTP cu ajutorul Volley** 33](#_Toc11449757)

[**4.3.11.** **Serviciul de interceptare a mesajelor venite prin Firebase Cloud Messaging (FCM)** 34](#_Toc11449758)

[4.4. Serverul aplicației 37](#_Toc11449759)

[**4.4.1.** **Definirea modelelor** 37](#_Toc11449760)

[**4.4.2.** **Definirea claselor pentru transmiterea datelor în rețea (DTO/data transfer object)** 38](#_Toc11449761)

[**4.4.3.** **Definirea repository-urilor** 39](#_Toc11449762)

[**4.4.4.** **Definirea serviciilor** 40](#_Toc11449763)

[**4.4.5.** **Definirea controller-elor** 41](#_Toc11449764)

[**4.4.6.** **Stabilirea distanțelor între puncte diferite prin Distance Matrix API** 42](#_Toc11449765)

[**4.4.7.** **Transformarea adreselor în coordonate prin Geocoding API** 43](#_Toc11449766)

[**4.4.8.** **Crearea carpool-urilor** 44](#_Toc11449767)

[**4.4.9.** **Trimiterea notificărilor prin Firebase Cloud Messaging** 47](#_Toc11449768)

[5. UTILIZAREA 48](#_Toc11449769)

[6. CONCLUZII 49](#_Toc11449770)

[6.1. Concluzii 49](#_Toc11449771)

[6.2. Viitoare îmbunătățiri 49](#_Toc11449772)

[7. BIBLIOGRAFIE 50](#_Toc11449773)

# **INTRODUCERE**

## **Contextul problemei**

Creșterea populației orașelor duce, inevitabil, la creșterea numărului de mașini înmatriculate în respectivele localități. Cu toate acestea, infrastructura nu este și nu poate fi îmbunătățită în același ritm. În acest context, apar două probleme pentru șoferi:

1. Se creează ambuteiaje
2. Găsirea unui loc de parcare devine tot mai dificilă

Aceste lucruri duc la creșterea considerabilă a timpului petrecut în trafic, timp pe care șoferii sau pasagerii l-ar putea petrece făcând lucruri productive. Sporirea agresivității și nervozității participanților la trafic nu este de neglijat, reprezentând un factor semnificativ pentru producerea accidentelor. De asemenea, costurile plătite cu sănătatea, în absența unor accidente, sunt considerabile, ca urmare a poluării accentuate, mai ales în intersecții.

Un studiu realizat în Statele Unite ale Americii arată că 85% din mașinile aflate în trafic la un moment dat transportă un singur pasager [1], în timp ce un alt studiu, realizat în Europa, arată că media de ocupare a locurilor dintr-un vehicul este în continuă scădere, în prezent la aproximativ 1.45 (incluzând șoferul) [2].

Numărul de mașini poate fi considerabil redus dacă se ia în considerare împărțirea unei mașini de către mai multe persoane care pornesc din locuri apropiate și merg înspre locuri apropiate. Ca urmare, apar numeroase avantaje:

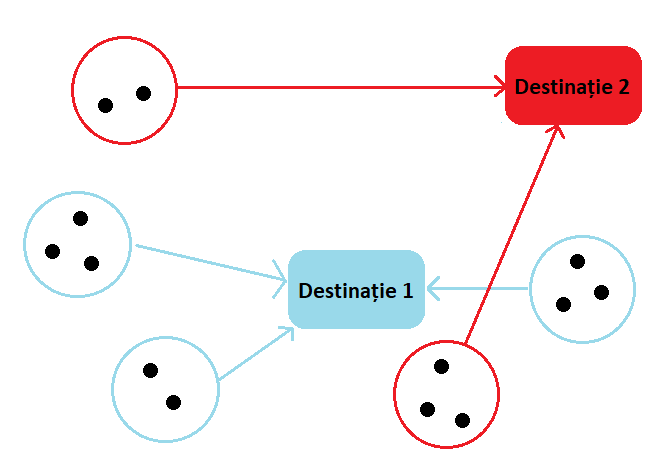
1. Timpul petrecut în trafic scade, în ciuda faptului că apar devieri de la traseu pentru a lua toți pasagerii sau pentru a-i lăsa într-un loc cât mai convenabil
2. Timpul petrecut în căutarea unui loc de parcare scade
3. Scad emisiile de dioxid de carbon, benefică sănătății oamenilor și mediului înconjurător
4. Costurile benzinei se împart la mai multe persoane
5. Șoferii se pot schimba prin rotație
6. Există posibilitatea de a socializa și de a ne destinde în drumul spre destinație

## **Descrierea proiectului**

Deși situațiile descrise mai sus sunt general valabile pentru orice zonă dintr-o localitate și pentru orice tip de activități la care șoferii sau pasagerii urmează să ia parte, proiectul de față își propune să facă un prim pas în direcția eficientizării transportului către și dinspre lucru, respectiv școală.

În acest scop, s-au ales o serie de puncte finale către care se vor realiza carpool-uri. Utilizatorul are posibilitatea de a introduce date privind orele de sosire/plecare către punctul său de interes din acea zi, urmând ca aplicația să realizeze grupuri de persoane care vor călători împreună spre acel loc, pentru același interval orar.

Aplicația creează carpool-uri atât pe baza orelor, cât și pe baza distanțelor dintre șoferi. Astfel, figura 1 arată felul în care 13 persoane pot fi grupate în 5 carpool-uri ce pleacă din locuri apropiate și merg către 2 destinații diferite. Presupunând că aceste persoane ar fi călătorit individual, se observă o posibilă reducere a numărului de mașini în trafic cu 60% (8 mașini din 13), precum și a numărului de mașini ce trebuie parcate în zonele respective aproximativ 40% în cazul ambelor destinații (3 mașini din 5 în cazul primeia, 5 mașini din 8 în cazul celei de-a doua).



*Fig. 1.2.1. – modul în care 13 persoane ar putea fi grupate în 5 carpool-uri*

*mergând către 2 destinații*

# **SPECIFICAȚII**

## **Cerințe**

Aplicația de ridesharing eficient este o aplicație mobilă, în cadrul căreia utilizatorul poate efectua o serie de acțiuni:

* Se poate înregistra, în cazul în care nu are deja un cont; validarea se face pe baza numărului de telefon, care trebuie să fie unic
* Odată autentificat, utilizatorul poate face următoarele acțiuni:
* își poate modifica profilul, inclusiv numărul de telefon, dar se va verifica existența acestuia în baza de date, caz în care modificarea nu poate avea loc
* poate adăuga cereri de carpool, selectând ziua, destinația, rolul pe care îl poate îndeplini în acea zi (doar șofer, doar pasager sau oricare din cele două) și orele de plecare, respectiv întoarcere
* poate modifica/șterge cererile existente
* poate vedea repartizarea din ziua respectivă, inclusiv traseul generat
* poate efectua o cursă la ora stabilită

Destinațiile pe care le poate alege utilizatorul sunt următoarele: Universitatea Politehnica Timișoara/Universitatea de Vest, Continental Automotive, Vox Techonology Park, City Business Center, United Business Center.

## **Utilizatori**

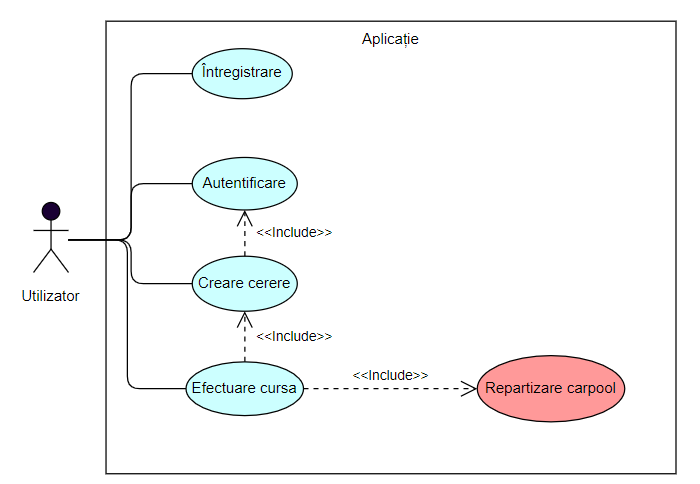
Destinațiile pe care un utilizator le poate alege sunt limitate în acest moment la cele cinci de mai sus. Deși aplicația nu limitează utilizarea sa doar de către persoanele care se deplasează pentru lucru sau școală către una dintre acestea, este puțin probabil ca ea să răspundă altor nevoi de transport. Ca urmare, utilizatorii principali sunt persoanele care se deplasează către cele cinci puncte.

## **Use case-uri**

Use case-ul principal al aplicației îl reprezintă efectuarea curselor de către utilizatori. Pentru a se ajunge la acest lucru, utilizatorul trebuie să efectueze următorii pași:

* Să își creeze un cont
* Să se autentifice
* Să facă o cerere de carpool

După realizarea acestor lucruri, utilizatorul trebuie să aștepte să fie repartizat într-un anumit carpool. Abia după ce se confirmă acest lucru, la ora stabilită în cazul șoferului, respectiv la ora aproximată de aplicație în cazul pasagerilor, cursa poate începe.



*Fig. 2.1. – Use case-ul principal al aplicației*

# **TEHNOLOGII**

## **Tehnologii folosite pe partea de Client (client-side)**

### **Sistemul de operare Android**

Android este un sistem de operare dezvoltat de către Google, destinat în principal dispozitivelor mobile cu ecran tactil, de tipul smartphone sau tabletă. Pe lângă acestea, compania a dezvoltat variațiuni ale Android, ce pot fi folosite la televizoare, automobile, ceasuri, console de jocuri, camere foto și alte dispozitive electronice.

Sistemul de operare a fost dezvăluit în anul 2007, iar primul dispozitiv comercial care îl rula a apărut pe piață în septembrie 2008. De atunci, Android a trecut prin multiple lansări majore, cea mai recentă datând din August 2018 - Android 9.0 („Pie”).

Începând cu 2011, Android este cel mai comun sistem de operare pentru telefoanele mobile la nivel mondial, iar, începând cu anul 2013, și pentru tablete. În mai 2017, Android avea 2 miliarde de utilizatori activi lunar.

Codul sursă Android este disponibil sub o licență open-source, dezvoltat în privat de Google și publicat în momentul apariției unei noi versiuni. Totuși, doar baza sistemului de operare (incluzând anumite aplicații) este open-source, majoritatea dispozitivelor Android fiind livrate cu o cantitate mare de software prorietar, precum Google Search, Google Chrome, Google Play Store, Youtube.

Aplicațiile ce extind funcționalitatea dispozitivelor sunt scrise folosind kitul de dezvoltare software (SDK) Android și, adesea, limbajul de programare Java. În mai 2017, Google a anunțat suportul pentru dezvoltarea aplicațiilor în limbajul de programare Kotlin, urmând ca la începutul lunii mai 2019 să devină limbajul de dezvoltare preferat de companie.

SDK-ul Android include un set cuprinzător de instrumente de dezvoltare, inclusiv un debugger, biblioteci software, un emulator de dispozitiv bazat pe QEMU, documentații, exemple de cod și programe de învățare. Inițial, mediul de dezvoltare integrată (IDE) recomandat de Google a fost Eclipse, folosind plugin-ul Android Development Tools (ADT). În decembrie 2014, compania a lansat Android Studio, având la bază IntelliJ IDEA, ca principalul IDE pentru dezvoltarea aplicațiilor Android.

Pe sistemul de operare Android pot rula o varietate în continuă creștere de aplicații third-party. Acestea pot rula fie prin descărcarea și instalarea fișierului APK (Android Application Package) al aplicației, fie prin descărcarea aplicației de pe un magazin de aplicații. Magazinul Google Play este principalul magazin pentru Android ce respectă cerințele de compatibilitate Google și are licența de software Google Mobile Services. În iulie 2013 existau mai mult de un milion de aplicații Android disponibile pe Magazinul Play. De asemenea, tot în iulie 2013, numărul de instalări ale aplicațiilor de pe Magazinul Play a depășit 50 de miliarde [3].

**Arhitectura Platformei Android**

1. *Nucleul linux*

Nucleul Android este bazat pe nucleul Linux. Totuși, pentru a crea un sistem de operare care să acopere nevoile unui dispozitiv mobil, nucleul Linux este modificat prin îmbunătățirea consumului de putere, a funcționalităților radio și a lucrului în rețea.

Spre deosebire de nucleul Linux, nucleul Android include puțin software GNU, excluzând chiar și Biblioteca C GNU, înlocuită de Bionic. Principalele avantaje ale folosirii Bionic sunt timpii reduși de rulare și optimizarea pentru CPU-uri cu frecvențe joase.

1. *Nivelul de abstractizare a resurselor hardware (Hardware Abstraction Layer/HAL)*

Nivelul de abstractizare a resurselor hardware pune la dispoziție interfețe standard ce expun caracteristicile hardware framework-ului de nivel mai înalt Java API. Acest nivel constă din mai multe module de biblioteci, implementând fiecare o interfață pentru o anumită componentă hardware. Atunci când un API al framework-ului cere accesul la o componentă hardware (de exemplu, camera), sistemul Android încarcă modulul corespunzător acesteia.

1. *Runtime-ul Android*

Până la versiunea 5.0, Android folosea Dalvik ca și mașină virtuală de proces, având compilare de tipul just-in-time (JIT) pentru a rula "dex-code" (Dalvik Executable) Dalvik, ce era de obicei tradus din bytecode Java. Urmând principiile JIT, pe lângă interpretarea majorității codului aplicației, Dalvik face compilarea și execuția nativă a anumitor secvențe de cod frecvent executate de fiecare dată când o aplicație este lansată.

Android 4.4 a introdus Android Runtime (ART) ca un nou mediu de runtime, ce folosește compilare de tipul ahead-of-time (AOT) pentru a compila în întregime bytecode-ul aplicației în cod mașină la instalarea aplicației. Experimentală în Android 4.4, a devenit aceasta a devenit singura opțiune de runtime odată cu lansarea următoarei versiuni majore de Android, 5.0.

Android include, de asemenea, o serie de biblioteci de runtime ce pun la dispoziție majoritatea funcționalităților oferite de limbajul de programare Java, inclusiv anumite funcționalități introduse de Java 8, folosite de framework-ul Java API.

1. *Bibliotecile native C/C++*

Multe dintre componentele și serviciile de bază ale sistemului Android , precum ART sau HAL, sunt construite din cod nativ ce necesită biblioteci native scrise în C și C++. Platforma Android pune la dispoziția framework-ului Java API-uri ce expun funcționalități ale unor bilbioteci native către aplicații. De exemplu, se poate accesa OpenGL ES prin API-ul Java OpenGL pentru a oferi suport în desenarea și manipularea graficii 2D și 3D în aplicații.

În cazul în care se dezvoltă o aplicație ce necesită cod C sau C++, se poate folosi Android NDK (Native Development Kit) pentru a accesa librăriile native ale platformei direct din codul nativ.

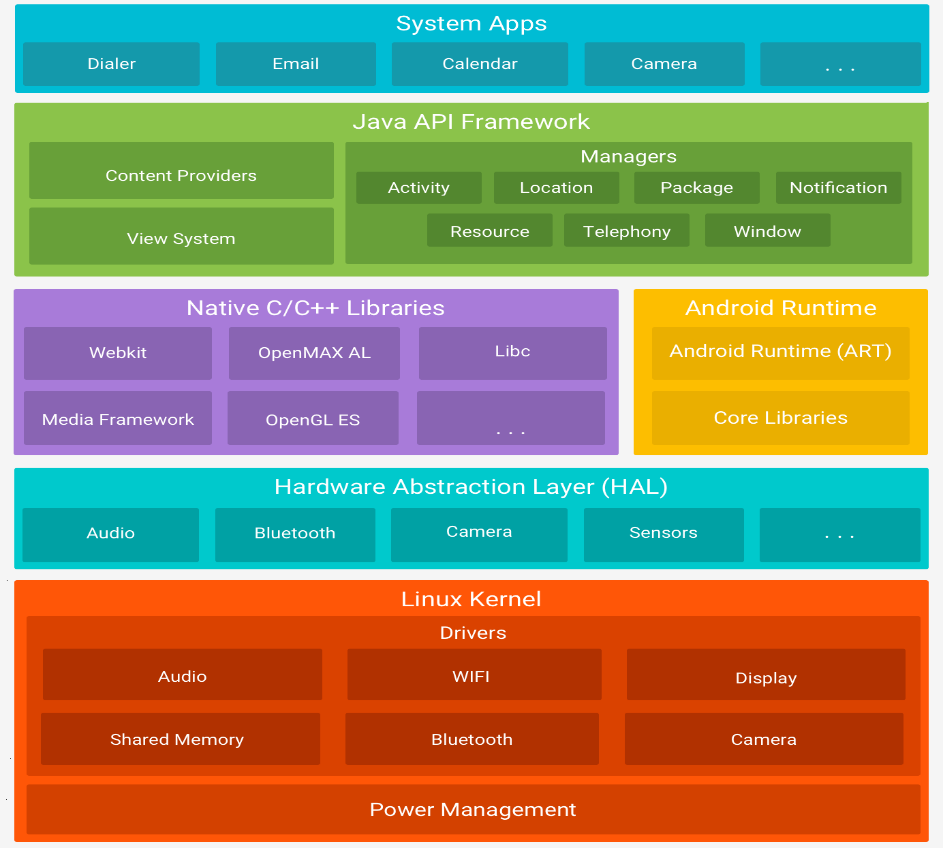
1. *Framework-ul Java API*

Întregul set de funcționalități ale sistemului de operare Android este disponibil dezvoltatorilor prin intermediul unor API-uri scrise în Java. Acestea includ:

* Un sistem de View-uri, bogat și extensibil, pentru construirea unei interfețe cu utilizatorul, ce include liste, căsuțe de text, butoane, precum și un browser web embedded
* Un Manager al Resurselor, ce face posibil accesul la resurse ce nu conțin cod, precum constante de tip șir de caractere, elemente de grafică, fișiere cu schița interfeței (layout)
* Un Manager al Notificărilor, ce le permite aplicațiilor să afișeze notificări în bara de status
* Un Manager al Activităților, ce gestionează ciclul de viața al aplicațiilor și pune la dispoziție o stivă de navigare
* Furnizori de conținut (Content Providers), ce fac posibil ca aplicațiile să acceseze date din alte aplicații sau să își distribuie propriile date

1. *Aplicațiile de sistem*

Platforma Android vine cu o serie de aplicații de bază pentru email, mesajerie SMS, calendar, browser de internet, contacte etc. Aplicațiile incluse în platformă nu au un status special față de cele pe care utilizatorul le poate instala, deci, de exemplu, acesta poate alege propriul broswer implicit, în defavoarea celui preinstalat.

Aplicațiile de sistem functionează atât ca aplicații pentru utilizator, cât și ca surse cheie de capabilități pe care dezvoltatorii le pot accesa din propriile aplicații. De exemplu, în cazul in care aceștia vor ca aplicația creata să trimită mesaje SMS, nu trebuie creată functionalitatea de la zero, ci doar invocată aplicația preinstalata care face deja acest lucru.

*Fig. 3.1.1.1. - Diagrama arhitecturii Android* [4]

**Componentele unei aplicații Android**

O aplicație Android este alcătuită din blocuri numite componente. Fiecare componentă este un punct de intrare prin care sistemul sau utilizatorul poate deschide aplicația și este definită în fișierul AndroidManifest.xml.

Există 4 tipuri diferite de componente, fiecare având propriul scop și un ciclu de viață distinct, ce definește felul în care componenta este creată și distrusă:

1. *Activitățile*

O activitate este un punct de intrare pentru interacțiunea cu utilizatorul, fiind un singur ecran cu o interfață. De exemplu, o aplicație de citire a presei poate avea o activitate care arată o listă cu articole și o altă activitate pentru citirea lor. Deși activitățile funcționează împreună pentru a oferi o experiență continuă, acestea sunt independente. Astfel, o altă aplicație poate porni oricare dintre aceste activități, atât timp cât aplicația de citire a presei o permite. Pornirea unuei activități dintr-o altă activitate se face printr-un Intent.

1. *Serviciile*

Serviciul reprezintă o componentă care se execută în background, realizând operații de lungă durată, fără a avea o interfață cu utilizatorul. De exemplu, un serviciu poate reda muzică în fundal în timp ce utilizatorul se află într-o altă aplicație sau poate prelua date din rețea fără a bloca intereacțiunea utilizatorului cu o activitate.

Serviciile sunt executate în firul de execuție principal, dacă nu se specifică altfel. Din motive de performanță, serviciile care folosesc intensiv resursele sistemului trebuie pornite într-un fir nou de execuție, pentru a nu bloca firul de execuție principal al aplicației.[5]

1. *Broadcast Receivers*

Componenta Broadcast Receiver este cea care recepționează notificări din sistem, precum ar fi un mesaj SMS recepționat și baterie scăzută, sau de la alte aplicații, precum în cazul în care o aplicație le transmite altora că anumite date au fost descărcate și pot fi folosite. Notificările pot fi trimise inclusiv aplicațiilor care nu rulează la momentul respectiv.

Un broadcast receiver nu implementează o interfață cu utilizatorul, dar poate crea o notificare în bara de status.

1. *Furnizorii de conținut (Content Providers)*

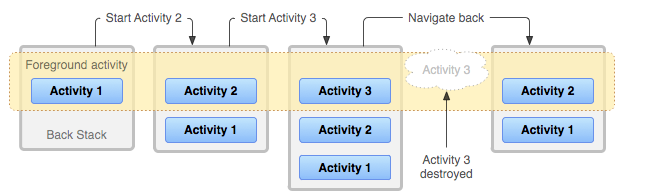
Componenta Content Provider permite transferul de date de la o aplicație la alta pe baza unor cereri. Aceasta face posibilă centralizarea datelor într-un singur loc, date ce pot fi accesate de aplicații diferite, în momentul în care fiecare are nevoie, deci se comportă ca un bază de date ce poate fi interogată, editată, în care se pot adaugă sau șterge intrări. Cel mai adesea, datele sunt stocate într-o bază de date de tipul SQLite, dar pot fi salvate și în fișiere sau chiar în rețea. [5]

**Ciclul unei aplicații**

În momentul în care utilizatorul deschide o aplicație, task-ul (o colecție de activități cu care interacționează utilizatorul în scopul îndeplinirii unei sarcini) acesteia este adus în foreground. Dacă nu există niciun task corespunzător (adică aplicația nu a fost utilizată recent), un nou task e creat, iar activitatea principală a aplicației e deschisă și este pusă ca prim element în stivă.

Dacă activitatea curentă pornește o noua activitate, cea din urmă este pusă în vârful stivei și preia primul plan. Activitatea precedentă rămâne în stivă, dar este oprită. Sistemul reține starea corespunzătoare interfeței cu utilizatorul din momentul opririi. În momentul în care utilizatorul apasă butonul Back, activitatea curentă e scoasă din stivă și este distrusă, iar activitatea precedentă se continuă, restaurându-se starea precedentă a interfeței cu utilizatorul.

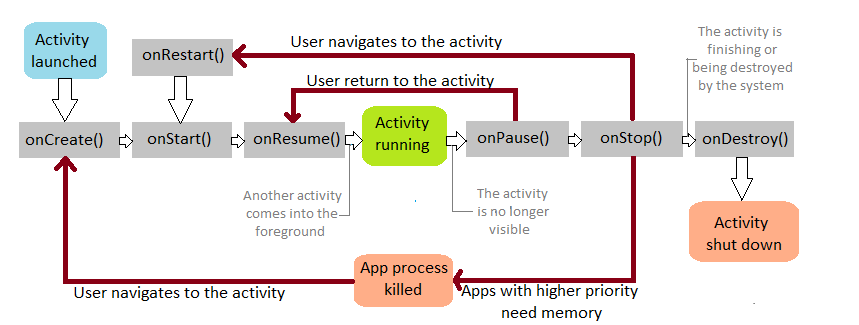
Astfel, activitățile din stivă nu sunt niciodată rearanjate, ci doar adăugate (atunci când sunt pornite din activitatea curentă) sau șterse (atunci când utilizatorul apasă butonul Back).



*Fig. 3.1.1.2. – Stiva de activități* [6]

**Ciclul de viață al unei activități**

În timp ce userul navighează printr-o, în afară și înapoi în aplicație, instanțele activităților acesteia trec prin diferite stări ale unui ciclu de viață.



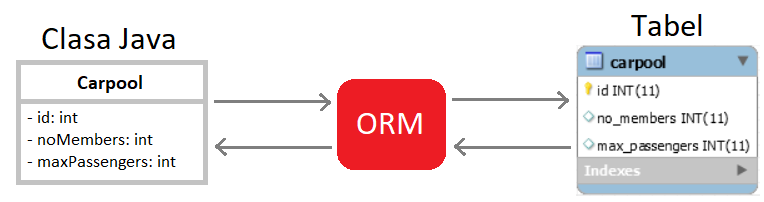
*Fig. 3.1.1.3. – Ciclul de viață al unei activități* [6]

## **Tehnologii folosite pe partea de server (server-side)**

### **JPA**

**Object-relational Mapping (ORM)**

Object-relational Mapping este procesul de dezvoltare și mentenanță a relației dintre obiecte Java și o bază de date relațională. De obicei, numele clasei obiectului corespunde numelui tabelului, iar fiecare câmp reprezintă o coloană. Odată creat tabelul, fiecare rând corespunde unei instanțe din aplicație.



*Fig. 3.2.1.1. – Maparea unei clase Java la un tabel; numele clasei corespunde numelui tabelului, iar numele câmpurilor - coloanelor*

**Java Persistence API (JPA)**

Java Persistence API reprezintă o specificație pentru accesul, persistența și administrarea datelor între obiecte simple Java (POJO – Plain Old Java Objects) și o bază de date relațională.

Configurarea componentelor se face în cadrul fișierelor unde sunt definite clasele, prin intermediul adnotărilor. Acestea sunt poziționate înainte clasei, a atributelor sau a metodelor la care se referă. Câteva adnotări de bază sunt prezentate în tabelul 1, totalitatea lor aflându-se în pachetul javax.persistence.

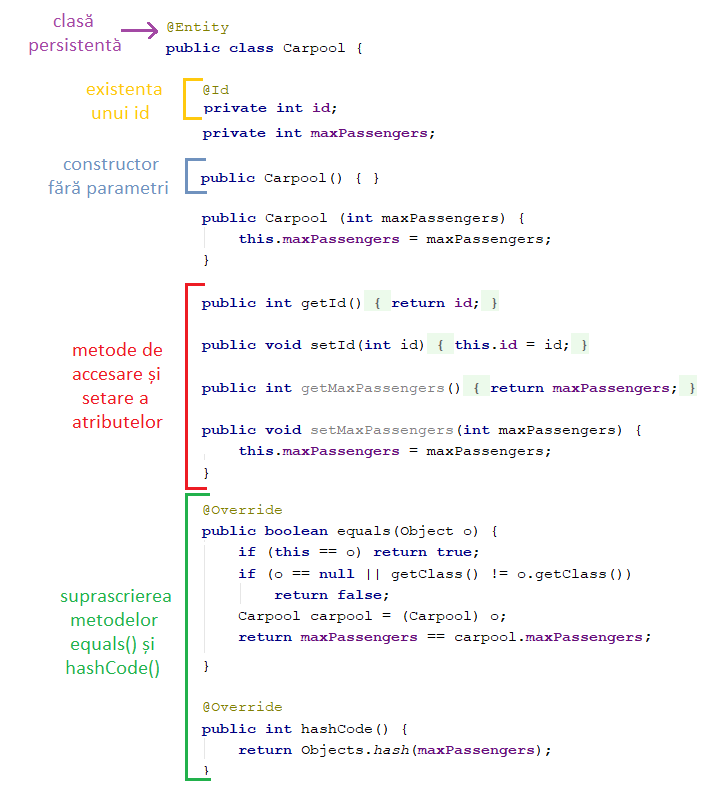
|  |  |  |
| --- | --- | --- |
| Adnotarea | Cui se aplică (Clasă/Câmp/Metodă) | Descriere |
| @Entity | Clasă | Specifică faptul că clasa este persistentă/mapată la un tabel |
| @Table | Clasă | Definește numele tabelului |
| @Column | Câmp/Metodă | Definește numele coloanei |
| @Id | Câmp/Metodă | Definește un câmp ca fiind identitatea unei clase (cheie primară) |
| @GeneratedValue | Câmp/Metodă | Valoarea câmpului este generată |
| @Embeddable | Clasă | Definește o clasă a cărei instanțe sunt stocate ca o parte a unei entități și reprezintă identitatea entității (cheie primară compusă) |
| @Embedded | Câmp/Metodă | Definește un câmp persistent al unei entități a cărei valoare e o instanță a unei clase embeddable. |
| @Basic | Câmp/Metodă | Definește un câmp ca persistent |
| @Transient | Câmp/Metodă | Câmpul nu este persistent (este tranzitiv) |
| @OneToOne | Câmp/Metodă | Definește acest câmp ca fiind într-o relație de 1 – 1 cu o altă entitate persistentă |
| @OneToMany | Câmp/Metodă | Definește acest câmp ca fiind într-o relație de 1 – N cu o altă entitate persistentă |
| @ManyToOne | Câmp/Metodă | Definește acest câmp ca fiind într-o relație de N – 1 cu o altă entitate persistentă |

*Tabel 1 – Adnotări JPA* [7]

**Hibernate**

Specificațiile JPA definesc cum se realizează persistența, dar este nevoie de o implementare pentru a se putea realiza operațiile propriu-zise.

Hibernate este un instrument ce realizează maparea obiectelor la baze de date relaționale, oferind o implementare pentru JPA.



*Fig. 3.2.1.2. – Normele de definire cu Hibernate a unei clase persistente: adnotarea @Entity în fața clasei, existența unui identificator, a unui constructor fără parametri, a metodelor de accesare și setare a atributelor și suprascrierea metodelor equals() și hashCode()*

### **Spring Framework**

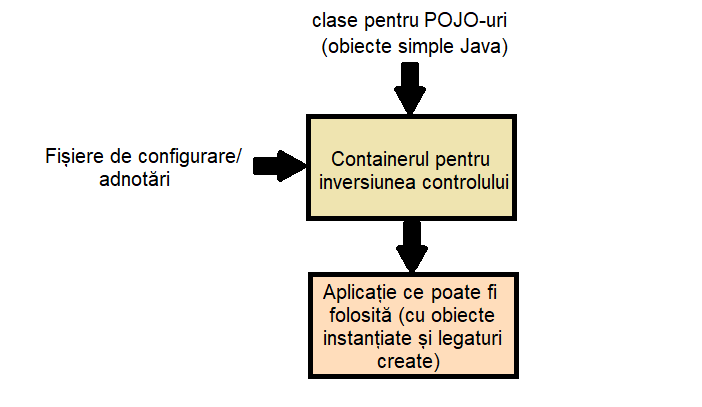
Lansat în anul 2003 de către Rod Johnson, Spring este un framework open-source ce asigură un model cuprinzător de programare și configurare pentru aplicațiile enterprise complexe. Deși ținta sa nu a fost niciun limbaj de programare, acesta a deveni popular în rândul comunității Java.

**Containerul pentru inversiunea controlului (dependency injection)**

Containerul pentru inversiunea controlului folosit de Spring face posibile configurarea și administrarea obiectelor Java folosind reflexia (reflection). Acesta este responsabil de crearea obiectelor, de apelul metodelor de inițializare a acestora și de configurarea lor prin crearea de legături.

Obiectele create de container se numesc obiecte administrate sau beans. Configurarea containerului se face prin încărcarea fișierelor XML (Extensible Markup Language) sau prin detectarea anumitor adnotări pe clasele de configurare. Aceste surse de date conțin definițiile bean-urilor, ce aduc informațiile necesare creării lor. [8]

Felul în care funcționează containerul este următoarul: dacă un proiect este format din două clase A și B, iar A conține o referință către B, obiectul de tip A este responsabil de obținerea referinței (a dependenței) de tip B. Astfel, codul este puternic cuplat și greu de testat. Folosind injectarea dependențelor, A nu va mai fi nevoit să își creeze dependențele, ci acestea vor fi injectate de către containerul Spring, responsabil de crearea obiectelor și a legaturilor dintre ele.



*Fig. 3.2.2.1. – funcționarea containerului pentru inversiunea controlului: se preiau POJO-uri și, pe baza fișierelor de configurare sau a adnotărilor, containerul instanțiază obiectele simple Java și creează legaturile dintre acestea, pentru ca aplicația să poată fi folosită*

**Framework-ul pentru accesarea datelor**

Framework-ul pentru accesarea datelor oferit de Spring soluționează dificultățile întâlnite atunci când se lucrează cu bazele de date, prin oferirea următoarelor functionalități:

* administrarea resurselor, prin obținerea și eliberearea automată a resurselor bazei de date
* tratarea excepțiilor referitoare la accesarea datelor
* participarea la tranzacții
* despachetarea resurselor venite de la baza de date sub forma unor obiecte specifice bazei de date [8]

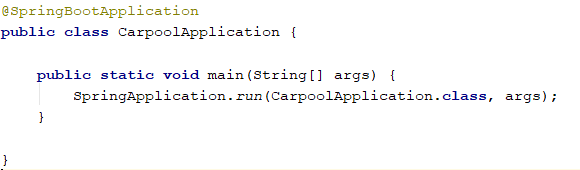
### **Spring Boot**

Spring Boot reprezintă o soluție a framework-ului Spring pentru crearea de aplicații de calitate de sine stătătoare care pot fi rulate așa cum sunt generate. Aceste aplicații sunt preconfigurate, reducându-se timpul necesar dezvoltării unui schelet și evitându-se complexitate fișierelor de configurare XML.

Spring Boot configurează automat aplicația pe baza dependențelor din proiect prin folosirea adnotării *@EnableAutoConfiguration*. De exemplu, dacă o bază de date MySQL se află în classpath, dar nu s-a stabilit nicio conexiune la baza de date, Spring Boot configurează în mod automat o bază de date în memorie (in-memory database).

Punctul de intrare al unei aplicații Spring Boot o reprezintă clasa ce conține adnotarea @*SpringBootApplication* și metoda main [9]. Adnotarea @*SpringBootApplication* include 3 adnotări:

* @*EnableAutoConfiguration*, descrisă mai sus
* @*ComponentScan*, folosită la scanarea pachetului curent și a tuturor subpachetelor în momentul inițializării aplicației, pentru a detecta toate bean-urile (a obiectelor care formează scheletul aplicației, instanțiate, asamblate și gestionate de containerul pentru inversiunea controlului în Spring)
* @*Configuration*, ce indică faptul că o anumită clasă poate fi folosită de containerul pentru inversiunea controlului ca sursă de definiție a bean-urilor



*Fig. 3.2.3.1. – Punctul de intrare al unei aplicații Spring Boot*

# **IMPLEMENTARE**

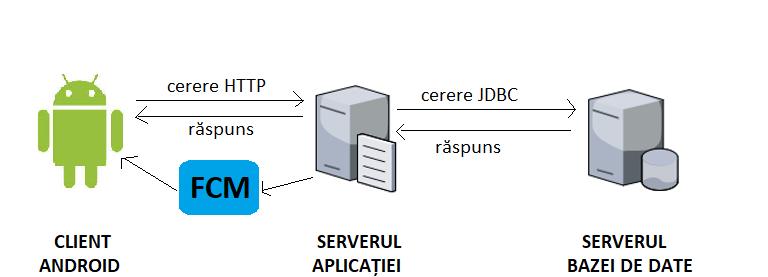
## **Arhitectura**

Componentele principale ale aplicației sunt:

* Clientul Android, responsabil de interacțiunea cu utilizatorul: de colectarea datelor de înregistrare, ale datelor de autentificare și ale datelor referitoare la zilele și orele pentru care utilizatorii solicită transport, de afișarea traseului în momentul în care o cursă este stabilită și de urmărirea locației în momentul efectuării cursei
* Serverul aplicației, implementat în Java cu framework-ul Spring, responsabil de generarea carpool-urilor, de înștiințatea utilizatorilor în momentul generării acestora, dar și de maparea claselor la tabelele bazei de date prin intermediul Hibernate (JPA)
* Serverul bazei de date, reprezentat de MySQL, responsabil de stocarea datelor

Comunicarea dintre clientul Android și serverul aplicației se realizează în două moduri, în funcție de cine inițiază comunicarea:

1. Dacă este vorba de clientul Android, acesta face cereri HTTP de tipurile GET (în cazul autentificării), POST (în cazul înregistrării unui utilizator) și PUT (în cazul modificării profilului utilizatorului, respectiv ale datelor referitoare la nevoile sale de transport) la adresa la care rulează serverul. Serverul răspunde doar în cazul autentificării, trimițând datele utilizatorului, în cazul în care autentificarea a avut loc cu succes.
2. Dacă este vorba de serverul aplicației, în momentul stabilirii unui carpool, acesta face o cerere HTTP de tipul POST la adresa pe care rulează serverul Firebase Cloud Messaging, împreună cu un token asociat dispozitivului către care se dorește trimiterea mesajului. Firebase Cloud Messaging realizează transferul mesajului către dispozitivul pe care rulează aplicația client.

Comunicarea dintre serverul aplicației și serverul bazei de date este unidirecțională, primul dintre acestea fiind cel care realizează cererile prin intermediul Java Database Connectivity (JDBC). Driverul JDBC stabileste protocolul de comunicare, asigurând conectarea directă la baza de date prin conversia apelurilor JDBC în apeluri spefice bazei de date.

*Fig. 4.1.1. – Componentele aplicației și comunicare dintre acestea*

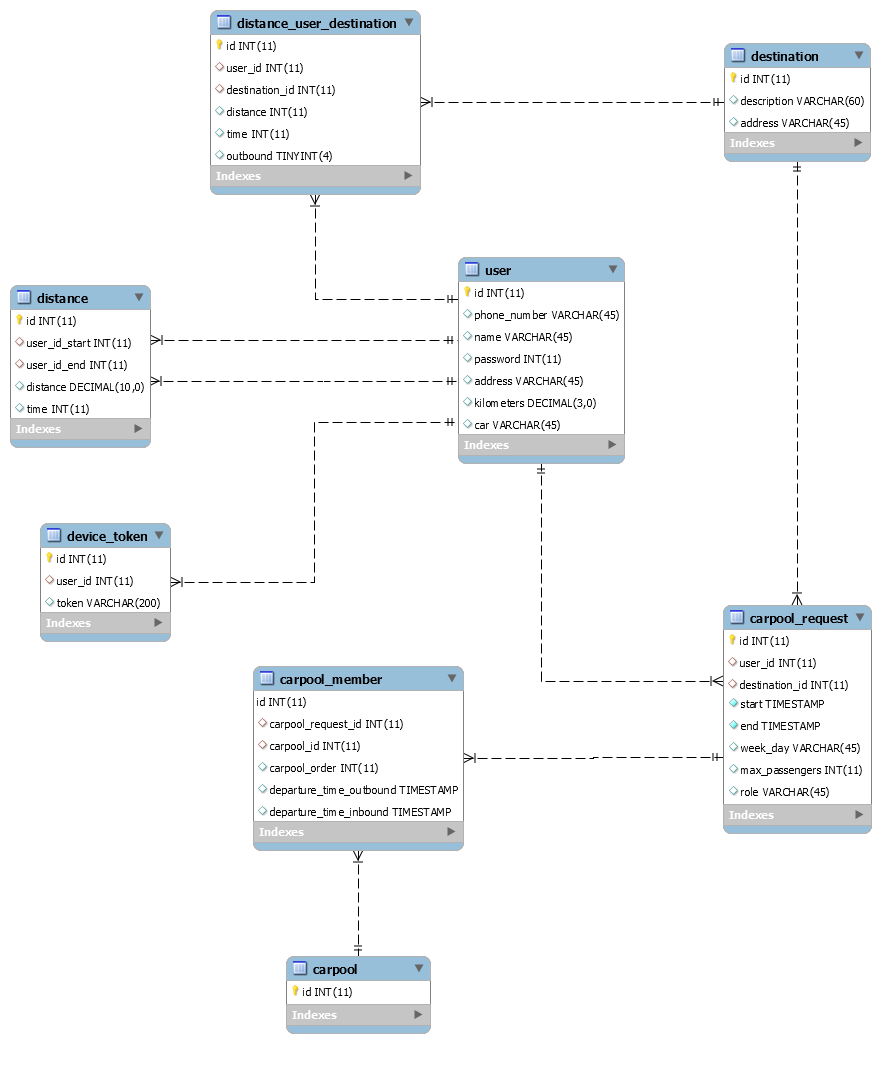
## **Baza de date**

Baza de date este formată din 8 tabele prezentate în figura 4.2.1. Acestea sunt următoarele:

* **user**, ce conține informații despre fiecare utilizator, printre care numele, numărul de telefon (unic), codul hash al parolei, adresa și numărul de kilometri parcurși ca șofer
* **device\_token**, ce reține token-ul dispozitivului pe care este conectat un utilizator; un singur utilizator corespunde unui singur dispozitiv și invers
* **distance**, pentru a cunoaște distanța dintre utilizatori și timpul necesar parcurgerii acesteia cu mașina; tabela este actualizată la fiecare înregistrare a unui utilizator cu distanțele și timpii dintre acesta și toți ceilalți utilizatori existenți; se rețin măsurătorile în ambele direcții, ca urmare intrarea (utilizator\_1, utilizator\_2) este diferită de (utilizator\_2, utilizator\_1)
* **destination**, ce conține informațiile referitoare la destinațiile posibile (adresă și descriere)
* **distance\_user\_destination**, analog **distance**, doar că de această dată nu e vorba de doi utilizatori, ci de un utilizator și o destinație
* **carpool\_request**, ce corespunde unei cereri de carpool și cuprinde orele de plecare/întoarcere, destinația, utilizatorul care a făcut cererea, rolul pe care îl poate indeplini utilizatorul, precum și numărul maxim de pasageri pe care îi poate lua (0 în cazul utilizatorilor ce nu pot fi șoferi)
* **carpool**
* **carpool\_member**, ce reține informații despre un membru al unui carpool: cererea corespunzătoare, carpool-ul, orele de plecare/întoarcere, precum și ordinea în carpool

Relațiile dintre acestea sunt următoarele:

* între **user** și **device\_token** avem relație 1..1, adică un utilizator poate avea un singur token și un token un singur utilizator
* între **user** și **distance** avem relație n..n, adică un utilizator are mai multe distanțe și o distanță mai mulți utilizatori
* între **user** și **distance\_user\_destination** avem o relație 1..n, adică unui singur utilizator îi corespund mai multe distanțe între el și destinații
* între **user** și **carpool\_request** avem o relație 1..n, adică un utilizator poate să aibă mai multe cereri, dar o cerere corespunde unui singur utilizator
* între **destination** și **distance\_user\_destination** avem o relație 1..n, adică unei destinații îi corespund mai multe distanțe între ea și utilizatori
* între **destination** și **carpool\_request avem** o relație 1..n, adică o destinație corespunde mai multor cereri, dar o cerere are o singură destinație
* între **carpool\_member** și **carpool\_request** avem o relație 1..1, adică unui membru al unui carpool îi corespunde o singură cerere și invers
* între **carpool** și **carpool\_member** avem o relație 1..n, adică unui carpool îi corespund mai mulți membri, dar unui membru un singur carpool



*Fig. 4.2.1. – Tabelele din baza de date și legăturile dintre ele*

## **Clientul Android**

Aplicația Android este compusă din 5 activități, fiecare cu propriile responsabilități, definite în subcapitolele următoare: MainActivity (4.2.1.), LoginActivity (4.2.2.), SignUpActivity (4.2.3.), TabsActivity (4.2.4) și MapNavigationActivity (4.2.5). Acestea extind clasa de bază *android.support.v7.app.AppCompatActivity* și suprascriu metode precum onCreate(), onCreateOptionsMenu(), onOptionsItemSelected().

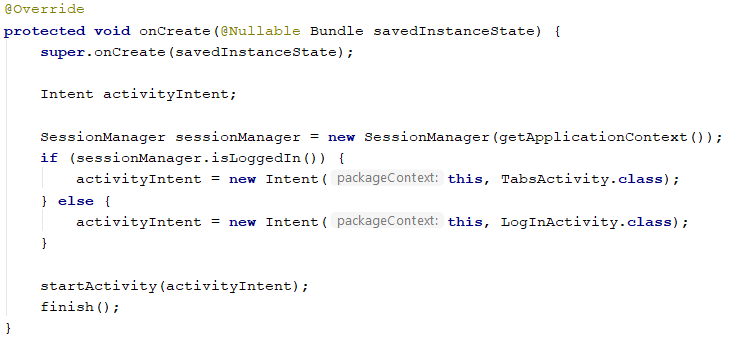
Pentru gestionarea unei sesiunii de autentificare, ce face posibilă reținerea locală, pe dispozitivul de unde s-a făcut autentificarea, a datelor utilizatorului, s-a creat un manager de sesiune, detaliat în subcapitolul 4.2.5.

Conexiunea cu serverul se realizează, la nivelul clientului, prin intermediul unei librării HTTP, detaliată în subcapitolul 4.2.6.

### **Pornirea aplicației – MainActivity**

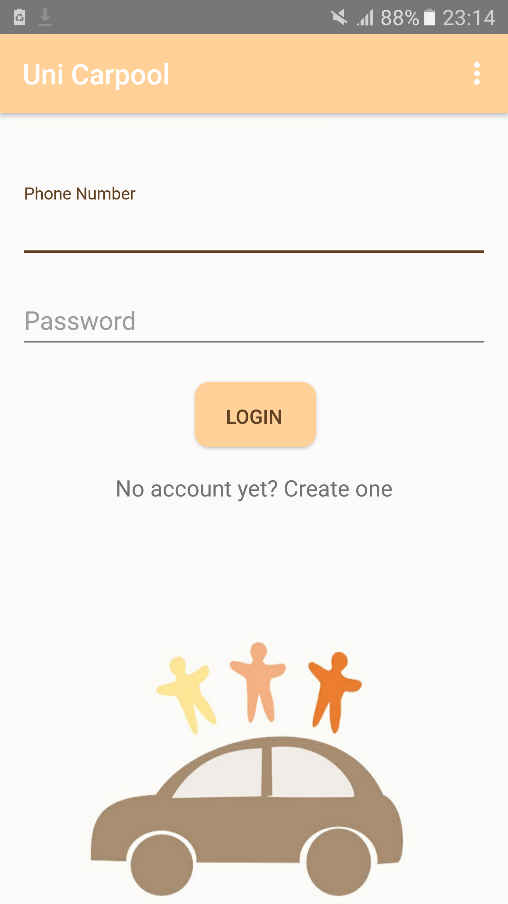
În momentul lansării aplicației, se deschide activitatea principală, MainActivity. Aceasta nu are interfață cu utilizatorul, având doar rolul de a-l redirecționa către activitatea corespunzătoare: în cazul în care acesta este autentificat, redirecționarea se va face către TabsActivity, iar, în caz contrar, către LoginActivity.

În figura 10, reprezentând codul metodei onCreate() ce aparține activității principale, se observă faptul că verificarea autentificării se face prin intermediul unei clase de gestiune a sesiunii, ce va fi descrisă în capitolul 4.2.6. Apelul metodei finish() face ca activitatea să fie distrusă și, implicit, să dispară din stiva activităților. Astfel, în momentul în care utilizatorul merge înapoi din TabsActivity sau LogInActivity, nu se va mai deschide activitatea MainActivity, ci se va ieși din aplicație.



*Fig. 4.3.1.1. – metoda onCreate() corespunzătoare activității principale*

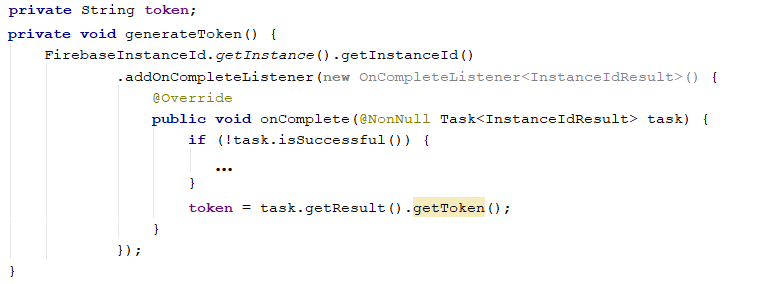
### **Autentificarea – LogInActivity**

Precum sugerează numele, activitatea LogInActivity este folosită la autentificarea utilizatorilor. Autentificarea se face pe baza numărului de telefon și a parolei stabilite în momentul înregistrării.

*Fig. 4.3.2.1. – formularul de autentificare din activitatea LogInActivity*

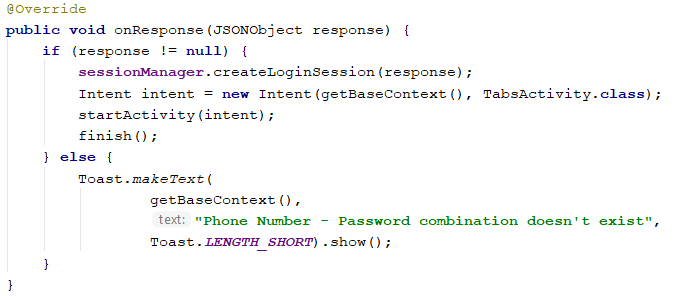
În momentul introducerii numărului de telefon și al parolei, apăsarea butonului Login va genera un apel HTTP de tipul GET, prin intermediul bibliotecii Volley, la adresa [http://localhost:8090/login/{numar}/{parola}/{token}](http://localhost:8090/login/%7bnumar%7d/%7bparola%7d/%7btoken%7d). Nu se va transmite parola propriu-zisă, ci codul hash al acesteia.

Se observă că, pe lângă parolă, se transmite și token-ul asociat aplicației respective, ce o indentifică în mod unic. Acesta este stocat în baza de date pentru a putea fi transmise ulterior mesaje de la server către aplicația. Transmiterea sa la fiecare autentificare are rolul de a asocia fiecare utilizator cu un singur dispozitiv. De exemplu, dacă pe un telefon avem aplicația cu token-ul T și se autentifică utilizatorul A, în baza de date T este asociat lui A. În cazul unei deconectări și a unei noi autentificări a utilizatorului B, în baza de date T nu ii va mai fi asociat lui A, ci lui B. În cazul în care utilizatorul A se conectează de pe 2 dispozitive, în baza de date se va reține doar ultimul dispozitiv.



*Fig. 4.3.2.2. – generarea token-ului asociat instanței activității aflată pe dispozitivul pe care se rulează codul.*

În cazul în care autentificarea are loc cu succes, răspunsul venit de la server este reprezentat de o instanță a clasei JSONObject ce conține informațiile utilizatorului. În acest moment, obiectul care gestioneză sesiunea (SessionManager) va fi înștiințat că a avut loc o autentificare, fiindu-i transmise și informațiile utilizatorului. De asemenea, utilizatorul va fi redirecționat către o nouă activitate, TabsActivity.

În cazul în care autentificarea nu are loc cu succes, utilizatorul va fi înștiințat printr-un toast că nu există un utilizator cu acea pereche număr de telefon – parolă. 

*Fig. 4.3.2.3. – crearea sesiunii de autentificare și deschiderea noii activități, respectiv afișarea toastului*

Dacă utilizatorul nu este înregistrat, apăsarea textului *No account yet? Create one* îl va redirecționa către activitatea unde o poate face.

### **Înregistrarea unui nou utilizator – SignUpActivity**

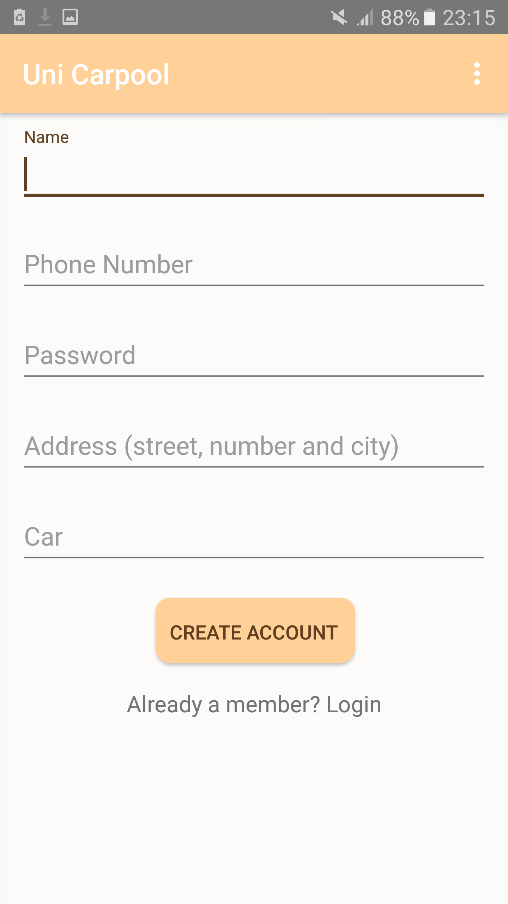
Activitatea SignUpActivity are rolul de a-i oferi utilizatorului un formular pe baza căruia se poate înregistra pentru a folosi aplicația.

Informațiile pe care le introduce utilizatorul includ numele, numărul de telefon, o parolă, adresa unde locuiește, precum și mașina pe care o posedă. Apăsarea butonului Create account va genera un apel HTTP de tipul POST la adresa <http://localhost:8090/users>. Din nou, se va transmite codul hash al parolei.

În cazul în care conexiunea la server și salvarea datelor se termină cu succes, se va afișa un toast cu un mesaj corespunzător: „Successfully registered“ („Întregistrat cu succes“).

În cazul în care utilizatorul introduce un număr de telefon ce corespunde deja unui cont, se va primi un mesaj de eroare: "Account already associated with the provided phone number.".

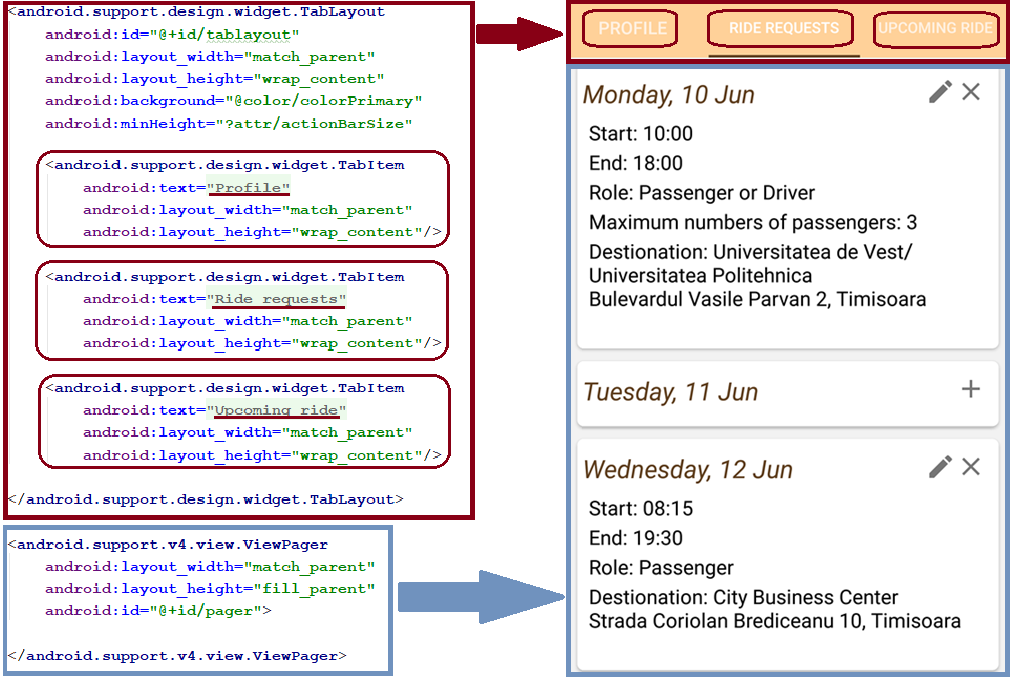
Apăsarea textului *Already a member? Login* va redirecționa utilizatorul către secțiunea de autentificare.



*Fig. 4.3.3.1. – formularul de autentificare din activitatea SignUpActivity*

### **Interfața după momentul autentificării - TabsActivity**

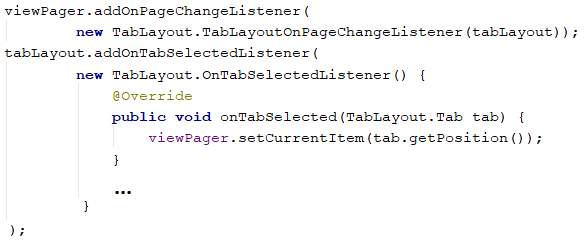
Activitatea TabsActivity cuprinde acțiunile principale pe care le poate face utilizatorul după autentificare, dispuse sub unor pagini, pe care utilizatorul le accesează fie apăsând pe numele corespunzător din bara de meniu, fie dând cu degetul la dreapta sau la stânga pe ecran. Activitatea cuprinde 2 elemente:

* Un TabLayout, prin intermediul căruia se creează o lista tabulară cu numele secțiunilor: Profile, Ride Requests, Upcoming ride
* Un ViewPager, un layout care gestionează felul în care sunt afișate secțiunile activității; acestuia îi este atașat un FragmentPagerAdapter, prin care sunt populate secțiunile propriu-zise

*Fig. 4.3.4.1. – interfața TabsActivity: lista tabulară cu titlurile paginilor (TabLayout) și pagina proriu-zisă (ViewPager)*

Cele trei fragmente cuprinse în activitatea TabsActivity corespund paginii profilului utilizatorului (ProfileFrgament), paginii în care se realizează cererile de carpool (RequestsFragment) și paginii în care este afișată repartizarea pentru ziua în curs (UpcomingRideFragment).

Pentru ca elementul arătat ca fiind selectat de TabLayout să corespundă cu cel afișat de către ViewPager este nevoie ca fiecare componentă să urmărească felul în care utilizatorul interacționează cu cealaltă componentă. Acest lucru se realizează prin atașarea unor obiecte de tip Listener, care “ascultă” acțiunile care se petrec în TabLayout, respectiv în ViewPager și generează răspunsuri din partea celeilalte componente.



*Fig. 4.3.4.2. – codul care ne asigură că ViewPagerul și TabLayoutul sunt sincronizate, prin afișarea paginii și a titlului corespunzător acesteia*

Secțiunile sunt reprezentate de fragmente, ce pot fi văzute ca niște sub activități, cu interfețe diferite, cu propriul ciclu de viață și propriile evenimente de intrare. Un fragment face parte întotdeauna dintr-o activitate, ciclul său de viață fiind direct afectat de ciclul de viață al activității gazdă. De exemplu, în momentul în care activitatea este pusă pe pauză sau este distrusă, la fel sunt și toate fragmentele pe care le conține. Cu toate acestea, cât timp o activitate este pornită, fragmentele pot fi manipulate independent.

Un fragment va extinde clasa Fragment și va implementa trei metode din ciclul de viața al fragmentului:

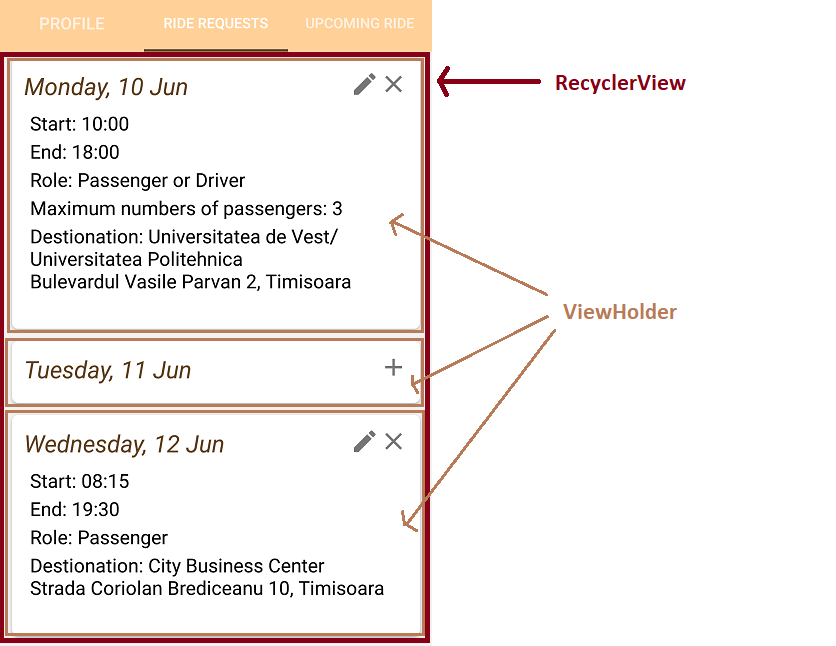
* onCreate(), metodă apelată de sistem în momentul creării fragmentului; acesta este locul unde sunt inițializate componentele sale esențiale, ce vor fi păstrate chiar dacă un fragment este în pauză sau este oprit, apoi repornit.
* onCreateView(), metodă apelată de sistem în momentul în care fragmentul își creează pentru prima dată interfața cu utilizatorul; această metodă returnează un View ce reprezintă piesa de bază (**ROOT**) a layoutului fragmentului sau null în cazul în care fragmentul nu furnizează o interfață
* onPause(), metodă apelată de sistem ca prim semn că utilizatorul părăsește un fragment, deși acest lucru nu presupune întotdeauna distrugerea lui; acesta este locul unde trebuie salvate orice modificări ce se doresc a fi menținute și în cadrul unor sesiuni ulterioare (pentru că utilizatorul ar putea să nu se mai întoarcă în activitate)

### **Pagina de profil a utilizatorului – ProfileFragment**

Așa cum îi arată numele, ProfileFragment pune la dispoziția utilizatorului o pagină cu datele sale. Acestea pot fi modificate prin completarea câmpurilor corespunzătoare și apăsarea butonului Update, care va genera un apel HTTP prin intermediul Volley, descris în capitolul 4.2.6. La fel ca în cazul creării unui nou profil, dacă numărul de telefon introdus este asociat deja unui cont, actualizarea nu va avea loc, iar utilizatorul va primi un Toast prin care va fi înștiințat de acest lucru.

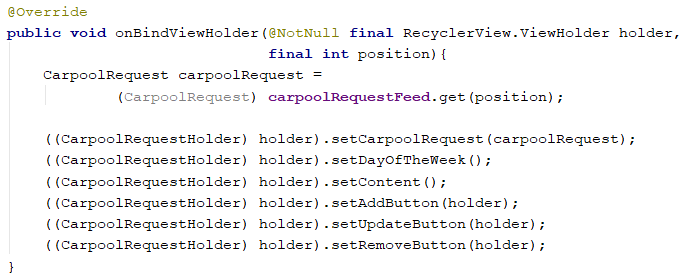
### **Pagina de cereri de carpool - RequestFragment**

Pagina RequestFragment este cea în care utilizatorul poate introduce datele referitoare la cursele pe care le solicită. Pentru afișarea zilelor pentru care se pot pune cereri s-a folosit un RecyclerView, un container pentru o listă de elemente ajustabilă ca dimensiune. Acesta face posibilă modificarea continuă a imaginii ecranului, pe măsură ce utilizatorul se deplasează cu degetul în sus sau în jos, pentru a putea vedea lista completă de elemente.

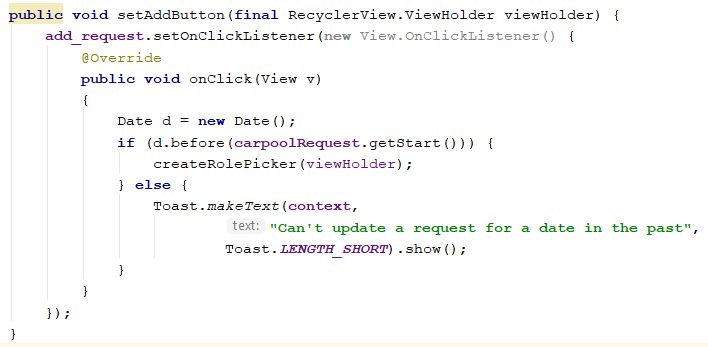
Elementele din listă sunt reprezentate de obiecte de tip “view holder”. Astfel, în cazul aplicației, un element este un card ce arată informațiile referitoare la cererea de carpool din ziua respectivă. Obiectele sunt instanțe ale unei clase ce extinde RecyclerView.ViewHolder. În momentul creării RecyclerView-ului, acesta creează doar un număr de obiecte ViewHolder ce sunt vizibile pe ecran (de exemplu, două), plus încă unul sau două, de rezervă. Pe măsură ce utilizatorul de deplasează în josul listei, containerul creează noi obiecte sau le reciclează pe cele care nu mai sunt vizibile pe ecran, prin popularea lor cu informațiile obiectelor ce sunt vizibile. Deplasarea în susul listei va avea același efect.

*Fig. 4.3.6.1. – Dispunerea elementelor de tip ViewHolder într-un RecyclerView*

Obiectele din listă (de tip ViewHolder) sunt administrate de un adaptor, creat prin extinderea clasei RecyclerView.Adapter. Acesta creează obiecte pe măsură ce sunt cerute și le populează cu datele corespunzătoare, prin asocierea fiecaruia cu o poziție și apelul metodei onBindViewHolder(), care determină ce informații vor fi afișate pe baza poziției în listă.

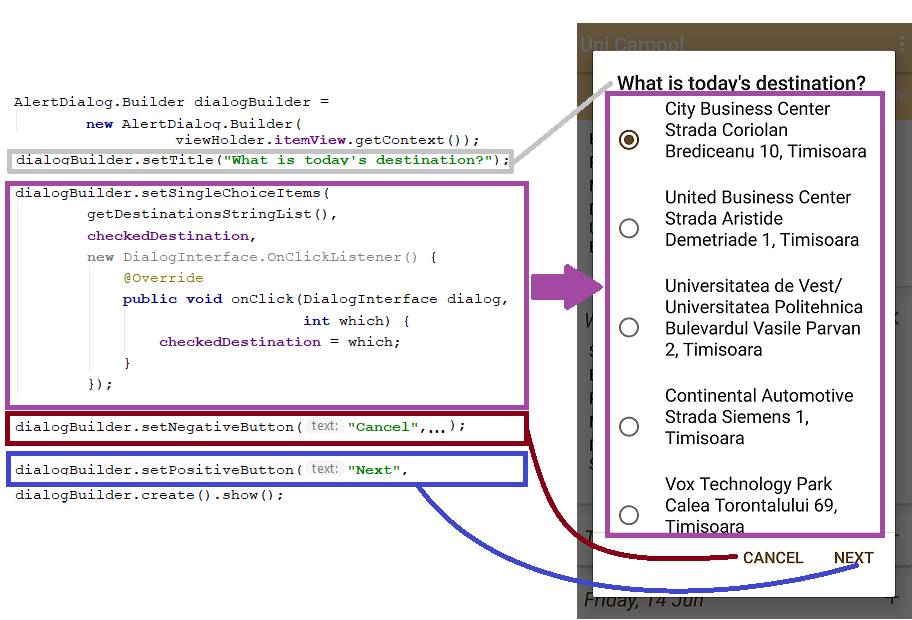


*Fig. 4.3.6.2. – Metoda onBindViewHolder din adaptorul corespunzător RecyclerView-ului, apelată în momentul în care utilizatorul se deplasează în susul și în josul ecranului pentru a popula elementul vizibil pe ecran*

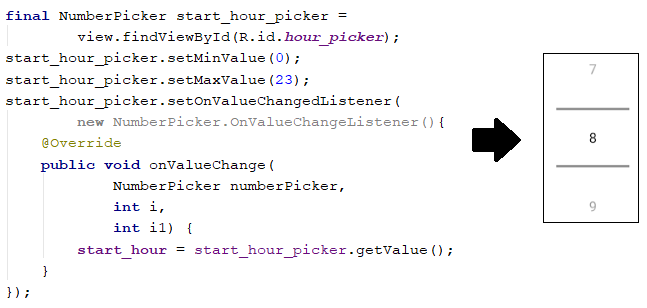
Elementele de tip ViewHolder sunt în număr de 5, corespunzătoare zilelor de luni, marți, miercuri, joi și vineri ale săptămânii curente (în cazul în care astăzi este cel târziu vineri), respectiv ale săptămânii viitoare, dacă este sâmbătă sau duminică. În momentul apăsării butoanelor de adăugare, modificare sau ștergere a unei cereri se va compara data corespunzătoare elementului cu data curentă. Dacă data curentă este ulterioară primeia, nu va fi posibilă adăugare/modificarea/ștergerea cererii, utilizatorul primind un mesaj prin intermediul unui Toast.

*Fig. 4.3.6.3. – Stabilirea funcționării butonului de adăugare a unei cereri: dacă data pentru care se face adăugarea este din trecut, se va anunța utilizatorul că nu poate fi realizată operația cerută. În caz contrar, se apelează funcția care dă startul adăugării unui cereri.*

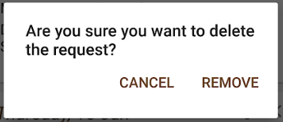
În cazul în care se poate realiza adăugarea/modificarea, se vor crea căsuțe de dialog (DialogBox) prin care utilizatorul alege nevoile/cerințele pentru cursa respectivă: rolul pe care este dispus să îl îndeplinească (pasager, șofer sau oricare din cele două), numărul de pasageri pe care îi poate lua (acest pas este omis în cazul pasagerilor), destinația finală (una dintre cele 5 disponibile), precum și ora de plecare, respectiv ora de întoarcere.

Rolul, numărul de locuri libere și destinația sunt alese prin intermediul unor liste în care un singur element poate fi bifat.

*Fig. 4.3.6.4. – Crearea căsuței de dialog (DialogBox) corespunzătoare alegerii destinației: selectarea titlului, a listei cu numele destinațiilor, din care doar un element poate fi ales, setaria acțiunilor butonului pozitiv (Next), respectiv negativ (Cancel)*

Orele sunt alese prin intermediul a două elemente de tip NumberPicker, unul corespunzător orei (cu valori între 0 și 23) și unul corespunzător minutelor (cu valori între 0 și 59). Se fac verificări prin care nu se poate finaliza o cerere în care ora de plecare depășește ora de întoarcere, cerândui-se utilizatorului să aleagă alte ore.

*Fig. 4.3.6.5. – Crearea NumberPicker-ului corespunzător orei de plecare; valoarea minimă este 0, iar valoarea maximă este 23; schimbarea valorii curente, prin mutarea degetului în sus sau în jos, determină actualizarea orei din cerere*

În cazul ștergerii, se va crea un DialogBox prin care utilizatorul este întrebat încă o dată dacă este sigur că dorește ștergerea cererii respective.

*Fig. 4.3.6.6. – Căsuța de dialog prin care utilizatorul este rugat să confirme faptul că dorește ștergerea unei cereri*

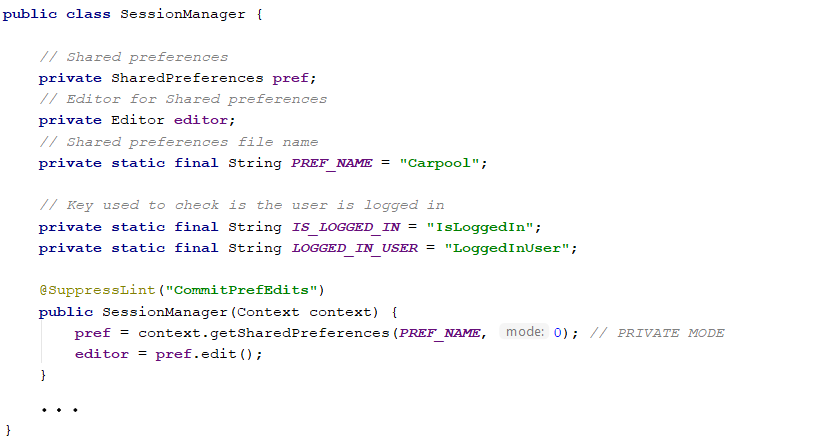
Ultimul pas în cadrul adăugării/modificării/ștergerii este crearea unui apel HTTP corespunzător prin intermediul Volley, descris în capitolul 4.2.8. Dacă lucrurile decurg bine, finalizarea apelului este marcată prin intermediul unui Toast primit de client, care îl anunța că modificarea a avut loc cu succes.

### **Pagina curselor din ziua respectivă – UpcomingRideFrament**

În momentul în care serverul realizează gruparea utilizatorilor în carpool-uri, trimiterea repartizării către utilizatori va determina încarărcea datelor referitoare la cursă în pagina UpcomingRideFragment. Aceasta conține o listă cu date referitoare la cursa din acea zi: numărul de membri ai carpool-ului, șoferul și pasagerii, destinația, precum și timpii de început ai curselor către și dinspre destinația din acea zi. Există, de asemenea, două butoane („See outbound ride“, respectiv „See inbound ride“), prin care se deschide MapNavigationActivity, ce face posibilă vizualizarea traseelor generate pentru cele două curse – de plecare, respectiv de întoarcere.

### **Vizualizarea traseului și efectuarea unei curse – MapNavigationActivity**

### **Managerul de sesiune SessionManager**

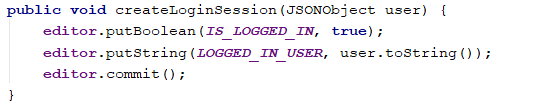
Pentru ca o sesiune de autentificare să fie valabilă după ce utilizatorul închide aplicația, este nevoie ca anumite informații să fie salvate și încărcate la următoarea utilizare. În acest scop, s-a folosit SharedPreferences, prin care se salvează perechi cheie-valoare de date primitive într-un fișier XML ce se află în director de date al aplicației.

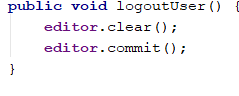
*Fig. 4.3.9.1. – Instanțierea SharedPreferences în SessionManager; numele fișierului este PREF\_NAME („Carpool“), iar modul este privat (fișierul e vizibil doar pentru către aplicația în cauză)*

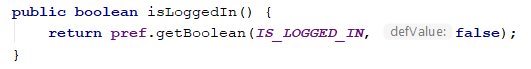
În momentul în care are loc cu succes o autentificare, acest lucru va fi vizibil în fișierul corespunzător SharedPreferences prin intermediul unei perechi cheie-valoare ({„IsLoggedIn“: true}. În plus, se salvează în fișier datele utilizatorului, pentru a nu fi nevoie să fie încărcate din backend la fiecare deschidere a aplicației, ci doar la autentificare.

În momentul deschiderii aplicației, se verifică existeța în fișier a câmpului „IsLoggedIn“. Dacă există, se returnează valoarea acestuia (true), iar, în caz contrar, false.

În momentul în care are loc deconectarea utilizatorului, conținutul fișierului este șters.







*Fig. 4.3.9.1 – funcțiile corespunzătoare autentificării, prin setarea câmpului IS\_LOGGED\_IN („IsLoggedIn“) ca adevărat (true), deconectării, prin ștergerea conținutului fișierului, și verificării autentificării, prin preluarea valorii câmpului IS\_LOGGED\_IN („IsLoggedIn“)*

### **Efectuarea apelurilor HTTP cu ajutorul Volley**

Volley este o librărie HTTP folosită de sistemele Android pentru realizarea de apeluri HTTP. În momentul în care se realizează un apel, acesta este adăugat în coada de apeluri deținută de Volley (RequestQueue). Coada are atât rolul de a trata apelurile, cât și de a intercepta răspunsurile. Pentru a nu se crea câte un RequestQueue pentru fiecare apel, s-a folosit tiparul de design singleton, prin care se creează o clasă VolleySingleton, căreia îi corespunde o singură coadă.

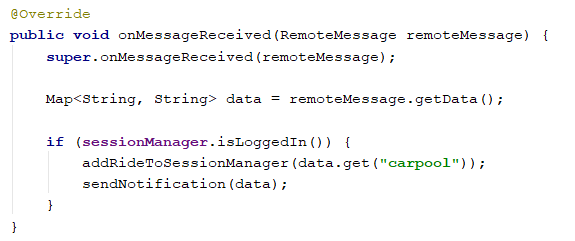
Pentru a putea fi posibilă parsarea răspunsului venit din rețea (NetworkResponse) și conversia sa la unul din tipurile Response<JSONObject>, respectiv Response<JSONArray>, a fost necesară crearea a două clase de cereri ce extind JsonObjectRequest (CustomJsonObjectRequest), respectiv JsonRequest<JSONArray> (CustomJsonArrayRequest) și suprascriu metoda parseNetworkResponse(NetworkResponse response).

În figura 25 se observă modul în care se adaugă o cerere de tipul CustomJsonObjectRequest în coada ce aparține clasei SingletonVolley. În primul rând, se creează un JSON corepunzător cererii, care se populează cu informațiile necesare (poate fi și null). Apoi, deoarece se așteaptă un răspuns JSONObject, se creează o cerere CustomJsonObjectRequest, ce primește ca parametrii: tipul apelului HTTP (GET, în cazul de față), adresa URL la care se face apelul, un obiect care tratează răspunsul venit, dar și un obiect care tratează posibila eroarea venită. Se obține instanța VolleySingleton (this este contextul activității de unde se face apelul), în a cărei coadă se adaugă cererea.

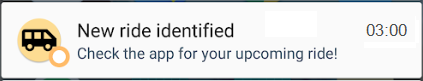


*Fig. 4.3.10.1. - Realizarea unui apel HTTP cu ajutorul Volley, prin crearea unui obiect de tipul CustomJsonObjectRequest, ce primește ca parametri tipul apelului HTTP, adresa URL la care se face apelul, cererea sub forma de JSON, precum și funcții de tratare a răspunsului, respective a erorii*

### **Serviciul de interceptare a mesajelor venite prin Firebase Cloud Messaging (FCM)**

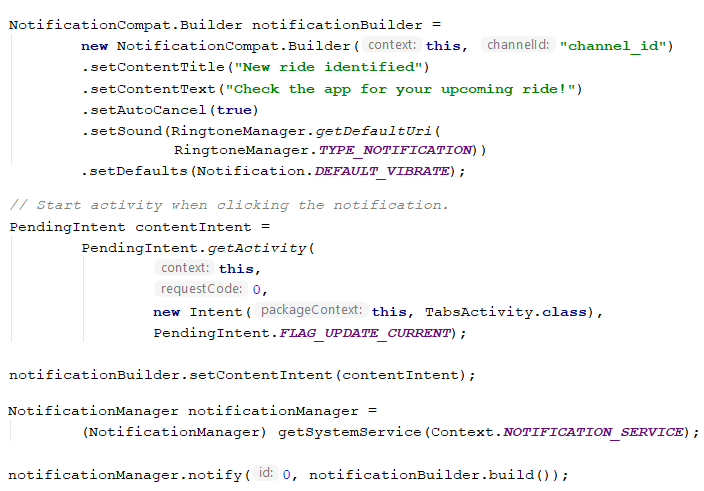
 Recepționarea mesajelor de la server prin care un utilizator este anunțat că a fost sau nu repartizat la un carpool se face prin intermediul unui serviciu. Serverul aplicației face un apel HTTP la serverul Firebase Cloud Messaging, care direcționează mai departe apelul către dispozitivul dat ca destinație. Serviciul aplicației, care extinde FirebaseMessagingService, preia acest mesaj prin metoda *onMessageReceived(),* salveazând local informațiile primite pentru a le pune la dispoziția utilizatorului și creând o notificare.

*Fig. 4.3.11.1 - Metoda onMessageReceived preia mesajul transmis de FCM în momentul în care serverul aplicației face apelul către FCM; în cazul în care utilizatorul este autentificat, se salvează local datele primite (prin metoda addRideToSessionManage) și se creează o notificare (prin metoda sendNotification)*

**

*Fig. 4.3.11.2. – Notificarea creată la sosirea unui mesaj de la FCM*

În figura .. se observă modul în care se construiește o notificare. Sunt setate titlul („New ride identified“ - „O nouă cursă identificată“), textul („Check the app for your upcoming ride!“ - „Verifică aplicația pentru următoarea cursă“), dar și felul în care utilizatorul este înștiintat de sosirea unei notificări (prin sunet și vibrații). Pe lângă acestea, se setează activitatea (TabsActivity) care se va deschide în momentul apăsării notificării, prin intermediul unui PendingIntent. În final, se preia o instanță a administratorului de notificări (NotificationManager), prin intermediul căruia se transmite notificarea pentru a fi afișată pe ecranul utilizatorului.



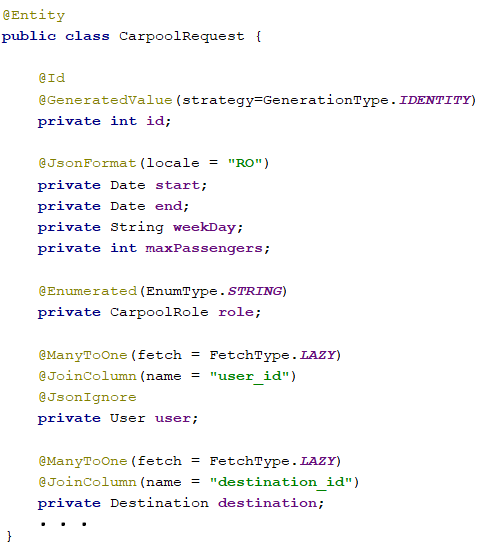
*Fig. 4.3.11.3. – Contruirea unei notificări ce deschide aplicația atunci când se execută click pe ea și publicarea ei de către o instanță NotificationManager.*

## **Serverul aplicației**

### **Definirea modelelor**

Realizarea persistenței între obiecte și baza de date se realizează prin definirea unei clase conținând adnotarea @Entity pentru fiecare tabel al bazei de date, ce conține câte un câmp corespunzător fiecărei coloane. Astfel, aceste clase sunt următoarele:

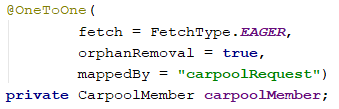
* User (id, nume, număr de telefon, parolă, adresă, mașină)
* CarpoolRequest (id, id-ul utilizatorului, rolul, data plecării, data întoarcerii, id-ul destinației)
* Destination (id, descrierea destinației, adresa)
* Distance (id, id-ul primului utilizator, id-ul celui de-al doilea utilizator, distanța, timpul mediu)
* DistanceUserDestination (id, id-ul utilizatorului, id-ul destinației, distanța, timpul mediu, direcția deplasării – spre utilizator sau spre destinație)
* Carpool (id, numărul membrilor, numărul maxim de pasageri)
* CarpoolMember (id, id-ul cererii, id-ul carpool-ului, numărul de ordine în carpool la plecare, ora plecării)
* DeviceToken (id, id-ul utilizatorului, token-ul asociat dispozitivului utilizatorului)



*Fig. 4.4.1.1. – definirea clasei model CarpoolRequest; aceasta conține câmpuri corespunzătoare coloanelor tabelului, precum și adnotări pentru stabilirea relațiilor dintre tabele*

În figura 4.4.1.1. se observă adnotarea @ManyToOne asociată unor referințe către modelele corespunzătoare altor tabele. Acest lucru marchează existența unei chei străine în tabelul căreia îi este asociat modelul nostru, reprezentată de identificatorul unui alt tabel (cheia sa primară). Adnotarea @JoinColumn arată coloana care va conține cheia străină.

La rândul său, id-ul tabelului căruia îi corespunde modelul CarpoolRequest (cheia primară) este cheie străină în tabelul CarpoolMember. Acest lucru va fi marcat în cod prin adnotarea @OneToOne (mappedBy = “carpoolRequest”), unde mappedBy arată câmpul din clasa CarpoolMember care conține referința către instanța clasei CarpoolRequest. Setarea „orphanRemoval = true“ are rolul a face posibilă ștergerea din tabelul carpool\_member a intrărilor care conțin ca și chei străine intrări din carpool\_request ce nu mai există.

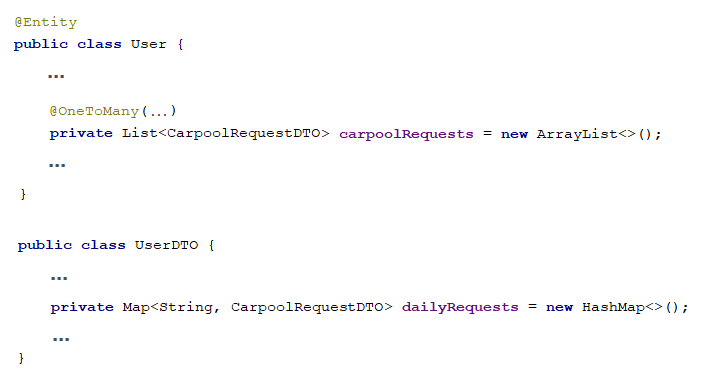


*Fig. 4.4.1.2. – Marcarea în entitatea CarpoolRequest a faptului că o instanță a altei clase (CarpoolMember) conține o referință către o instanță a CarpoolRequest (tabelul carpool\_member conține o cheie străină reprezentată de cheia primară a tabelului carpool\_request)*

### **Definirea claselor pentru transmiterea datelor în rețea (DTO/data transfer object)**

Scopul claselor definite anterior este de a modela tabelele bazei de date. Instanțele acestora nu vor fi transmise în rețea asa cum au fost extrase din baza de date, ci vor fi transformate în alte obiecte, care încapsulează sub o formă mai potrivită datele relevante pentru client.

Un exemplu în acest sens este prezent în figura .. . Modelul User conține o listă cu toate cererile făcute vreodată de un utilizator. Cu toate acestea, aplicația client va avea nevoie, la un moment dat, doar de cinci cereri, cele corespunzătoare fie săptămânii în curs (dacă este între luni și vineri), respctiv săptămânii următoare (dacă este sâmbătă sau duminică). Acestea vor fi trimise sub forma unei hărți, unde fiecare cerere este asociată unui șir de caractere (String) reprezentat de ziua săptămânii pentru care s-a făcut cererea. Astfel, în momentul interogării bazei de date se obțin obiecte de tip User, dar acestea sunt transformate în obiecte de tip UserDTO înaintea transferului în rețea.



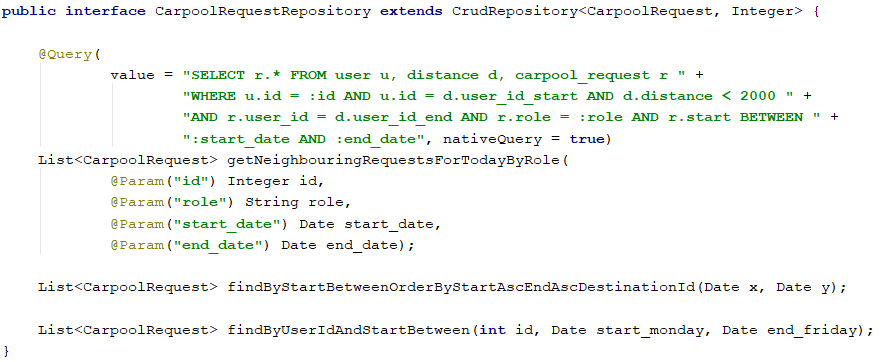
*Fig. 4.4.2.1. – clasa User mapează tabelul din baza de date, dar nu va avea instanțe ce sunt transmise în rețea; câmpul carpoolRequests va fi împachetat sub o formă mai potrivită pentru client înainte de a fi transmis sub forma câmpului dailyRequests din UserDTO*

### **Definirea repository-urilor**

Pentru realizarea cu ușurință a operațiilor specifice bazei de date (CRUD – Create, Read, Update, Delete), Spring definește o interfață, CrudRepository, ce trebuie extinsă și particularizată fiecărui model descris anterior.

Interfața vine cu o serie de metode predefinite, ce pot fi traduse în limbajul SQL (de exemplu, findAll() aplicată repository-ului corespunzător entității CarpoolRequest se traduce în SELECT \* FROM carpool\_request).

Pe lângă acestea, interfața permite definirea semnăturii unor metode noi, cu anumite cuvinte cheie, ce pot sunt traduse automat în limbajul SQL (de exemplu, findByUserIdAndStartBetween(id, x, y) din figura .. se traduce în SELECT \* FROM carpool\_request WHERE user\_id = id AND start IS BETWEEN x AND Y).

Dacă metodele noi sunt complicate și nu pot fi definite prin cuvintele cheie menționate anterior, interfața permite definirea de metode ce conțin adnotarea @Query, urmată de instrucțiunea SQL ce dorim să se execute. În figura , se observă această adnotare asupra metodei getNeighbouringRequestsForTodayByRole().

*Fig. 4.4.3.1. – definirea repository-ului corespunzător entiății CarpoolRequest și definirea unor noi metode, fie printr-o instrucțiune SQL (getNeighbouringRequestsForTodayByRole), fie prin utilizarea unor cuvinte cheie, traduse automat în limbajul SQL (findByStartBetweenOrderByStartAscEndAscDestinationId / findByUserIdAndStartBetween)*

### **Definirea serviciilor**

Serviciile au rolul de a prelua informații de la baza de date (în acest caz, de la repository) și de a lucra cu acestea în scopul producerii unor rezultate de interes pentru utilizator.

 În acest scop, fiecare entitate prezentată anterior va avea nevoie de un serviciu pentru a putea fi accesată ulterior preluării din baza de date de către repository. Clasa corespunzătoare va avea adnotarea @Service și va avea injectată (prin adnotarea @Autowired) o referință către repository-ul corespunzător, de la care va prelua date pe care le va prelucra.

*Fig. 4.4.4.1. – Definirea serviciului pentru CarpoolRequest cu adnotarea @Service și injectarea repository-ului CarpoolRequestRepository, de la care se preiau date*

Serviciile nu trebuie să fie asociate unei entități. Un exemplu de astfel de serviciul este cel care realizează împărțirea utilizatorilor în carpool-uri, definit în capitolul 4.3.6.

### **Definirea controller-elor**

În framework-ul Spring, controller-ele definesc metode ce sunt apelate la sosirea unor apeluri HTTP la adresa la care rulează serverul. Definirea unui controler se face prin adnotarea @RestController înaintea clasei respective.

Pentru a mapa metodele la o anumită adresă se folosește adnotarea @RequestMapping, specificându-se, de asemenea, tipul apelului HTTP (GET – implicit, PUT, POST, DELETE). Cererile sunt preluate de controler, dar acțiunile propriu-zise sunt făcute de servicii, pe care controller-ele de au injectate și ale căror metode le apelează.

În figura .. se observă două adnotări: @RequestBody, care arată faptul că apelul este însoțit de un JSON ce poate fi deserializat într-un obiect de tip CarpoolRequest, și @PathVariable, care arată faptul că valoarea variabilei respective este preluată din URL-ul la care s-a făcut apelul.



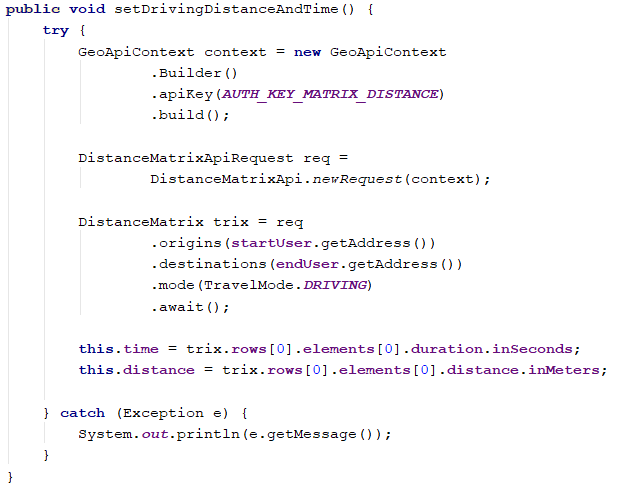
*Fig. 4.4.5.1. – Definirea unui Controller ce metode apelate la sosirea unor apeluri HTTP la adresa la care rulează serverul*

### **Stabilirea distanțelor între puncte diferite prin Distance Matrix API**

În momentul adăugării unui utilizator nou sau al schimbării adresei acestuia, este necesară actualizarea bazei de date cu informații legate de distanțele între noul utilizator și utilizatorii deja existenți, precum și dintre el și punctele finale existente.

Pentru acest lucru s-a folosit un Distance Matrix API. În primul rând, s-a creat un GeoApiContext, adică un punct de intrare pentru cererile către API-urile Google Geo. Cererile către acestea vor primi răspuns doar dacă se atașează contextului o cheie validă.

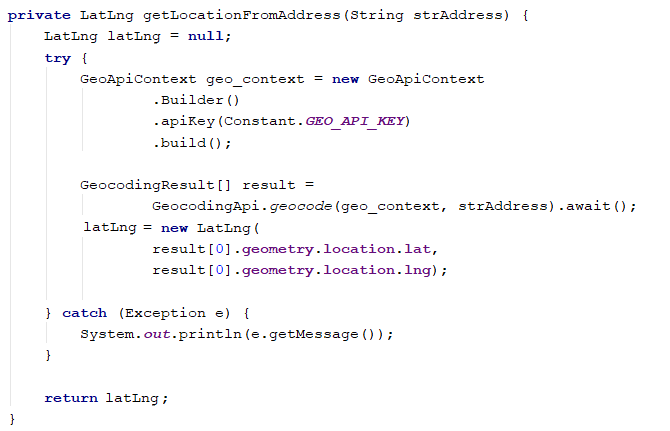
Printr-o cerere DistanceMatrixApiRequest ce conține adresa de start, adresa de final și tipul de deplasare (în cazul nostru, cu mașina – DRIVING), se obține ca rezultat o matrice din care se pot extrage distanța, precum și timpul deplasării. În cazul a două puncte A si B, s-au calculat atât valorile pentru (A – punct de start, B – punct de final), cât și pentru (B – punct de start, A – punct de final).



*Fig. 4.4.6.1. – crearea unei cereri către Distance Matrix API ce conține două adrese se obțin distanța dintre ele, precum și timpul necesar deplasării*

### **Transformarea adreselor în coordonate prin Geocoding API**

Deși baza de date reține adresele sub forma unor șiruri de caractere (obiecte String), dispunerea pe hartă a acestora, în clientul Android, se face prin intermediul coordonatelor adreselor. Astfel, în momentul în care se creează un carpool și se dorește transmiterea sa către utilizatori, se foloseste Geocoding API pentru obținerea latitudinii, respectiv a longitudinii adreselor respective, pentru a putea fi puse ulterior pe hartă.

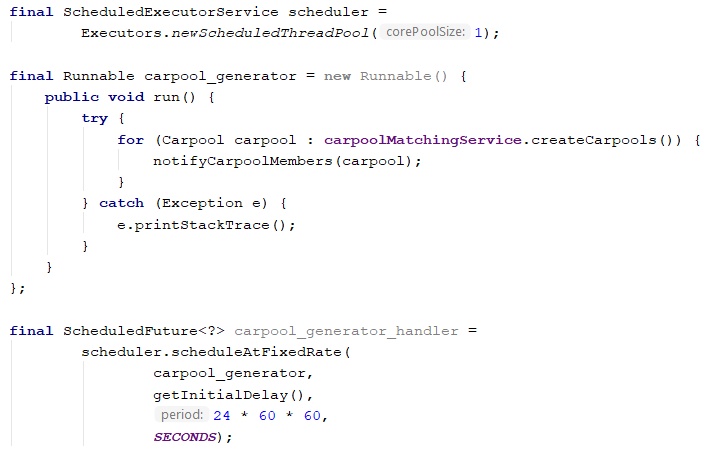
În primul rând, s-a creat un GeoApiContext, un punct de intrare pentru cererile către API-urile Google Geo. Cererile către acestea vor primi răspuns doar dacă se atașează contextului o cheie validă (GEO\_API\_KEY). Cererea va conține, pe lângă context, adresa care ne interesează). Rezultatul primit este sub forma unei matrici GeocodingResult, din care se pot extrage latitudinea și longitudinea.

*Fig. 4.4.7.1. – crearea unei cereri către Geocoding API, ce conține o adresă sub forma de String; din răspunsul primit se extrag latitudinea, respectiv longitudinea corezpunzătoare adresei*

### **Crearea carpool-urilor**

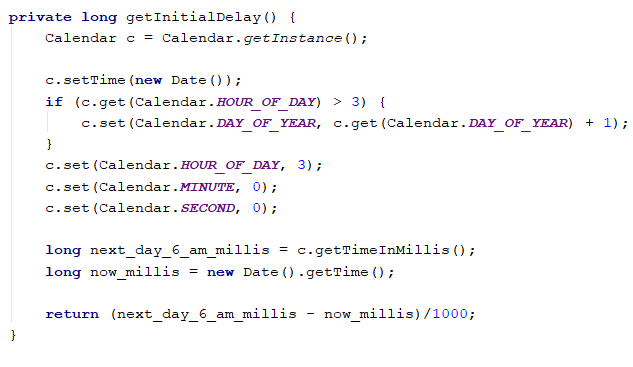
Crearea carpool-urilor se face pe baza unui algoritm programat să ruleze în fiecare zi, la aceeași oră (3 noaptea), pentru a ne asigura că acestea sunt generate după ce utilizatorii nu mai au posibilitatea de a modifica cererea, precum și înaintea orei la care ar putea începe un carpool.

Pentru rularea începând cu o oră exactă din viitor s-a folosit, dupa cum reiese din figura 4.4.8.1., un ScheduledExecutorService cu un fir de execuție. În continuare, are loc definirea unui fir, *carpool\_generator,* a cărui metodă run() arată ce se va executa în cadrul său: metoda de generare a carpool-urilor, prezentă într-un serviciu, precum și apelul metodei de notificare a membrilor carpool-ului. Urmează setarea scheduler-ului să funcționeze periodic, prin intermediul funcției scheduleAtFixedRate, ce ia ca parametri firul de execuție ce se va executa (carpool\_generator), întârzierea initială, cu rolul de programare e primei execuții (dată de apelul funcției getInitialDelay()), perioada în secunde (24 de ore înmultite cu 60 de minute înmulțite cu 60 de secunde), precum și unitatea de măsură a timpului folosită la întârziere și perioadă (secunde).



*Fig. 4.4.8.1. – Instanțierea unui ScheduledExecutorService cu un fir de execuție ce rulează în fiecare zi, la ora 3 noaptea.*

Funcția Funcția getInitialDelay() stabilește întârzierea către prima oră 3 noaptea începând cu prima rulare a programului. Astfel, dacă este trecut de ora 3 când se începe rularea, se va considera următoarea zi, iar, în caz contrar, ziua curentă. Întârzierea necesară este dată de diferența dintre prima ora 3 noaptea următoare și ora de început a rulării. Diferența fiind în milisecunde, rezultatul va fi împărțit la 1000 (calculele în scheduler se fac în secunde).



*Fig. 4.4.8.2 – Setarea întârzierii prin calcul diferenței dintre prima oră de 3 noaptea din viitor și ora curentă.*

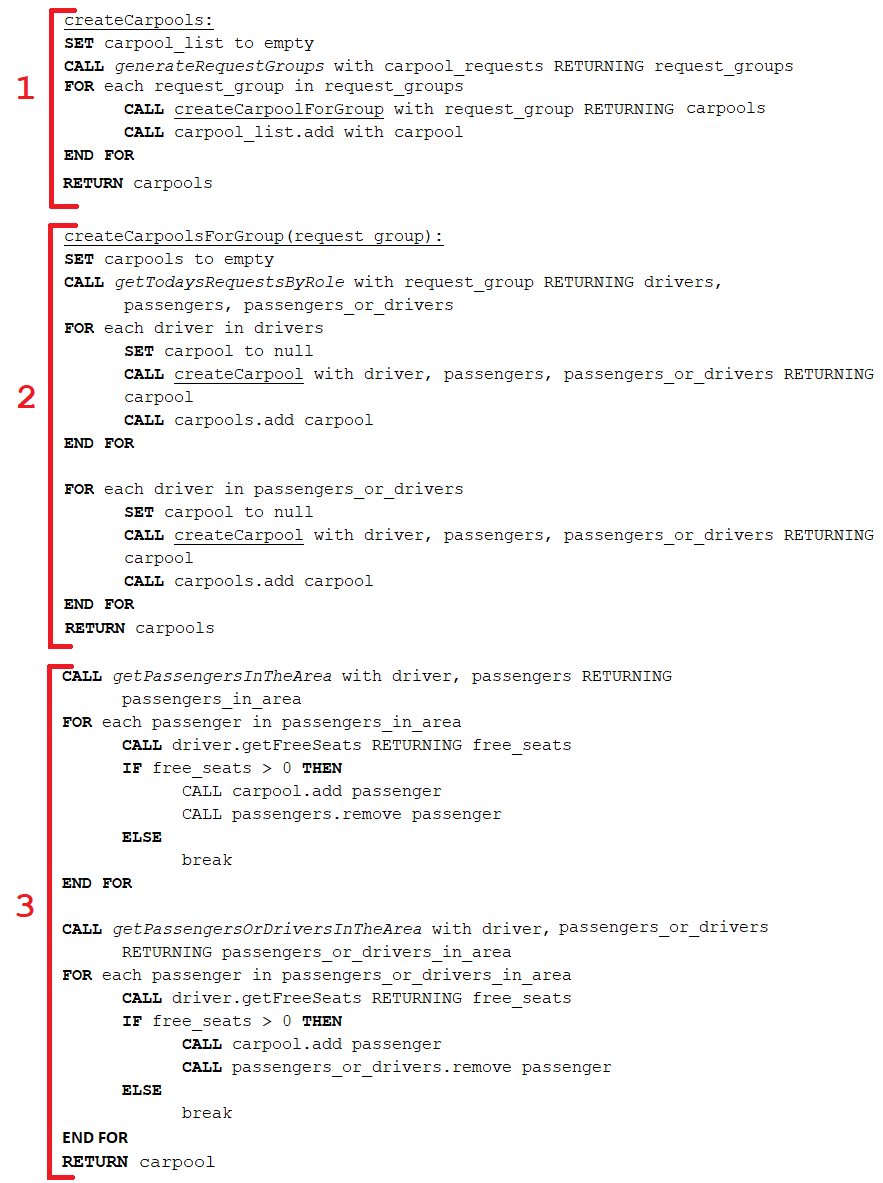
Algortimul de împărțire a utilizatorilor cu cereri de carpool pentru ziua respectivă este descris în următorii pași:

1. Se creează grupuri de cereri pentru care corespund orele de plecare și de întoarcere, dar și destinația
2. Se împart aceste grupuri în funcție de rolul utilizatorilor: pasageri (grupul 1), șoferi (grupul 2) și pasageri sau șoferi (grupul 3). Pentru fiecare membru al grupului 2 se creează un carpool (pasul 3), la fel și pentru membri grupului 3 rămași cu cereri nesoluționațe la creările anterioare, cu diferența că grupul 3 va fi prima dată sortat în funcție de numărul de kilometri parcurși în calitate de șofer de către utilizatorii corespunzători.
3. Se generează lista cu membri ai grupului 1, respectiv ai grupului 3 din vecinătate (prin vecinătate se înțelege o rază de 2 kilometri în jurul șoferului): se adaugă prima dată membri ai listei 1, apoi ai listei 2, până când acel carpool este plin (adăugarea se face în ordine aleatoare, pentru a se asigura împărțirea echitabilă de-a lungul timpului); pe măsura adăugării, se marchează cererile respective ca fiind soluționate; se repetă acest pas până în momentul în care s-au creat carpool-uri pentru toți șoferii

Pe parcursul creării carpool-urilor se calculează și timpii de plecare ai fiecărui membru al unui carpool astfel încât să se asigure, la drumul de dus, sosirea la destinația cu 15 minute înaintea orei stabilite.

Ora de întoarcere va fi la 15 minute după terminarea programului utilizatorilor, pentru a ne asigura că fiecare pasager ajunge la mașină în timp util.

Calculul cererilor venite de la utilizatorii din vecinătate se realizează prin stabilirea unui perimetru acceptat ca fiind „vecinătate“, stabilit a fi la 2km.

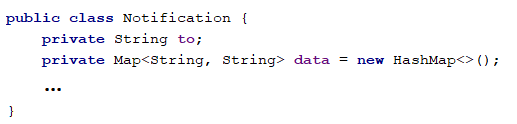


*Fig. 4.4.8.3. – pseudocodul corespunzător creării carpool-urilor pe baza cererilor: pasul 1 – crearea grupurilor în funcție de ore și destinații; 2 – stabilirea posibililor șoferi; 3 – crearea carpool-urilor pentru acei șoferi pe baza cererilor din vecinătate*

### **Trimiterea notificărilor prin Firebase Cloud Messaging**

Odată generate carpool-urile, utilizatorii vor fi înștiințați de de acest lucru prin intermediul unor notificări. Acestea sunt generate de aplicația client la primirea unui mesaj de la FCM ca răspuns al unui apel efectuat de server.

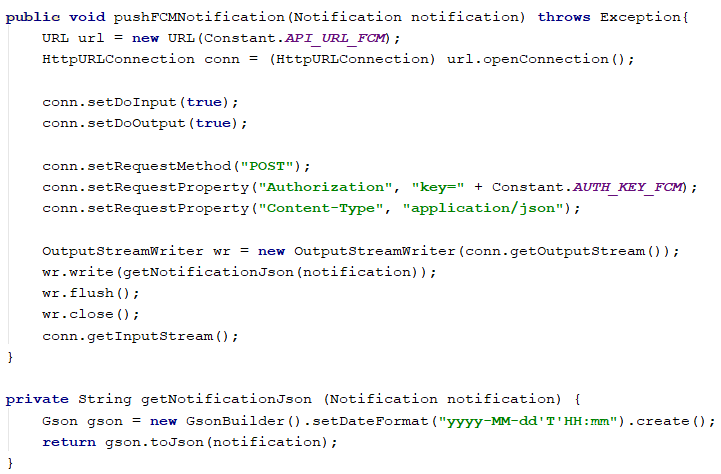
S-a creat o clasă, Notification, ce va încapsula o cerere către FCM. Aceasta conține un câmp *to*, căruia îi este atașat token-ul corespunzător aplicației unui utilizator, precum și un câmp *data*, căruia îi sunt atașate informațiile ce se doresc a fi transmise.



*Fig. – clasa Notification conține informațiile cuprinse cererii către FCM*

Pentru transmiterea mesajului, se va realiza o conexiune HTTP către adresa FCM (<https://fcm.googleapis.com/fcm/send>). Se stabilesc tipul apelului (POST), forma de serializare a mesajului (JSON), precum și autorizarea, prin intermediul unei chei valide pentru acest API (AUTH\_KEY\_FCM).

Notificarea va conține, în câmpul destinatarului (*to*) token-ul dispozitivului utilizatorului către care se dorește transmiterea mesajului, și, în câmpul de date, informații referitoare la carpool-ul în cauză. Obiectul de tip Notification este transformat în JSON prin intermediul bibliotecii GSON.

**

*Fig. 4.4.9.1 – Transmiterea unei notificări prin intermediul API-ului FCM.*

# **UTILIZAREA**

*Fig. - căsuțele de dialog (DialogBox) primite de utilizator în momentul adăugării/modificării unei cereri; se aleg rolul, numărul de pasageri (nu și în cazul în care s-a selectat pasager în pasul anterior), ora de plecare și ora de întoarcere*

# **CONCLUZII**

## **Concluzii**

Îndatoririle zilnice, fie ca vorbim de educație sau serviciu, ne obligă să ne deplasăm fizic către anumite puncte mai apropiate sau mai îndepărtate de locuință. Dacă distanțele sunt suficient de mari încât deplasarea pe jos să nu fie posibilă, iar transportul în comun să nu fie ales din diferite motive, vom recurge la o ultimă soluție: vom lua mașina. Această soluție, deși convenabilă la prima vedere, este din ce în ce mai problematică din cauza traficului tot mai intens și a lipsei locurilor de parcare.

Ținând cont de faptul că o mare parte din mașinile aflate în trafic au locuri libere și că există anumite puncte „de foc“ în orașul Timișoara, către care se deplasează zilnic o mulțime de persoane, aplicația de ridesharing eficient vine ca o soluție naturală la încercarea rezolvării acestei probleme. Astfel, aplicația își propune realizarea de carpool-uri compuse din oameni care au același program într-o anumită zi, se îndreaptă către aceeași destinația și pornesc din aceeași zonă.

Obiectivele sunt atinse prin reducerea numărului de mașini participante la trafic care are, ca urmări imediat, reducerea timpului petrecut în trafic și facilizarea găsirii locurilor de parcare, și, ca urmări pe termen lung, reducerea poluării, crearea de legături interpersonale între cei care împart un carpool.

## **Viitoare îmbunătățiri**

În viitor, aplicația poate fi dezvoltată pentru a putea oferi diverse noi facilități. Una dintre acestea ar fi adăugarea de noi destinații (în prezent sunt în număr de 5), precum și a unor noi orașe pentru care să se ofere posibilitatea de ridesharing. O altă îmbunătățire ce poate fi adusă este reprezentată de posibilitatea stabilirii unor cereri care se repetă săptămânal (în acest momentul, cererile pot fi făcute doar pentru săptamâna în curs).

# **BIBLIOGRAFIE**

|  |  |
| --- | --- |
| [1] | "https://www.thealternativedaily.com/why-ridesharing-matters/," [Online]. |
| [2] | "https://www.eea.europa.eu/data-and-maps/indicators/occupancy-rates-of-passenger-vehicles/occupancy-rates-of-passenger-vehicles," [Online]. |
| [3] | "https://en.wikipedia.org/wiki/Android\_(operating\_system)," [Online]. |
| [4] | "https://developer.android.com/guide/platform," [Online]. |
| [5] | "https://www.tutorialspoint.com/android/android\_application\_components.htm," [Online]. |
| [6] | "https://developer.android.com/guide/components/activities/activity-lifecycle.html," [Online]. |
| [7] | "https://www.tutorialspoint.com/jpa/jpa\_orm\_components.htm," [Online]. |
| [8] | "https://en.wikipedia.org/wiki/Spring\_Framework," [Online]. |
| [9] | "https://www.tutorialspoint.com/spring\_boot/spring\_boot\_introduction.htm," [Online]. |