

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și tehnologia informației
SPECIALIZAREA: Tehnologia informației

Aplicație android pentru serviciul de transport public

LUCRARE DE LICENȚĂ

Coordonator științific:
Ș.l.dr.ing. Paul Herghelegiu

Absolvent:
Condriea Ștefan-Cătălin

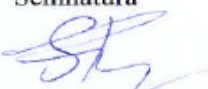
Iași, 2020

**DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
PROIECTULUI DE DIPLOMĂ**

Subsemnatul CONDRIEA ȘTEFAN-CĂTĂLIN,
legitimat cu CI seria VS nr. 775651, CNP 1971215375200
autorul lucrării APLICAȚIE ANDROID PENTRU SERVICIUL
DE TRANSPORT PUBLIC

elaborată în vederea susținerii examenului de finalizare a studiilor de licență, programul de studii TEHNOLOGIA INFORMAȚIEI organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea IULIE a anului universitar 2019-2020, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTL.POM.02 - Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
19. 07. 2020

Semnătura


Cuprins

Introducere.....	1
Capitolul 1. Fundamentarea teoretică și documentarea bibliografică pentru tema propusă.....	3
1.1. Referințe la teme/subiecte similare.....	3
1.2. Tehnologii folosite pentru implementare.....	4
1.2.1. Sistemului de operare Android și o scurtă istorie a acestuia.....	4
1.2.1.1. Arhitectura sistemului de operare Android.....	4
1.2.2. Android Studio.....	6
1.2.2.1. Componentele unei aplicații Android.....	6
1.2.2.2. Ciclul de viață a unei activități.....	7
1.2.2.3. Structura de directoare și fișiere în Android Studio.....	9
1.2.3. Firebase.....	10
1.2.4. Limbajul Java.....	11
1.2.5. JavaScript.....	11
1.2.6. XML.....	12
Capitolul 2. Proiectarea aplicației.....	13
2.1. Aplicația client Android.....	13
2.1.1. Arhitectura aplicației client Android.....	14
2.2. Diagrame UML.....	14
2.2.1. Diagramele use-case.....	15
Capitolul 3. Implementarea aplicației.....	18
3.1. Crearea și configurarea unui proiect în Android Studio și conectarea acestuia la serverul Firebase.....	18
3.2. Logarea și înregistrarea.....	18
3.3. Setarea API-ului Google Maps.....	22
3.4. Actualizarea periodică a locațiilor utilizatorilor.....	22
3.5. Salvarea și actualizarea coordonatelor geografice ale locațiilor în baza de date Firebase.....	25
3.6. Plasarea unei cereri de transport și găsirea unui șofer disponibil.....	26
3.7. Salvarea și extragerea informațiilor din baza de date Firebase.....	30
3.8. Funcția de Place AutoComplete.....	32
3.9. Crearea rutelor și afișarea acestora pe interfața aplicației.....	34
3.10. Crearea istoricului curselor efectuate și popularea acestuia.....	36
3.11. Implementarea sistemului de plată PayPal.....	40
3.11.1. Setarea contului Paypal.....	40
3.11.2. Implementarea sistemului de plată PayPal în aplicație.....	40
3.12. Implementarea cereri http al șoferului către PayPal.....	43
3.12.1. Implementarea PayPal – setarea și configurarea host-ului și a funcțiilor.....	43
3.12.2. Implementarea cererii http în JavaScript.....	44
3.12.3. Apelul în aplicație a cererii http pentru realizarea plății șoferului.....	48
Capitolul 4. Testarea aplicației și rezultate experimentale.....	51
4.1. Testarea aplicației Android.....	51
4.2. Rezultate experimentale.....	54
Concluzii.....	56
Bibliografie.....	57

Anexe.....	58
Anexa 1. Crearea și configurarea proiectului în Android Studio și conectarea la serverul Firebase.....	58
Anexa 2. Activarea metodei de înregistrare a utilizatorilor din consola Firebase.....	61
Anexa 3. Crearea unei chei API și importarea acesteia în proiect.....	61
Anexa 4. Codul cererii http pentru realizarea plății șoferului.....	61

Aplicație android pentru serviciul de transport public

Condriea Ștefan-Cătălin

Rezumat

Proiectul reprezintă o aplicație Android menită să servească serviciului de transport persoane. Aplicația vine în ajutorul atât celor ce doresc să ajungă dintr-un loc în altul, cât și celor ce sunt dispuși să ofere servicii de transport contra cost. Astfel aplicația a fost proiectată în așa fel încât să aibă două interfețe separate, aferente celor două tipuri de utilizatori, pasageri respectiv șoferi.

Aplicația poate fi utilizată pe dispozitive ce rulează sistemul de operare Android, care au versiunea cel puțin 4.1. Datorită acestui fapt, aplicația are o acoperire a dispozitivelor ce folosesc acest sistem de operare de aproximativ 99,8%. Pentru ca aplicația să funcționeze, aceasta are nevoie în permanență de accesul la internet, cât și de serviciul GPS care trebuie să fie activat.

După instalarea aplicației Android, utilizatorii vor fi nevoiți să își creeze un cont personal, prin intermediul căruia se vor autentifica în secțiunea aferentă tipului de utilizator, șofer respectiv pasager. Odată ce logarea a avut succes și utilizatorii au accesul la interfața propriu-zisă a aplicației unde pot accesa toate funcționalitățile aplicației, aceștia vor fi nevoiți să își seteze mai întâi profilul cu niște date personale, date ce vor fi folosite în momentul în care are loc serviciul de transport, făcându-se schimbul de date între șofer și pasager. Datorită serviciilor Google Maps, utilizatorii vor fi identificați pe hartă și afișați, actualizarea poziției lor făcându-se în timp real. O cerere de transport este inițiată în momentul în care un utilizator de tip pasager selectează tipul de transport dorit și opțional, selectarea unei destinații, după care va apăsa un buton ce va inițializa căutarea unui șofer ce respectă criteriile selectate, moment în care va avea loc căutarea șoferilor disponibili pe o anumită rază, iar când acestei cereri i se va asocia un șofer, pe interfața șoferului va avea loc desenarea unor rute pe harta afișată pe dispozitiv, între el și pasager, cu ajutorul cărora șoferul va primi o ghidare în trafic.

De asemenea, o altă funcționalitate importantă a aplicației este reprezentată de sistemul de plată, prin care pasagerii vor putea plăti serviciile de transport. Ca și sistem de plată a fost ales PayPal, așadar pasagerii vor fi nevoiți să își creeze un cont de PayPal prin intermediul căruia se vor loga în momentul în care vor efectua plata. Plățile efectuându-se către contul firmei, ca mai apoi șoferii prin intermediul unor cereri de plată către firmă, să primească suma aferentă transporturilor efectuate.

Aplicația mai dispune și de o interfață prin care le sunt afișate utilizatorilor istoricul curselor efectuate. Istoricul unei curse conținând data și ora când a fost efectuată cursa respectivă, cât și informațiile privind șoferul și pasagerul ce a efectuat respectiva cursă. De asemenea, aici pasagerii vor putea oferi un rating cursei și totodată de aici se poate efectua plata serviciului de transport.

Introducere

Știm cu toții că trăim în era tehnologiei, o eră care a explodat în ultimele decenii și care a revoluționat toate domeniile de activitate, având drept scop creșterea eficienței și reducerea semnificativă a timpului și costului pentru realizarea anumitor produse sau activități, crearea de servicii etc. Dacă la începutul anilor '90, ideea ca o companie sau firmă să își mute activitatea în mediul online nici nu se punea în discuție, acum mediul online reprezintă startup-ul¹ fiecărei afaceri, fiind principalul mediu prin care firmele își expun serviciile și prin care își efectuează respectivele servicii.

Dacă mutarea activității unei firme în mediul online era greu de realizat în acei ani, implementarea acestora pe dispozitivele mobile era aproape imposibilă, chiar și mult mai recent, la sfârșitul anilor 2000, acest lucru datorându-se nu numai dispozitivelor mult mai slabe ca și performanță și a tehnologiilor ce le dețineau, ci și numărului redus de dispozitive ce populația le dețineau, făcând orice afacere ineficientă din punctul acesta de vedere.

Însă acele vremuri s-au schimbat. În ultimul deceniu domeniul dispozitivelor mobile a fost revoluționat total oferind dispozitive mobile performante la prețuri reduse, performanța acestora fiind comparabilă cu cea a unui calculator. Din acest motiv, în zilele noastre oamenii sunt nelipsiți de un astfel de dispozitiv încă de la cele mai fragede vârste, acesta reprezentând în momentul de față, cel mai folosit dispozitiv în întreaga lume. Astfel dispozitivul mobil îndeplinind cel mai important criteriu pentru o afacere și anume, piața de desfacere.

Din acest motiv am ales implementarea unei aplicații Android, acest sistem de operare fiind unul din principalele sisteme de operare pentru dispozitivele mobile, alături de iOS și Windows Mobile.

Aplicația Android pentru serviciul de transport public este destinată atât cetățenilor ce doresc a se deplasa dintr-un loc în altul prin plasarea unei cereri de transport, prin intermediul interfeței aplicației, cât și a celor ce sunt dispuși să ofere servicii de transport contra cost, prin punerea la dispoziție a propriei mașini și a timpului personal.

Aplicația Android poate fi instalată pe dispozitive ce rulează sistemul de operare Android, versiunea minima necesară fiind 4.1. A fost ales acest sistem de operare din simplu fapt că este cel mai răspândit în rândul dispozitivelor mobile, cât și pentru faptul că dispune de o platformă open-source de dezvoltare al aplicațiilor, prin intermediul programului Android Studio oferit de către cei de la Google.

Pentru dezvoltarea proiectului au fost necesare cunoștințe în limbajul de programare Java, acesta reprezentând unul din limbajele pe care le folosește programul Android Studio pentru realizarea back-end-ului aplicațiilor. De asemenea pentru partea de back-end s-a folosit și JavaScript pentru implementarea serviciului de plată ce îl deține aplicația. Pentru realizarea front-end-ului au fost folosite elemente de tip XML.

Lucrarea de față este împărțită în patru capitole, capitole ce vor afișa în amănunt informațiile necesare implementării principalelor funcționalități ale acestui proiect, cât și descrierea tehnologiilor și a conceptelor folosite pentru realizarea proiectului.

Așadar în primul capitol reprezentat de „Fundamentarea teoretică și documentarea bibliografică pentru tema propusă”, are loc introducerea în tehnologiile și conceptele folosite pentru implementarea proiectului, unde se va pune accentul mai mult pe arhitectura și modul de execuție al unei aplicații, fiind două lucruri esențiale pe care orice dezvoltator de aplicații trebuie

¹ Startup-ul este un tip de organizație temporară, care are scopul de a căuta un model de business repetabil și scalabil, ca răspuns la o problemă existentă nerezolvată, și care constituie etapa de trecere de la o simplă idee la o afacere.

sa le stapânească în momentul în care dorește a dezvolta o aplicație Android.

În al doilea capitol, numit „Proiectarea aplicației”, se va descrie modul în care aplicația a fost gândită pentru a fi implementată și folosită de către utilizatori. Aceste lucruri vor fi puse în evidență prin intermediul unor diagrame UML, necesare pentru procesul de proiectare a aplicației.

În al treilea capitol, numit “Implementarea aplicației”, va avea loc descrierea amănunțită a principalelor funcționalități ale aplicației cât și implementarea acestora.

Ultimul capitol, „Testarea aplicației și rezultate experimentale”, conține metodele prin care aplicația a fost testată și modalitățile de detectare a erorilor. Tot aici fiind prezentate performanțele aplicației, privind utilizarea resurselor hardware.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică pentru tema propusă

Tema acestui proiect este reprezentată de crearea unei aplicații Android destinată serviciului de transport public, astfel încât utilizatorii să beneficieze de o experiență simplă și plăcută în momentul utilizării acesteia și care să conțină funcționalitățile necesare acestui tip de aplicație.

1.1. Referințe la teme/subiecte similare

Datorită faptului ca acest concept de aplicație exista deja pe piață, s-a urmărit doar implementarea unei aplicații cât mai simple posibil, cu o interfață intuitivă, care să conțină minimul de funcționalități prin care o persoană poate să efectueze o cursă de transport sau să ofere servicii de transport.

Unele dintre aplicațiile deja existente pe piața similare cu tema aleasă fiind:

- Uber – este una din cele mai populare aplicații de smartphone care conectează pasagerii cu șoferii unor mașini dispuși să ofere contra cost servicii de transport. Aplicația este disponibilă gratuit, atât pentru Android, cât și pentru iOS, nu există dispecerat sau aparate de taxat, totul realizându-se online. Uber este o aplicație ce permite oricărui șofer care respectă anumite cerințe să transporte persoane și să fie plătit online, fiind o nouă alternativă la serviciile de taxi. Pasagerii pot vedea poza și numele șoferului, numărul de înmatriculare al mașinii și rating-ul șoferului înainte ca mașina să ajungă. Șoferii înregistrați pe platformă sunt plătiți săptămânal, prin transfer bancar.
- BlaBlaCar – este o comunitate și o platformă de ridesharing², cu un serviciu profesionist de transport. Interconectează pasagerii care caută o cursă cu acei conducători auto care au locuri libere în mașină și doresc să își împartă costurile călătoriei. Scopul acestei aplicații nu este cel de a produce profit celui ce transportă persoane, ci acela de a reduce din costurile totale ale respectivei călătorii. Are un sistem de rating implementat, astfel îndemnând șoferii să ofere servicii bune pentru a fi în continuare recomandați de către aplicație. De asemenea, aplicația dispune și de un messenger, astfel șoferii pot comunica foarte ușor cu alți membri din comunitate.
- Pony – reprezintă o aplicație ce se bazează pe conceptul de car-sharing³, prin care utilizatorii își pot închiria mașini din flota de mașini pe care firma le deține. Respectiv mașini sunt parcate pe strada, acestea fiind afișate în timp real pe harta aplicației. După terminarea sesiunii de închiriere, utilizatorul poate lăsa mașina în orice loc din zona operațională Pony, iar alimentarea mașinii cu combustibil este opțională, de acest lucru ocupându-se personalul firmei.

2 Ridesharing – reprezintă un concept ce presupune împărțirea costurilor unei călătorii cu o persoană sau un grup de persoane.

3 Car-sharing – este un serviciu de închiriere a mașinilor pe perioade scurte de timp.

1.2. Tehnologii folosite pentru implementare

1.2.1. Sistemului de operare Android și o scurtă istorie a acestuia

Android este o platformă software și un sistem de operare pentru dispozitive și telefoane mobile, bazat pe nucleul Linux, versiunea 2.6, dezvoltat de consorțiul comercial Open Handset Alliance, un grup de companii hardware, software și telecomunicații, dedicate pentru a inova și a ridica standardele dispozitivelor mobile, astfel oferind consumatorilor o experiență mai plăcută la costuri mai reduse [1].

Inițial, acest sistem de operare a fost dezvoltat de Android Inc, o mică companie din Statele Unite ale Americii, fiind mai apoi preluat de Google în anul 2005 și lansat în 2007, în colaborare cu Open Handset Alliance [2].

În august 2008, Google lansează Android market, unde dezvoltatorii pot încarca propriile aplicații destinate dispozitivelor mobile, aplicații pe care utilizatorii le pot accesa și descărca gratuit. Acest lucru reprezintă o caracteristică de bază a dispozitivelor ce rulează Android, prin intermediul căreia are loc extinderea funcționalității dispozitivului. Lansarea inițială a magazinului nu a oferit suport pentru plata aplicațiilor, această caracteristică fiind adăugată mai apoi, la începutul anului 2009 [1].

Sistemul de operare Android a fost oferit ca open-source în anul 2008, datorită unui plan de marketing foarte bine pus la punct. Acesta oferea posibilitatea unei distribuții libere și totodată îmbunătățirea lui, atât de către utilizatori, cât și de producătorii de dispozitive smart. Creșterea numărului de dispozitive de tip smart pe întregul glob însemna un număr mai mare de utilizatori ce foloseau sistemul de operare Android, respectiv motorul de căutare Google, ceea ce determina obținerea de venituri prin intermediul anunțurilor publicitare (ad revenue).

Astfel, Android a realizat un impact semnificativ pe piața smartphone. Doi ani după ce primul dispozitiv Android a fost lansat (Octombrie 2008), Android a devenit a doua (după BlackBerry) cea mai răspândită platformă pentru smartphone-uri, acesta acaparând 26% din totalul smartphone-urilor din USA [1].

1.2.1.1. Arhitectura sistemului de operare Android

Sistemul de operare Android este alcătuit din componente software, dispuse pe mai multe straturi care comunică între ele. Acesta este împărțit în cinci secțiuni și patru straturi principale, afișate în diagrama din Figura 1.1.

Componentele sistemului de operare Android sunt:

- Linux Kernel
- Libraries
- Android Runtime
- Application Framework
- Applications

Stratul de bază, reprezentat de nucleul Linux, versiunea 2.6, este responsabil de controlul securității, a memoriei, a proceselor și bineînțeles, domeniul în care excelează Linux, partea de rețea. De asemenea, se ocupă și de o varietate de drivere cum ar fi, driverele pentru cameră, display, usb, tastatură etc, ce sunt esențiale în timpul rularii unei aplicații.

În următorul strat, numit „Libraries”, se regăsesc un set de librării native scrise în limbajul de programare C/C++, multe dintre acestea fiind open-source, ce sunt responsabile de

performanța diverselor componente. De exemplu, sursă manager este responsabil de compunerea diferitelor suprafețe de desen pe ecranul dispozitivului și gestionează accesul pentru diferite procese pentru a compune straturi grafice 2D și 3D. OpenGL reprezintă un nucleu de librării grafice și sunt folosite în consecință pentru accelerarea hardware 3D și 2D. Pentru stocarea de date, Android folosește SQLite. WebKit reprezintă un motor de căutare open-source, ce afișează conținutul web, SSL sunt librării pentru securitatea pe internet etc [3].

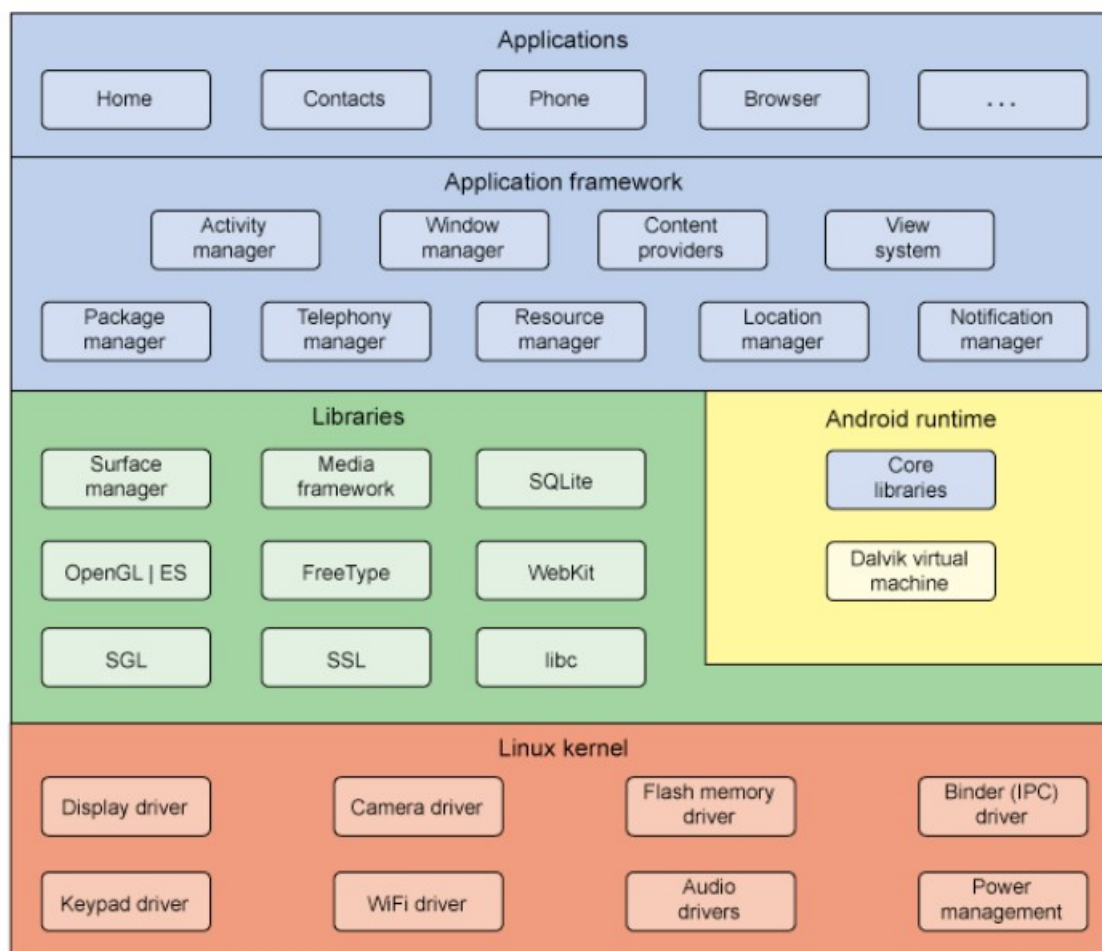


Figura 1.1: Arhitectura sistemului de operare Android [11]

Pe același nivel există și Android Runtime, unde se află componenta principală Dalvik Virtual Machine. A fost conceput special pentru Android unde rulează într-un mediu limitat, unde bateria limitată, procesorul, memoria și stocarea datelor sunt principalele probleme. Android oferă o ustensilă „dx” integrat care transformă codul byte generat de fișierul .jar, în fișier .dex. Cel din urmă fiind mult mai eficient pentru a rula pe procesoarele mici [3].

Stratul Application Framework, scris în limbajul Java, este un set de instrumente pe care îl folosesc toate aplicațiile, ce vin odată cu dispozitivul mobil, precum „Contacte” sau caseta SMS. Sau aplicații scrise de Google și dezvoltatorii Android. Manager-ul de activități gestionează ciclul de viață al aplicațiilor și oferă un sistem de navigare comun pentru aplicații ce se execută în diferite procese. Package Manager ține evidența aplicațiilor instalate pe dispozitiv. View System generează un set de butoane și liste folosite pe interfața utilizatorilor. Notification Manager este folosit pentru a personaliza alertele de afișare și alte funcții, etc [3].

Ultimul strat al arhitecturii Android este reprezentată de toate aplicațiile ce sunt folosite

de către utilizator. Acestea fiind accesate și descărcate prin intermediul magazinelor de aplicații, și care pot pune în evidență caracteristicile „smart” ale dispozitivului.

1.2.2. Android Studio

Android Studio este IDE⁴-ul oficial al celor de la Google, bazat pe software-ul IntelliJ IDEA⁵ și conceput pentru dezvoltarea de aplicații Android. Este disponibil pe Windows, Mac OS⁶ și sistemele de operare bazate pe Linux.

1.2.2.1. Componentele unei aplicații Android

Cele mai importante componente ale unei aplicații Android sunt:

Activitatea (activity): o activitate reprezintă o interfață cu utilizatorul, fereastră sau formular. O aplicație Android poate avea una sau mai multe activități, fiecare activitate având propriul său ciclu de viață, independent de ciclu de viață al procesului asociat aplicației. Fiecare activitate are propria stare și datele acesteia pot fi salvate sau restaurate. Are un ciclu de viață complex deoarece aplicațiile pot avea activități multiple și doar una din ele este în prim-plan. Utilizând managerul de activități, sistemul Android gestionează o stivă de activități, care se găsesc în diferite stări (pornire, în execuție, întreruptă, oprită, distrusă) [4].

Intent: este un obiect de mesagerie pe care îl putem utiliza pentru a solicita o acțiune de la o altă componentă a aplicației. Practic este o structură pasivă de date ce conține o descriere abstractă a unei acțiuni ce trebuie executată. Deși intent-urile facilitează comunicarea între componente în mai multe moduri, există trei cazuri fundamentale de utilizare:

- încărcarea unei activități: se poate da start unei noi instanțe a unei activități prin intermediul metodei „startActivity()”, care va primi ca și parametru un obiect de tip intent. Dacă se dorește returnarea unor rezultate de la o anumită activitate, se va folosi metoda „startActivityForResult()”, rezultatele constând prin primirea unui intent separat de care se va ocupa callback-ul „onActivityResult()”, al activității unde se așteaptă rezultatele.
- pornirea unui service: se poate da start unui service ce va produce o anumită operație (ex: descărcarea unui fișier) prin intermediul metodei „startService()”, ce va primi ca și parametru un intent.
- livrarea unui broadcast: se poate livra un broadcast către alte aplicații prin intermediul metodei „sendBroadcast()” sau „sendOrderedBroadcast()”, ce vor primi ca și parametru un intent.

Service: este o componentă a aplicației ce reprezintă fie dorința unui solicitant de a efectua o operațiune mai lungă în timp ce nu interacționează cu utilizatorul, fie procesul de a furniza funcționalități pentru alte aplicații pe care sa le utilizeze. Un lucru important de reținut este faptul ca un service nu este un proces separat sau un thread⁷.

4 IDE (Integrated Development Environment) – reprezintă o aplicație software care oferă un set de facilități programatorilor pentru dezvoltarea de aplicații. Un IDE constă din cel puțin un editor de cod sursă și un debugger.

5 IntelliJ IDEA – reprezintă un IDE scris în Java pentru dezvoltarea aplicațiilor software pentru calculator fiind dezvoltat de către cei de la JetBrains.

6 Mac OS – este un sistem de operare produs de către firma Apple Inc. pentru computerele sale

7 Thread – este un fir de execuție și reprezintă cea mai mică secvență de instrucțiuni ce poate fi gestionată de către un sistem. Rolul său este de a ajuta în execuția sarcinilor în paralel. Thread-urile lucrează independent și asigură utilizarea la capacitate maximă a procesorului, sporind performanța acestuia.

Acesta are două caracteristici principale:

- facilitatea prin care aplicația spune sistemului despre ceva ce vrea să execute în fundal (chiar și atunci când utilizatorul nu interacționează direct cu aplicația). Acesta corespunde prin apelarea metodei „*startService()*”, care solicită sistemului să planifice execuția serviciului, ce va rula până când serviciul s-a executat sau altcineva îl oprește explicit.

- facilitatea prin care aplicația expune o parte din funcționalitatea ei, altor aplicații. Aceasta corespunde apelării metodei „*bindService()*”, care permite realizarea unei conexiuni de lungă durată serviciului pentru a interacționa cu acestea.

Content Provider: poate ajuta o aplicație să gestioneze accesul la datele stocate de ea înșiși sau de alte aplicații, și să ofere o modalitate de a partaja date cu alte aplicații. Acesta încapsulează datele și oferă mecanisme pentru definirea securității datelor. Cel mai important aspect la content provider este acela că aceștia se pot configura pentru a permite altor aplicații să acceseze și să modifice în siguranță informațiile dintr-o baza de date.

Broadcast: Aplicațiile Android pot trimite sau primi mesaje broadcast de la sistemul Android și din alte aplicații Android. Aceste broadcast-uri sunt trimise atunci când apare un eveniment de interes. De exemplu, sistemul Android trimite mesaje broadcast atunci când apar diverse evenimente ale sistemului, cum ar fi momentul când sistemul pornește sau dispozitivul începe să se încarce. Aplicațiile pot trimite de asemenea, broadcast-uri personalizate, de exemplu, pentru a notifica alte aplicații despre ceva de care ar putea fi interesate (ex: descărcarea de noi date).

1.2.2.2. Ciclul de viață a unei activități

Activitatea este una din cele mai importante componente ale unei aplicații Android deoarece este strâns legată de interfața cu utilizatorul prin care acesta poate să interacționeze.

Înțelegerea modului în care se controlează activitatea va permite [4]:

- utilizarea ciclului de viață al activității pentru a crea, vizualiza, utiliza și opri activitățile.
- salvarea datelor utilizatorului înainte ca activitatea să fie oprită și restaurarea acestora atunci când aceasta este reafișată pe interfața dispozitivului.
- crearea de aplicații cu mai multe activități.

Din momentul în care activitatea este creată și până în momentul în care este distrusă, ea trece printr-o serie de etape, cunoscute sub denumirea de ciclu de viață al activității:

- în execuție (eng. *running*) – activitatea se află în prim-plan și este vizibilă, astfel încât utilizatorul poate interacționa cu aceasta prin intermediul interfeței grafice pe care o ofera [5].
- intreruptă temporar (eng. *paused*) – activitatea pierde prim-planul, deoarece o altă activitate este executată. Activitatea se află în fundal și este (parțial) vizibilă, o astfel de situație având loc în momentul în care o altă activitate a fost pornită, însă interfața sa grafică este transparentă sau nu ocupă întreaga suprafață a dispozitivului de afișare. Un exemplu ar fi lansarea unei fereastre de dialog. De asemenea, în cazul în care dispozitivul intra în modul *sleep*, activitatea este oprită temporar. Activitatea își poate relua execuția (*onResume()*) și este plasată înapoi în prim-plan.
- oprită (eng. *stopped*) – activitatea se află în fundal și este complet ascunsă. O astfel de situație este întâlnită în momentul în care o altă activitate a fost pornită, iar interfața sa grafică ocupă întreaga suprafață a dispozitivului de afișare. Activitatea ce a fost scoasă din prim-plan rămâne însă activă în memorie, însă poate fi distrusă de sistemul de operare

- dacă necesarul de memorie disponibilă nu poate fi întrunit din cauza sa [5].
- inexistentă (eng. *Destroyed*) – activitatea a fost terminată sau distrusă de către sistemul de operare, rularea sa impunând crearea tuturor elementelor de interfață ca și când ar fi fost accesată inițial [5].

Ciclul de viață a unei activități este descrisă în schema de mai jos:

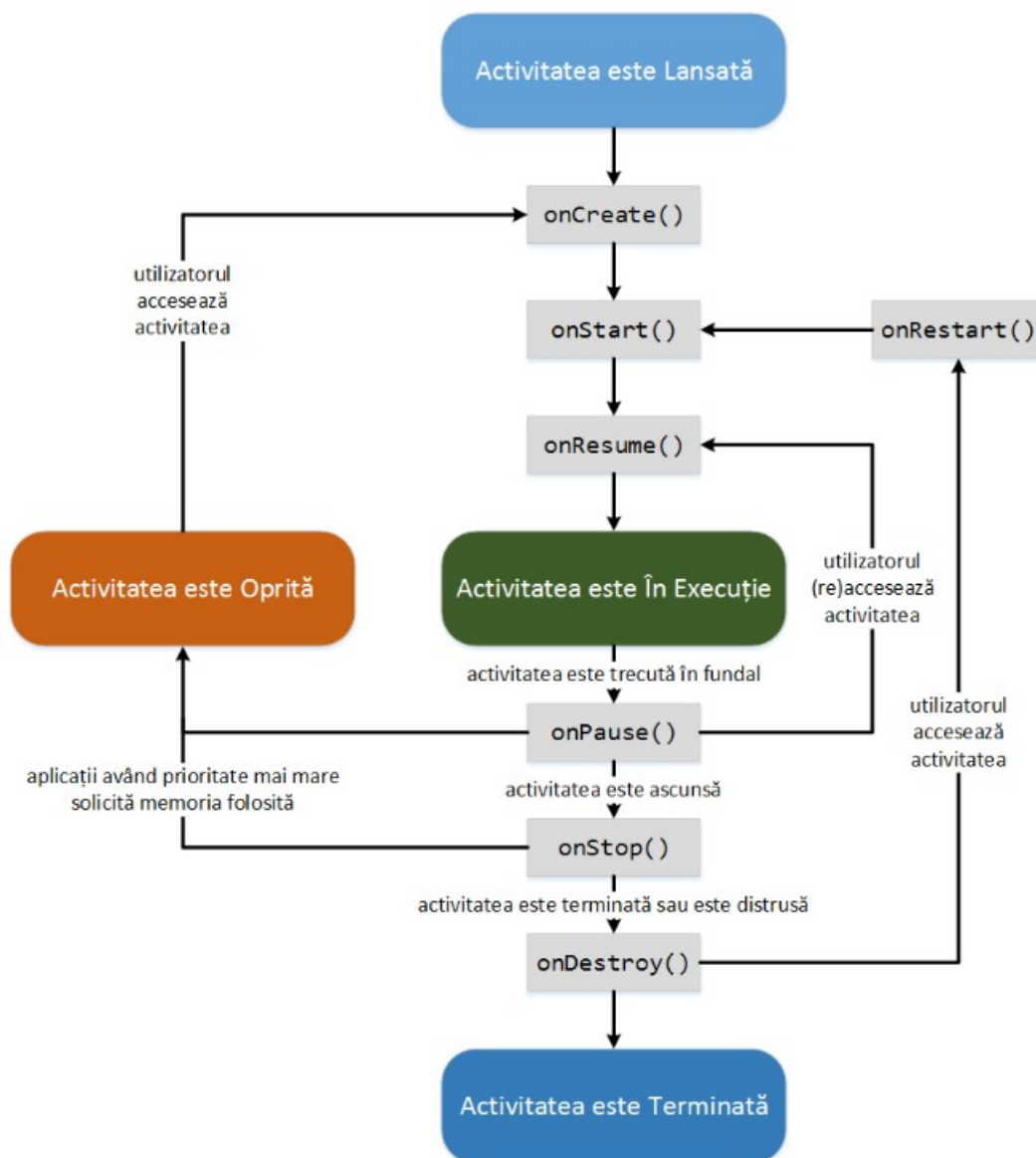


Figura 1.2: Ciclul de viață al unei activități [5]

Tranziția unei activități dintr-o stare în alta este notificată prin intermediul unor metode, care pot fi suprascrise pentru a realiza diferite operații necesare pentru gestiunea memoriei, asigurând persistența informațiilor și a consistenței aplicației Android, în situația producerii de diferite evenimente [5]:

- *onCreate()* - apelată în momentul în care activitatea este creată. Este responsabilă de încărcarea interfeței grafice, obținerea de referințe către elementele grafice, indicarea unor obiecte de tip “*listener*”, care gestionează evenimentele legate de interacțiunea cu

utilizatorul și realizarea unor conexiuni către alte modele de date [5].

- *onRestart()* - apelată atunci când activitatea a fost oprită și ulterior repornită; este urmată întodeauna de metoda *onStart()* [5].
- *onStart()* - apelată înainte ca activitatea să apară pe ecran. Poate fi urmată de metoda *onResume()* dacă activitatea trece în prim-plan sau de metoda *onPause()* dacă activitatea trece în fundal [5].
- *onResume()* - apelată înainte ca activitatea să interacționeze cu utilizatorul. Această metodă va fi folosită pentru a porni servicii sau cod care trebuie să ruleze atâta timp cât aplicația este afișată pe ecran. Este urmată întodeaună de metoda *onPause()* [5].
- *onPause()* - apelată înainte ca activitatea să fie întreruptă temporar, iar o altă activitate să fie rulată. Metoda va fi utilizată pentru a opri servicii sau cod care nu trebuie să ruleze atâta timp cât activitatea se află în fundal (pentru a nu consuma timp de procesor) [5].
- *onStop()* - apelată în momentul în care activitatea este ascunsă, fie din cauza că urmează a fi distrusă, fie din cauza că o altă activitate, a cărei interfață grafică ocupă întreaga suprafață a dispozitivului de afișare, urmează să devină vizibilă. Poate fi urmată de metoda *onRestart()*, dacă activitatea urmează să interacționeze din nou cu utilizatorul sau de metoda *onDestroy()*, dacă activitatea urmează să fie terminată sau distrusă de sistemul de operare [5].
- *onDestroy()* - apelată înainte ca activitatea să se termine sau să fie distrusă de către sistemul de operare (automat sau manual) din lipsă de memorie. Este apelată pentru eliberarea resurselor ocupate [5].

1.2.2.3. Structura de directoare și fișiere în Android Studio

Fiecare aplicație Android are un fișier numit *AndroidManifest.xml*, ce poate fi accesat din folderul *manifest* (vezi Figura 1.3) reprezentând fișierul de configurare al aplicației. Acesta conține informații esențiale despre aplicație, necesare instrumentelor de construire Android, sistemului de operare Android și magazinului Google Play, precum:

- numele pachetului aplicației: acesta identifică aplicația pe dispozitivele mobile și în magazinul Google Play.
- componentele aplicației: ce include toate activitățile, serviciile, furnizorii broadcast și de conținut.
- permisiunile de care are nevoie aplicația pentru a accesa părți protejate ale sistemului sau alte aplicații.
- caracteristicile hardware și software pe care aplicația le solicită dispozitivelor ce doresc rularea aplicației.

Fișierul *java* conține fișierele cod sursă Java ale aplicației, organizate în pachete. Pot exista mai multe pachete în aplicația Android. Întodeauna este o practică bună de a împărți codul sursă al aplicației în pachete diferite pe baza funcționalităților sale principale. Toate fișierele sursă ale activităților, serviciilor etc. se introduc în acest fișier [6].

Folderul *res* este locul unde toate resursele externe ale aplicației sunt stocate. Printre care se pot enumera: imagini, animații, fișierele audio, fișierele XML ale activităților, etc.

Fișierul *build.gradle (module: app)*, permite configurarea setărilor pentru modulul specific în care se află. De aici se poate seta nivelul API necesar ca un dispozitiv să poată rula aplicația (*minSdkVersion*), specifică nivelul API folosit pentru testarea aplicației, importarea de librării etc.

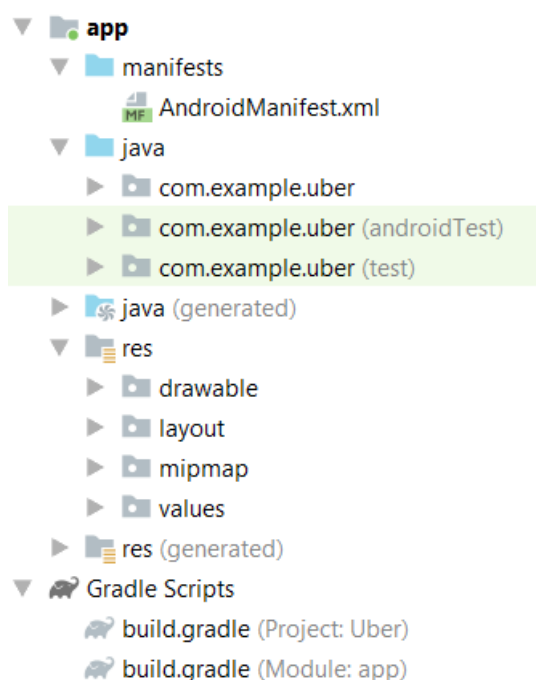


Figura 1.3: Structura directoarelor și fișierelor în Android Studio

1.2.3. Firebase

Firebase este o platformă pentru construirea aplicațiilor mobile și aplicațiilor web. Acesta a fost înființată în anul 2011 de către Firebase, ca mai apoi în anul 2014 să fie preluată de către cei de la Google.

Firebase reprezintă o tehnologie ce oferă posibilitatea dezvoltării de aplicații mobile fără a mai fi necesară implementarea părții de server, astfel făcând dezvoltarea aplicațiilor mobile mult mai ușoară și rapidă.

În momentul în care se construiește o aplicație, sunt o mulțime de lucruri ce trebuie a fi luate în calcul. Setarea și mentenanța unui server poate fi destul de dificilă deoarece asta ar implica:

- crearea sau cumpararea unui server ce trebuie menținut online 24/7.
- trebuie implementat tot codul din back-end.
- crearea și optimizarea unei baze de date.
- administrarea unui server necesită, de asemenea, multă atenție și timp.

Din aceste motive au apărut serviciile Firebase care nu doar au grijă de neccesitățile menționate mai sus, dar și oferă o mulțime de alte beneficii pe care i le oferă dezvoltatorului ca acesta să poată crea aplicații mult mai complexe.

Printre cele mai importante servicii oferite de Firebase fiind:

- Realtime Database – oferă posibilitatea stocării datelor și totodată, aceasta se ocupă de sincronizarea datelor între utilizatorii conectați la baza de date, în timp real.
- Stocare – prin care utilizatorul poate să stocheze un anumit conținut necesar aplicației, precum imagini sau videoclipuri.
- Autentificare – oferindu-i dezvoltatorului o varietate de modalități prin care aplicația lui să efectueze autentificarea (facebook, google, yahoo, mail și parola, twitter, github, etc.),

- cât și metode ce efectuează autentificarea propriu-zisă și a interogării acesteia.
- Funcțiile Cloud – funcții JavaScript sau TypeScript prin care dezvoltatorul poate implementa anumite funcționalități ce necesită securizare sporită, precum efectuarea plăților într-o aplicație.
- Hosting – ce oferă gazduire rapidă și sigură aplicației web, conținutului static sau dinamic și a microservicilor.

1.2.4. Limbajul Java

Java este un limbaj de programare orientat-obiect, puternic tipizat, conceput de James Gosling la Sun Microsystems (acum filială Oracle) la începutul anilor '90, fiind lansat în anul 1995. A fost inițial folosit pentru aplicații sau applets⁸. Java folosește un format special, cunoscut sub numele de *byte code*, în locul codului mașină. Cu ajutorul unui program de interpretare Java, *byte code*-ul poate fi executat pe orice calculator. Acest program de interpretare Java se numește Java Virtual Machine (JVM) și este disponibil astăzi pe majoritatea dispozitivelor [7].

Programele Java constau în seturi de clase, fișierele sursă ale acestora fiind scrise în limbajul Java. Aceste fișiere sursă sunt apoi compilate în fișiere “.class”, fișiere ce conțin *byte code* în locul codului mașina specific procesorului. Ca mai apoi, fișierele “.class”, fiind executate de către un JVM [7].

Datorită faptului că programe sunt rulate de către un JVM și nu accesează în mod direct sistemul de operare, face ca aceste programe Java să fie foarte portabile. Un program Java (care respectă standardul și anumite reguli) poate rula nemodificat pe toate platformele.

O altă proprietate al limbajului Java este gestionarea automată a memoriei. Java gestionează alocarea și distribuirea memoriei pentru crearea de noi obiecte. Programul nu are acces direct la memorie. Așa numitul “*garbage collector*”, șterge automat obiectele la care nu există niciun indicator activ.

1.2.5. JavaScript

JavaScript a fost dezvoltat prima dată de către firma Netscape, cu numele de Live Script, un limbaj de script care extindea capacitățile HTML. După lansarea limbajului Java, Netscape a început să lucreze cu firma Sun Microsystems, cu scopul de a crea un limbaj de script cu o sintaxă și o semantică asemănătoare cu cea a limbajului Java. Iar din motive de marketing, numele noului limbaj de script a fost schimbat în JavaScript. JavaScript a apărut din nevoia ca logica și inteligența să fie și pe partea de client, nu doar pe partea de server. Dacă toată logica este pe partea de server, întreaga prelucrare este făcută la server, chiar și pentru lucruri simple, așa cum este validarea datelor. Astfel, JavaScript îl înzestrează pe client și face ca relația să fie un adevărat sistem client-server [8].

Limbajul HTML oferă autorilor de pagini Web o anumită flexibilitate, dar statică. Documentele HTML nu pot interacționa cu utilizatorul în alt mod mai dinamic, doar pune la dispoziția acestuia, legături la alte resurse (URL-uri). Crearea de CGI-uri⁹ a dus la îmbogățirea posibilităților de lucru. Astfel un pas important către interacțiunea paginilor web a fost realizat de JavaScript, care permite inserarea în paginile web a script-urilor care se execută în cadrul paginii

8 Applets – un program Java compilat, al cărui nume este referit într-o pagină web. Când pagina respectivă este încărcată într-un browser, programul va fi și el încărcat și lansat în execuție. Acesta va fi capabil de afișarea de informații, cât și să citească date și să le prelucereze.

9 CGI (Common Graphics Interface) – programe care rulează pe serverul Web și care acceptă informații primite din pagina de web și returnează cod HTML.

web, mai exact în cadrul browser-ului utilizatorului, ușurând astfel și traficul dintre server și client [8].

JavaScript este dependent de mediu, acesta fiind un limbaj de scriptare. Software-ul ce rulează de fapt programul este browser-ul web. JavaScript este un limbaj în totalitate interpretat, codul scriptului fiind interpretat de browser înainte de a fi executat. Acesta nu necesită a fi compilat sau preprocesat, ci rămâne parte integrantă a documentului HTML. JavaScript este bazat pe obiecte, acesta nefiind un limbaj de programare orientat obiect, ca Java. Modelul de obiect JavaScript este bazat pe instanță și nu pe moștenire. O altă caracteristică importantă este acela că reprezintă un limbaj foarte flexibil, în JavaScript fiind posibilă lucrul cu o variabilă, deși nu i se cunoaște tipul specificat înainte de rulare [8].

1.2.6. XML

Extensible Markup Language (XML) este un meta-limbaj utilizat în activitatea de marcare structurală a documentelor, a cărei specificație a fost dezvoltată începând cu 1996 în cadrul Consorțiului World Wide Web (W3C).

XML este un set de reguli pentru a crea formate text care permit structurarea datelor. Prin intermediul fișierului XML, calculatorul gestionează și citește cu ușurință datele, cât și asigură că structura datelor este corectă. Este extensibil, independent de platformă și complet compatibil cu unicode¹⁰ [9].

Documentele XML sunt realizate din unități de stocare numite entități, ce conțin date parsabile sau neparsabile. Datele parsabile sunt realizate din caractere, unele din ele formând date caracter. Marcajele codifică o descriere a schemei de stocare a documentului și structura logică. XML a fost elaborat pentru separarea sintaxei de semantică pentru a furniza un cadru comun de structurare a informației, construirea de limbaje de mark-up pentru aplicații din orice domeniu și asigurarea independenței de platforma [9].

Android Studio folosește formatul XML pentru crearea fișierelor layout, în care sunt adăugate elementele grafice ce sunt afișate pe interfața dispozitivelor.

¹⁰ Unicode - este un format pentru codarea, stocarea și interpretarea textelor pe suporturi informatice. Este proiectat pentru ca oricărei litere (caracter) din orice limbă, de pe orice platformă de hardware sau software, să îi corespundă un număr unic și neechivoc.

Capitolul 2. Proiectarea aplicației

Aplicația a fost dezvoltată prin intermediul IDE-ului Android Studio destinat dezvoltării aplicațiilor mobile. Aplicația dispune de două interfețe distincte, ce corespund tipului de utilizator ce dorește a folosi aplicația. Aceasta fiind destinată atât utilizatorilor de tip pasager, ce doresc a se transporta, cât și utilizatorilor de tip șofer, ce oferă serviciile de transport.

2.1. Aplicația client Android

Aplicația vine cu o serie de funcționalități pe care utilizatorii le pot folosi pentru a-și desfășura serviciile în siguranță, toate acestea fiind expuse printr-o interfață foarte simplă și intuitivă.

Principalele funcționalități ale aplicației sunt reprezentate de:

- Înregistrare/autentificarea utilizatorului:
 - necesară pentru ca utilizatorii să poată folosi serviciile puse la dispoziție de către aplicație.
 - permite utilizatorilor să își creeze un cont sau să se autentifice în cadrul aplicației.
 - opțiunea ca utilizatorii să-și reseteze parola, prin primirea unui mail unde se va introduce noua parola.
- Interfața reprezentată de o hartă în care utilizatorul va fi localizat și afișat
 - permite utilizatorului să vizualizeze poziția sa în timp real pe hartă
- Funcția de Place Auto Complete
 - această funcție este oferită pasagerilor pentru a-i ajuta în selectarea destinației la care aceștia doresc să se deplaseze prin oferirea unor variante de adrese pe baza cuvintelor sau grupurilor de litere introduse.
- Setarea profilului
 - utilizatorii vor dispune de o interfață în care aceștia vor putea să își seteze numele, numărul de telefon, o imagine de profil iar șoferii pe lângă aceste opțiuni vor mai avea posibilitatea setării mărcii mașinii, numărului de înmatriculare cât și tipul de serviciu oferit. Toate aceste informații fiind necesare pentru identificarea utilizatorilor și folosite pentru realizarea schimbului de date între aceștia în momentul în care se efectuează o cursă de transport.
- Plasarea unei cereri de transport
 - pasagerii pot plasa o cerere de transport, cerere ce va fi preluată de către cel mai apropiat șofer disponibil.
- Schimbul de informații dintre șofer și pasager.
 - în momentul în care un șofer preia cererea de transport a unui pasager, va avea loc un schimb de date între aceștia astfel încât pasagerul va putea vizualiza informațiile setate de șofer pe profilul acestuia cât și invers, șoferul va putea vizualiza informațiile pasagerului prin intermediul interfeței aplicației.
- Desenarea rutelor
 - funcție ce le este oferită șoferilor în momentul în care trebuie să efectueze o cerere de transport, fiindu-le afișată ruta dintre aceștia și locul în care se afla pasagerul, cât și în momentul în care aceștia preiau pasagerul, fiindu-le afișată ruta către destinația selectată de pasager, în cazul în care acesta ar fi selectat o anumită destinație.
- Implementarea unui istoric al curselor
 - este reprezentată de o interfață în care utilizatorii pot vizualiza cursele efectuate de

aceștia cât și anumite informații despre acestea, precum ruta, șoferul, respectiv pasagerul ce au efectuat respectiva cursă, distanța etc.

- Implementarea unui sistem de rating
 - funcție oferită pasagerilor prin care pot aprecia calitatea cursei efectuate de către un șofer, oferindu-i un număr de stele.
- Sistemul de plată prin PayPal
 - ca și formă de plată, utilizatorii vor putea plăti prin intermediul unor conturi PayPal.
- Afișarea tuturor șoferilor disponibili
 - pasagerii vor putea vizualiza toți șoferii disponibili de pe o anumită rază.

Majoritatea funcționalităților de mai sus, fiind descrise mai pe larg atât din punct al funcționalității, cât și al implementării lor, în cadrul capitolului trei.

2.1.1. Arhitectura aplicației client Android

Utilizatorul interacționează cu aplicația prin intermediul interfeței acesteia, moment în care va avea loc interacțiunea client-server, prin care utilizatorul își expune dorința de a utiliza anumite funcționalități, prin intermediul unor cereri de servicii și așteaptă răspuns de la server. Serverele Firebase, Google și PayPal, analizează și interoghează respectivele cereri, luând decizia dacă pot oferi respectivele cereri sau nu. În caz afirmativ, acestea interoghează bazele lor de date și vor oferi utilizatorului, serviciile și informațiile de care acesta are nevoie (vezi Figura 2.1).

Server-ul Firebase conține baza de date a aplicației, iar datorită acestui fapt, majoritatea funcționalităților implementate în aplicație, se învârt în jurul acestui server. Acesta intervine în mod direct în procesul de autentificare a utilizatorilor, atât pentru crearea conturilor, autentificarea utilizatorilor în aplicație și deconectarea lor, cât și pentru resetarea parolei unui anumit cont, prin punerea la dispoziție a unor metode folosite pentru apelarea serviciilor Firebase. De asemenea, acesta va conține și funcțiile JavaScript implementate și încărcate pe server, responsabile de plata utilizatorilor de tip șofer.

Server-ul Google interoghează cererile privind afișarea unor sugestii de locații, cereri create de funcția Place AutoComplete și de asemenea, cererile privind accesarea punctelor cardinale și a direcțiilor necesare creării rutelor pe hartă, interogând baza lor de date și oferind aceste date.

Server-ul PayPal fiind responsabil de efectuarea plăților utilizatorilor.



Figura 2.1: Arhitectura aplicației client Android

2.2. Diagrame UML

UML (Unified Modeling Language) este un limbaj de modelare bazat pe notații grafice folosit pentru a specifica, vizualiza, construi și documenta componentele unui program. UML este un limbaj cu ajutorul căruia se pot construi (descrie) modele. Un model surprinde un anumit

aspect al unui program și același model poate fi descris la diferite modele de abstractizare [10].

2.2.1. Diagramele use-case

Diagramele use-case au scopul de a descrie secvența de acțiuni pe care un program le execută atunci când interacționează cu entități din afara lui (actori) și care conduc la obținerea unui rezultat observabil și de folos actorului. În astfel de diagrame nu se precizează nimic despre cum este realizată (implementată) o anumită funcționalitate [10].

Mai jos, sunt prezentate cele trei diagrame UML de tip use-case ce corespund procesului de autentificare/înregistrare și modul de folosire al aplicației de către șofer/pasager, după ce aceștia s-au logat cu succes în aplicație. Fiind două interfețe separate, pentru cele două tipuri de utilizatori, ce conțin funcționalități diferite.

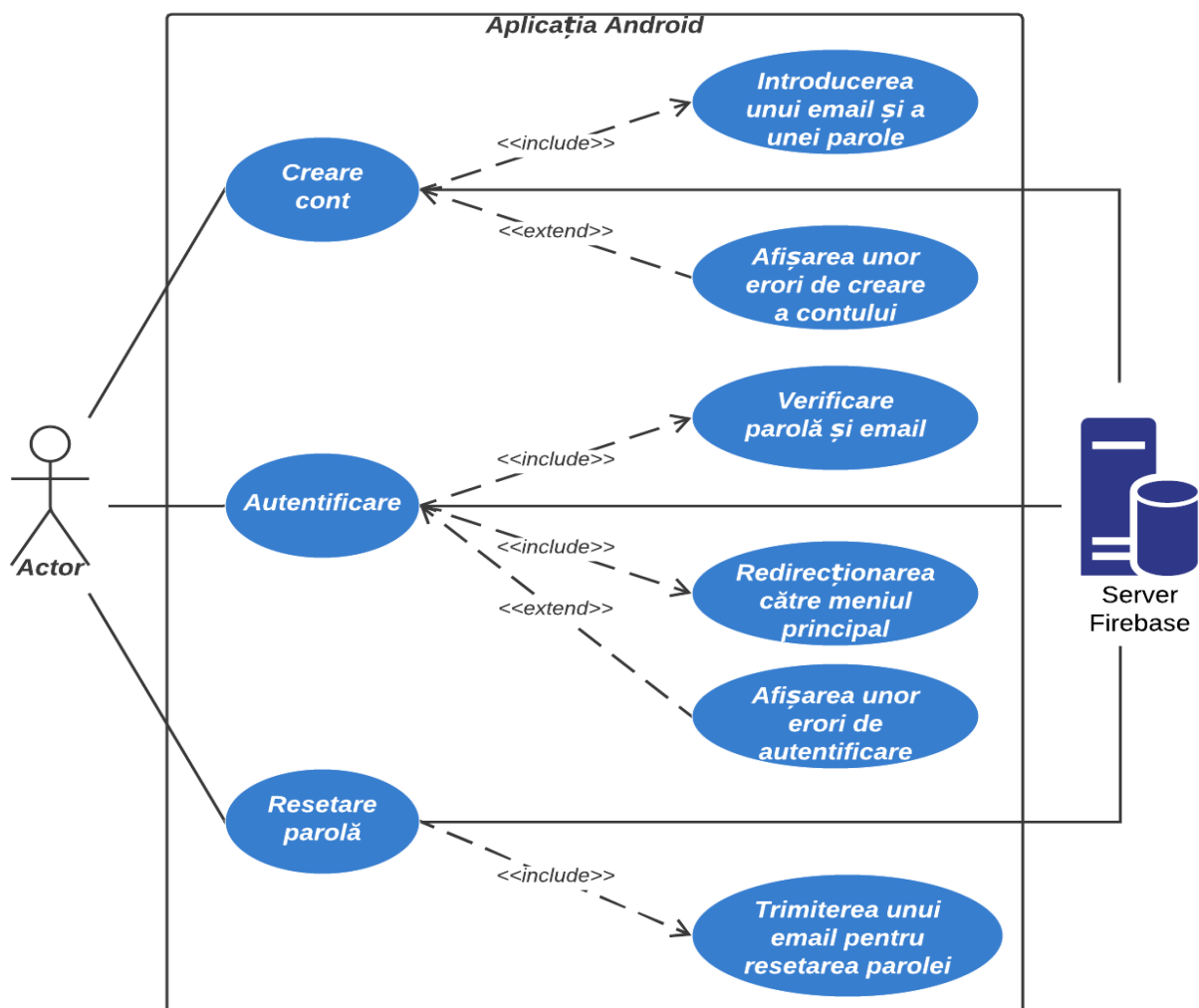


Figura 2.2: Diagrama use-case pentru autentificare

După cum se poate observa în diagrama de mai sus, în momentul în care utilizatorul accesează aplicația, este întâmpinat de o interfață ce îi oferă posibilitatea creării unui nou cont,

autentificarea în aplicație și respectiv, resetarea parolei unui cont deja existent. Toate aceste operațiuni fiind realizate prin intermediul serviciilor oferite de către serverul Firebase.

În Figura 2.3, sunt puse în evidență funcționalitățile de care vor dispune utilizatorii de tip pasager în momentul în care s-au autentificat în aplicație cu succes. Cele mai importante funcționalități ce caracterizează interfața pusă la dispoziție acestui tip de utilizatori fiind reprezentate de inițializarea cererii de transport, implementarea unui sistem de plată și setarea rating-ului șoferului. Funcționalitățile enumerate anterior reprezentând și diferențele dintre cele două interfețe oferite pentru cele două tipuri de utilizatori.

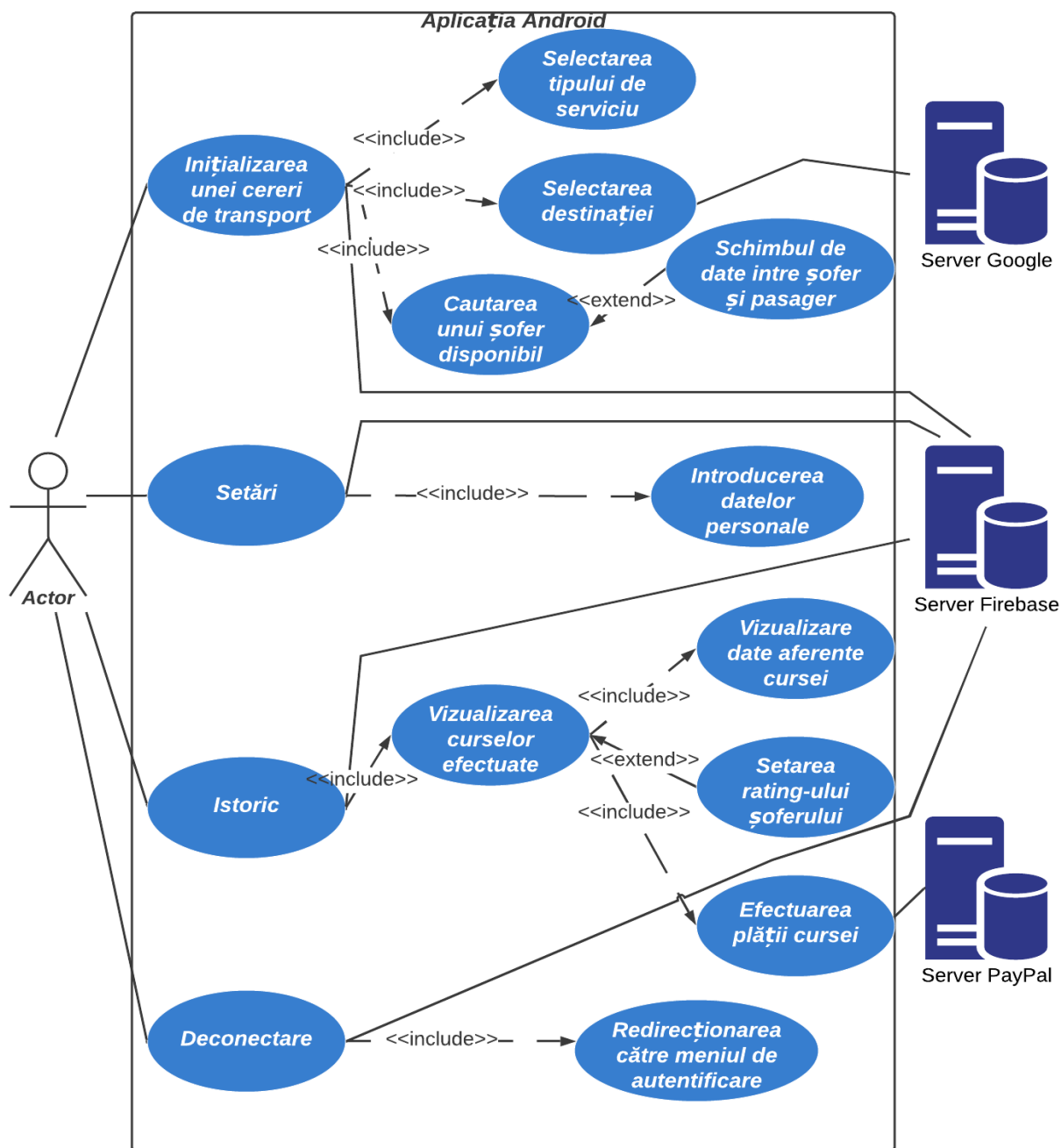


Figura 2.3: Diagrama use-case pentru utilizatorii de tip pasager

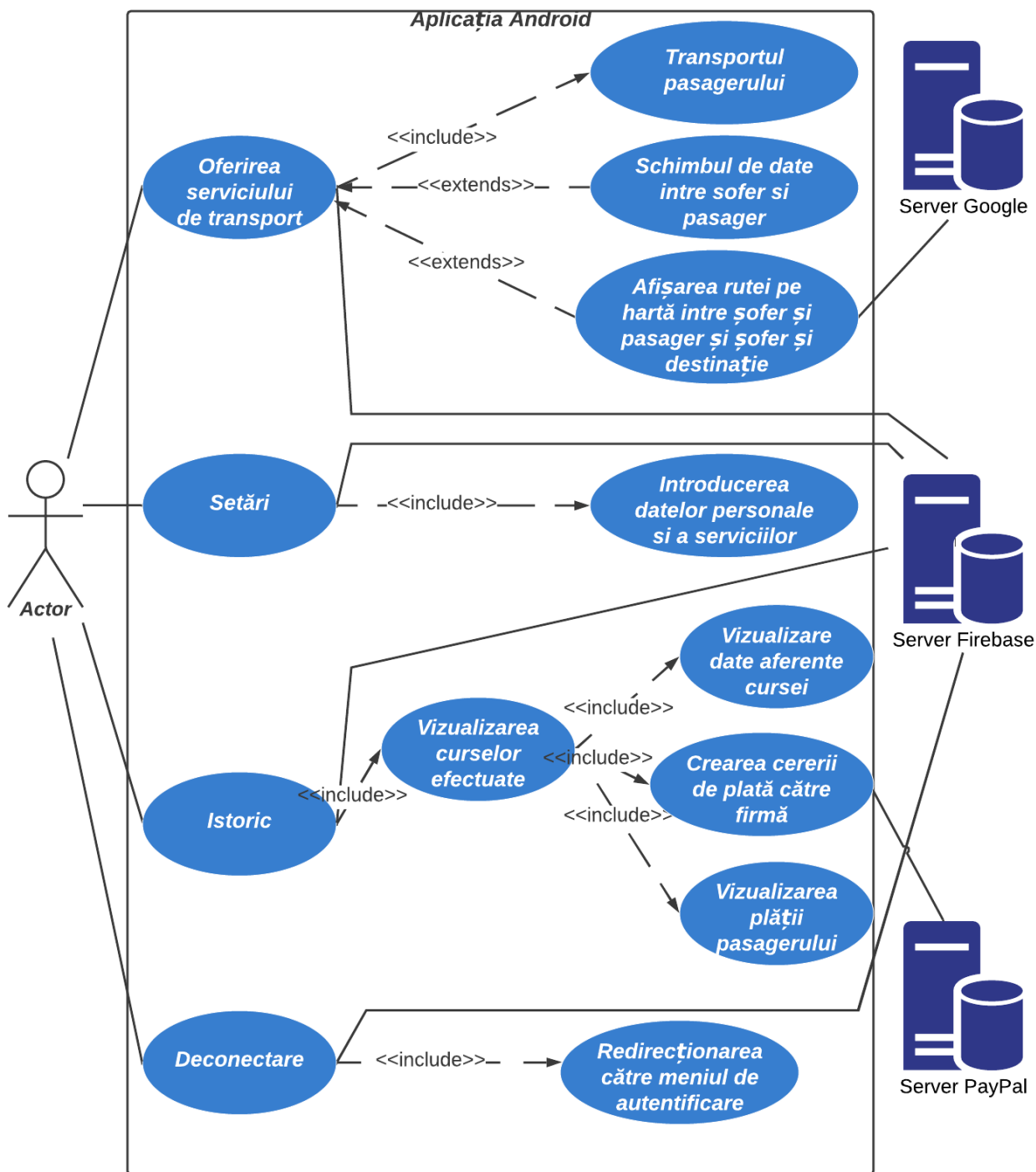


Figura 2.4: Diagrama use-case pentru utilizatorii de tip șofer

În Figura 2.4, sunt afișate funcționalitățile de care vor dispune utilizatorii de tip șofer, după autentificarea cu succes în aplicație. Funcționalitățile caracteristice acestei interfețe fiind reprezentate de oferirea serviciului de transport, afișarea rutelor pe harta interfeței aplicației, crearea cererii de plată către firma ce deține aplicația și vizualizarea plății pasagerului.

Capitolul 3. Implementarea aplicației

3.1. Crearea și configurarea unui proiect în Android Studio și conectarea acestuia la serverul Firebase

Crearea unui proiect în Android Studio se realizează prin apăsarea butonului „Create new project”, moment în care va fi afișată o fereastră prin intermediul căreia se va configura proiectul (pentru vizualizarea ferestrei de configurare consultați Figura 1 din Anexa 1.). Această configurare constă în alegerea unui nume pentru proiect și a limbajul de programare prin intermediul căruia dezvoltatorul va implementa aplicația, având de ales între Java și Kotlin. De asemenea dezvoltatorul va trebui să seteze id-ul aplicației ce se aseamănă cu numele unui pachet java (*com.example.myapp*). Prin intermediul acestui id, aplicația este identificată de către dispozitivul ce o rulează și de asemenea de către app stores¹¹. Tot aici are loc și selecția API level-ului, astfel dezvoltatorul își poate seta dispozitivele țintă ce rulează anumite versiuni de Android, pe care aplicația va fi rulată (vezi Figura 2 din Anexa 1.). După care se va apăsa butonul „Finish”, moment în care va dura câteva secunde până când aplicația se va configura.

Conectarea proiectului Android Studio la serverul Firebase se realizează prin accesarea consolei Firebase prin intermediul unui browser web (Chrome, Firefox, Opera etc.) și crearea unui nou proiect căruia i se va asocia un nume. În interfața imediat următoare se va introduce numele pachetului aferent aplicației Android, nume ce a fost setat la crearea proiectului Android și care poate fi vizualizat în modulul “*app*” din “*build.gradle*”. Tot aici se adaugă și SHA 1¹² asociat proiectului din Android ce poate fi găsit în tabul “*Grandle*”, în folderul “*android*”, fișierul “*signingReport*”.

După înregistrarea aplicației se va genera un fișier JSON ce va conține serviciile Google ce va trebui introdus în aplicația Android în folderul `<project>/<app-module>` printr-un simplu drag and drop al fișierului (vezi Figura 3 din Anexa 1.). De asemenea se va genera și un dependency¹³ ce trebuie introdus în fișierul “`<android>/build.gradle`” (vezi Figura 4 din Anexa 1.), dar și un plugin¹⁴ ce trebuie introdus în folderul `<project>/<app-module>/build.gradle` (vezi Figura 5 din Anexa 1.)

3.2. Logarea și înregistrarea

Pentru ca un utilizator să poată utiliza aplicația, acesta trebuie mai întâi să își creeze un cont prin intermediul unui email și o parolă, ca mai apoi să se logheze cu respectivul cont pentru a beneficia de funcționalitățile aplicației. În acest sens, aplicația vine cu o interfață simplă prin care utilizatorul își alege modul de folosire al aplicației, șofer respectiv pasager (vezi Figura 3.1),

11 App store – reprezintă un magazin online prin intermediul căruia utilizatorii pot cumpăra sau descarca diferite aplicații pentru dispozitivul lor mobile.

12 SHA 1 (Secure Hash Algorithm) – este un certificat de înregistrare reprezentată de o cheie unică asociată calculatorului. Este folosită în principal pentru a înainta celor de la Google dorința de a folosi anumite servicii API.

13 Dependency – reprezintă un modul de componente (clase, funcții, interfețe, metode etc) ce realizează anumite funcții sau operații, ce se bazează pe un alt modul de componente, pe care dezvoltatorii de aplicații îl pot importa în proiect cu scopul de a elimina crearea de la zero a respectivelor funcționalități. În momentul în care un modul de componente ce oferă dependențe suferă modificări de sintaxă, toate aplicațiile ce depind de respectivul modul sunt nevoite să-și schimbe la rândul lor modul de folosire al componentelor conform noii sintaxe.

14 Plugin – reprezintă un modul sau parte a unui software ce poate fi adăugat unui program, pagină web etc, cu scopul de a introduce noi funcționalități și caracteristici.

ca mai apoi să fie întâmpinat de interfața în care acesta își poate crea sau se poate loga în aplicație cu un cont deja existent (vezi Figura 3.2).

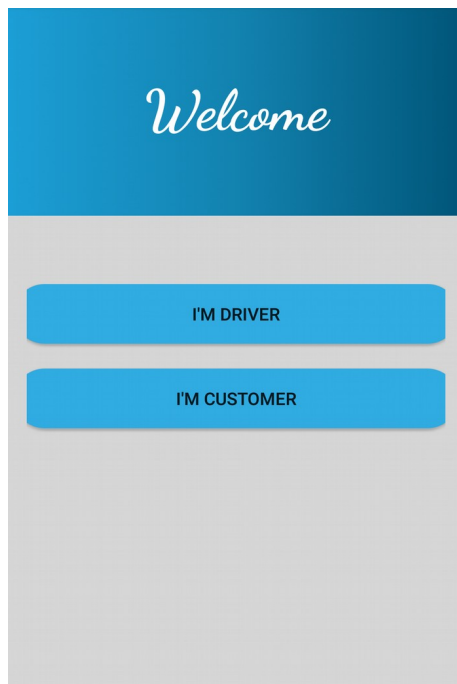


Figura 3.1: Selectarea tipului de utilizator

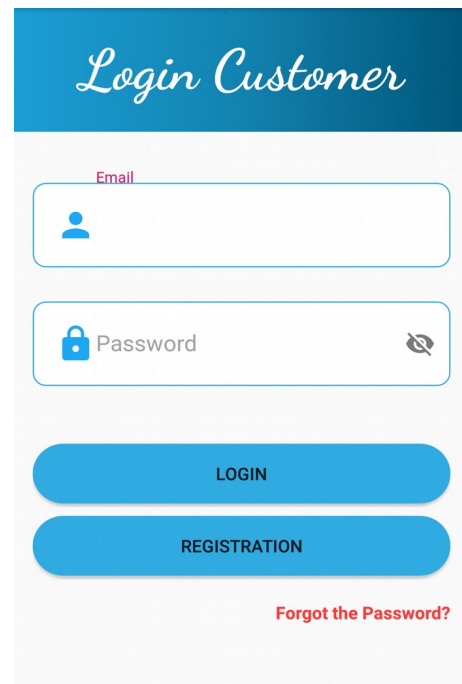


Figura 3.2: Logarea și înregistrarea unui utilizator

Logarea și autentificarea se realizează cu ajutorul unui serviciu Firebase și mai exact „*Firebase Auth*”. Pentru a conecta acest serviciu la aplicație trebuie ca mai întâi să selectăm din consola Firebase metoda prin care să se realizeze înregistrarea, în cazul de față, printr-un email și o parolă. Firebase oferă o varietate largă de posibilități prin care să se realizeze autentificarea, precum facebook, twitter, phone, yahoo, google etc. Selectarea metodei de autentificare se realizează din meniul „*authentication*” din consola Firebase, secțiunea „*sign-in*”, după care se setează pe *enable* varianta de înregistrare dorită (vezi Anexa 2.).

Pentru a dispune de serviciul de autentificare Firebase, prin intermediul căruia se va realiza funcția de autentificare a aplicației, va trebui ca în proiect să se realizeze importarea unui dependency specific Firebase și anume *firebase-auth*. Această dependency pune la dispoziție niște metode cum ar fi „*createUserWithEmailAndPassword()*” și „*signInWithEmailAndPassword()*” prin intermediul cărora are loc crearea utilizatorilor în baza de date și înregistrarea lor.

De asemenea pentru ca aplicația să beneficieze de serviciul de stocare a informațiilor utilizatorilor în baza de date a celor de la Firebase, trebuie ca în proiect să fie importată o dependentă specială, mai exact *firebase-database*, ce poate fi accesată din documentația Firebase.

Crearea conturilor utilizatorilor se realizează în momentul în care un utilizator apasă pe butonul „*Registration*” de pe interfață, buton ce va avea un listener¹⁵ ce va prelua acțiunea dată de utilizator pe interfața aplicației și care va conține o metodă specifică *firebase-auth* și mai exact „*createUserWithEmailAndPassword()*”, care va primi ca parametri datele introduse de către

¹⁵ Listener – reprezintă o metodă ce este apelată în momentul în care un utilizator interacționează cu un element de pe interfața dispozitivului (ex: click pe un buton, imagine, text etc.) și care are rolul de a produce un eveniment în funcție de tipul interacțiunii.

utilizator (email-ul și parola), prin intermediul cărora se va crea înregistrarea acestuia în baza de date.

```
mRegistration.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
    if (validateEmail() | validatePassword() && (validateEmail() &&
validatePassword())) {
        final String email = mEmail.getText().getText().toString();
        final String password = mPassword.getText().getText().toString();
        mAuth.createUserWithEmailAndPassword(email,
password).addOnCompleteListener(DriverLoginActivity.this, new
OnCompleteListener<AuthResult>() {
@Override
public void onComplete(@NonNull Task<AuthResult> task) {
    if (!task.isSuccessful()) {
        Toast.makeText(DriverLoginActivity.this, "Error: This email is
already used", Toast.LENGTH_SHORT).show();
    } else {
        String user_id = mAuth.getCurrentUser().getUid();
        DatabaseReference current_user_db =
FirebaseDatabase.getInstance().getReference().child("Users").child("Drivers").child(user_id).child("name");
        current_user_db.setValue(email);
    }
}}});
```

Mauth reprezintă o instanță a serviciului de înregistrare FirebaseAuth.

Însă înaintea creării propriu-zise a contului are loc o serie de verificări a datelor introduse, apelându-se metodele „*validateEmail()*” și „*validatePassword()*”, astfel email-ul și parola vor fi verificate astfel încât acestea să respecte un anumit pattern¹⁶.

După ce logarea sau înregistrarea are loc cu succes, se va apela listener-ul „*AuthStateListener*”, ce este pornit când activitatea ce îl conține este lansată, listener ce conține o metoda *override*¹⁷, „*onAuthStateChanged()*”, care se activează automat în momentul în care starea logării se schimbă, mai exact în momentul în care un utilizator se loghează, se deloghează sau își înregistrează un nou cont, fapt ce va duce la redirecționarea acestuia la următoarea interfață.

```
firebaseAuthListener = new FirebaseAuth.AuthStateListener() {
@Override
public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
    FirebaseUser user = FirebaseAuth.getInstance().getCurrentUser();
    if(user!=null){
        getUserAccType();
    }
}};
```

¹⁶ Pattern – un tipar pe care o entitate trebuie să îl respecte sau pe baza căruia este creată.

¹⁷ Metoda *override* – în programarea orientată-obiect, reprezintă o caracteristică de limbaj ce permite unei subclase să realizeze o implementare proprie pentru o metodă care deja este implementată de superclase sau clasele părinte.

În codul de mai sus, metoda creată, „*getUserAccType()*”, identifică tipul de utilizator ce dorește a utiliza aplicația, astfel încât utilizatorii de tip șoferi nu vor putea să se logheze în aplicație în secțiunea destinată pasagerilor, cât și invers. De asemenea, această metodă răspunde și de redirecționarea utilizatorilor către interfața principală în care aceștia pot folosi funcționalitățile principale ale aplicației, după ce s-au logat. Acest lucru efectuându-se prin intermediul unui intent.

Un lucru foarte important constă în faptul că listener-ul din codul de mai sus va trebui inițializat și activat în metoda „*onStart()*”, metodă ce este apelată mereu când o activitate este lansată.

```
mAuth.addAuthStateListener(firebaseAuthListener);
```

De asemenea, în metoda „*onStop()*”, când utilizatorul parasește activitatea și aceasta este mutată în background, va trebui eliminat acest listener.

```
mAuth.removeAuthStateListener(firebaseAuthListener);
```

Aplicația dispune și de metoda de recuperare a contului în cazul în care utilizatorul și-a uitat parola. Acesta funcție este activată în momentul în care utilizatorul apasă pe textul de pe interfața aplicației, „*Forgot the password?*”, moment în care va fi redirecționat către o nouă interfață în care va trebui să-și introducă email-ul, fapt ce va produce trimiterea unui link pe email-ul respectiv, prin care utilizatorul își va putea reseta parola.

Această funcționalitate a fost implementată cu ajutorul metodei „*sendPasswordResetEmail()*”, fiind o metodă pusă la dispoziție de către serviciul FirebaseAuth, care primește ca și parametru email-ul prin care va fi trimis link-ul de resetare al parolei. Acestei metode aplicându-se un listener de tip „*onCompleteListener*”, care este activat în momentul în care un task s-a executat, în cazul de față, în momentul în care se termină procesul de încercare a trimiterii mail-ului de resetare a parolei.

```
sendEmail.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        progressBar.setVisibility(View.VISIBLE);
        firebaseAuth.sendPasswordResetEmail(userEmail.getText().toString()).
        addOnCompleteListener(new OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                progressBar.setVisibility(View.GONE);
                if(task.isSuccessful()){
                    Toast.makeText(ForgotPassword.this, "The link to reset your
password was sended", Toast.LENGTH_LONG).show();
                }else
                {
                    Toast.makeText(ForgotPassword.this,
task.getException().getMessage(), Toast.LENGTH_LONG).show();
                }
            }
        });
    }
});
```

3.3. Setarea API-ului Google Maps

Deoarece interfața principală a aplicației va fi reprezentată de o hartă, prin intermediul careia utilizatorii pot vizualiza informații privind distanțele dintre pasageri și șoferi, denumirile anumitor locații și rutele de transport oferite șoferilor drept ghid în trafic, este necesar ca aplicația să folosească serviciile Google Maps oferite de cei de la Google.

Activarea acestor servicii se realizează accesând prin intermediul unui browser web, consola Google, de unde se poate crea o cheie API¹⁸ și totodată activarea SDK-ului¹⁹ „*Maps SDK for Android*”, prin simplul click pe acesta și setarea lui pe enable. Crearea unei chei API se face din meniul „*APIs & Services*”, tabul „*Credentials*” de unde se va apăsa pe butonul „*Create Credentials*” (vezi Anexa 3.).

API-ul „*Maps SDK for Android*”, gestionează accesul la serverele Google Maps, descărcarea datelor, afișarea hărților și răspunsurile la gesturile făcute pe hartă de către utilizatori. Putem utiliza de asemenea apelurile API pentru a adăuga markeri, poligoane și suprapuneri la o hartă de bază și pentru a schimba vizualizarea unei anumite părți a hărții. Aceste obiecte oferă informații suplimentare pentru locațiile hărții și permit interacțiunea utilizatorilor cu harta.

Pentru ca aplicația să dispună de o interfață reprezentată de o hartă, este necesară crearea unei activități speciale în Android Studio și anume „*Google Maps Activity*”. O astfel de activitate va conține în metoda *onCreate()*, inițializarea mapei și crearea acesteia și totodată această activitate va conține o metodă de tip *override*, sub denumirea „*onMapReady()*”, ce va primi ca parametru, mapa inițializată și creată, în care se pot adăuga diferite elemente de afișare (segmente, poligoane, markers etc.) și unde se pot face verificarea de permisiuni (permisiunea de afișare a locației utilizatorului).

În fișierul XML ce se va crea, aferent activității, trebuie introdusă cheia API generată mai sus și astfel se va face legătura dintre serviciile oferite de Google și aplicație.

3.4. Actualizarea periodică a locațiilor utilizatorilor

Pentru ca aplicația să dispună de funcția de actualizare a locațiilor, astfel încât utilizatorii să aibă în permanență locația în timp real și precisă pe hartă, s-a folosit un obiect de tipul „*FusedLocationProviderClient*”.

```
private FusedLocationProviderClient mFusedLocationClient;
```

Acest tip de obiect se folosește pentru a interacționa cu locațiile folosind fuziunea locațiilor. Pentru folosirea acestui concept, utilizatorii vor fi nevoiți să aibă activat GPS-ul pe dispozitivul lor. Obiectul declarat mai apoi se inițializează în metoda „*onCreate()*” al activității de tip Google Maps.

```
mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
```

Pentru ca aplicația să afișeze și să actualizeze locația utilizatorului, este necesar mai întâi

18 Cheile API – prin intermediul acestor chei putem accesa anumite date sau servicii. O cheie Api este un șir de caractere ce identifică o aplicație și cererile efectuate de respectiva aplicație. Astfel în momentul în care aplicația folosește niște servicii API, cheia API va face parte din cererea pentru folosirea respectivului serviciu API. Serviciu API care va identifica aplicația și va decide dacă oferă sau nu răspuns cererii pe baza unei analize făcute (număr de accesări etc.), aceste validări fiind necesare pentru securitatea sistemului.

19 SDK(Software Development Kit) – reprezintă un ansamblu de unelte (biblioteci, documentații, structuri de cod etc.) ce permit dezvoltatorului de a crea aplicații software pe o anumită platformă specifică..

verificarea și implementarea unor metode care vor cere utilizatorului permisiunile necesare pentru a dispune de locația sa. Astfel în momentul în care utilizatorul se loghează în aplicație, are loc testarea faptului dacă respectivul utilizator a oferit permisiunea de a accesa și afișa locația geografică a sa, dacă nu, se va crea o cerere ce va fi afișată pe interfața aplicației acestuia sub forma unui mesaj prin care îi este cerut acest lucru. Fără primirea acestei permisiuni, utilizatorul nu va putea utiliza funcția de a înainta o cerere de transport.

Mai jos este afișată metoda ce se apelează în momentul în care un șofer se loghează în aplicație:

```
private void connectDriver() {
    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M){
        if(ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)== PackageManager.PERMISSION_GRANTED){
            mFusedLocationClient.requestLocationUpdates(mLocationRequest,
mLocationCallback, Looper.myLooper());
            mMap.setMyLocationEnabled(true);
        }else{
            checkLocationPermission();
        }
    }
}
```

După cum se poate observa în codul de mai sus, în momentul în care un șofer se loghează în aplicație se face verificarea dacă acesta a oferit permisiunea de a accesa locația. În caz afirmativ, se va apela metoda „*requestLocationUpdates()*”, prin intermediul obiectului „*mFusedLocationClient*”. Metodă ce va oferi actualizarea periodică a locației utilizatorului.

În caz contrar, se apelează metoda creată „*checkLocationPermission()*”, ce va conține cererea ce va fi înaintată utilizatorului prin care i se va cere permisiunea.

```
private void checkLocationPermission() {
    if(ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED){

        if(ActivityCompat.shouldShowRequestPermissionRationale(this,Manifest.permission.AC
CESS_FINE_LOCATION)){
            new AlertDialog.Builder(this)
                .setTitle("Give Permission")
                .setMessage("Give permission message")
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        ActivityCompat.requestPermissions(DriverMapActivity.this,
new String[] {Manifest.permission.ACCESS_FINE_LOCATION},1);
                    }
                })
                .create()
                .show();
        }else{
            ActivityCompat.requestPermissions(DriverMapActivity.this, new String[]
{Manifest.permission.ACCESS_FINE_LOCATION},1);
        }
    }
}
```

```
    }  
  }  
}
```

În momentul în care are loc verificarea permisiunilor, orice schimbare de stare privind permisiunile va activa metoda de tip override, „*onRequestPermissionsResult()*”, care va verifica starea permisiunilor. Această metodă va folosi și obiectul de tip „*FusedLocationProviderClient*”, în momentul în care utilizatorul a oferit permisiunea accesării propriei locații, obiect ce va apela metoda „*requestLocationUpdates()*”, prin care se înaintea o cerere pentru actualizarea locațiilor. Metoda primește ca prim parametru un obiect de tip „*LocationRequest*”, ce conține setări privind modul în care se efectuează actualizarea locațiilor.

```
private LocationRequest mLocationRequest;  
@Override  
public void onMapReady(GoogleMap googleMap) {  
    mMap = googleMap;  
  
    mLocationRequest=new LocationRequest();  
    mLocationRequest.setInterval(1000);  
    mLocationRequest.setFastestInterval(1000);  
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);  
}
```

Crearea obiectului „*LocationRequest*” și setarea acestuia are loc în momentul în care harta s-a încărcat cu succes pe interfața dispozitivului, moment în care este apelată automat metoda „*onMapReady()*”. Setările acestui obiect constau în introducerea unui interval de timp în care să se facă actualizarea („*setInterval()*”), acesta fiind de 1000 de milisecunde, astfel având o influență directă asupra cantității de resurse utilizate de aplicație. De asemenea, metoda „*setFastestInterval()*”, reprezintă cea mai rapidă rată prin care aplicația va primi actualizările de locații, iar prin valoarea setată prin intermediul metodei „*setPriority()*”, se setează precizia locațiilor ce le va primi ca și actualizări, acest parametru influențând cantitatea de energie folosită de dispozitiv.

Totodata, metoda „*requestLocationUpdates()*”, mai primește drept parametru un obiect al clasei „*LocationCallback*”, care este responsabil de setarea noii locații pe hartă cât și în codul back-end, toate acestea realizându-se în loop, astfel efectuându-se constant actualizările de locație.

```
final int LOCATION_REQUEST_CODE =1;  
@Override  
public void onRequestPermissionsResult(int requestCode, @NonNull String[]  
permissions, @NonNull int[] grantResults){  
    super.onRequestPermissionsResult(requestCode,permissions,grantResults);  
    switch(requestCode){  
        case LOCATION_REQUEST_CODE:{  
            if (grantResults.length>0 && grantResults[0] ==  
PackageManager.PERMISSION_GRANTED){  
                if  
(ContextCompat.checkSelfPermission(this,Manifest.permission.ACCESS_FINE_LOCATION)=  
=PackageManager.PERMISSION_GRANTED){
```

```

        mFusedLocationClient.requestLocationUpdates(mLocationRequest,
mLocationCallback, Looper.myLooper());
        mMap.setMyLocationEnabled(true);
    }
    }else{
        Toast.makeText(getApplicationContext(),"Please provide the
permission", Toast.LENGTH_LONG).show();
    }
    break;
}}}

```

Obiectul de tip „*LocationCallback*”, ce este responsabil de actualizarea noilor locații generate de metoda „*requestLocationUpdates()*”, pe interfața aplicației:

```

LocationCallback mLocationCallback = new LocationCallback(){
    @Override
    public void onLocationResult(LocationResult locationResult) {
        for(Location location : locationResult.getLocations()){
            if(getApplicationContext()!=null){
                mLastLocation = location;

                LatLng latLng = new LatLng(location.getLatitude(),
location.getLongitude());
                mMap.moveCamera(CameraUpdateFactory.newLatLng(latLng));
                mMap.animateCamera(CameraUpdateFactory.zoomTo(13));
            }
        }
    }
};

```

În momentul în care utilizatorul se deconectează din aplicație, este importantă oprirea cererilor pentru actualizarea locațiilor. Acest lucru realizandu-se prin intermediul metodei „*removeLocationUpdates()*”.

```

mFusedLocationClient.removeLocationUpdates(mLocationCallback);

```

3.5. Salvarea și actualizarea coordonatelor geografice ale locațiilor în baza de date *Firebase*

În baza de date *Firebase* se vor stoca coordonatele geografice ale locațiilor, cât și destinațiile introduse de către utilizatori prin intermediul unor chei generate de către librăria *GeoFire*.

GeoFire este o librărie open-source²⁰ ce oferă posibilitatea de a stoca și a genera un set de chei pe baza locațiilor geografice. De asemenea, aceasta conține metode ce oferă o manipulare mai ușoară a coordonatelor geografice cât și posibilitatea interogării cheilor dintr-o anumită zonă geografică folosită în momentul în care are loc căutarea șoferilor disponibili pe o anumită rază. *GeoFire* folosește conceptul „*Firebase Realtime Database*” pentru stocare, ceea ce înseamnă că actualizarea rezultatelor are loc în timp real în momentul în care acestea suferă schimbări. Pentru a dispune de această librărie este necesară importarea unui dependency ce se numește

²⁰ Librăriile open-source – reprezintă librării pentru care codul sursă original este disponibil gratuit și poate fi redistribuit sau modificat conform cerinței utilizatorului.

„*firebase:geofire-android*” în fișierul „*build.gradle*” al aplicației.

Un obiect GeoFire este folosit pentru citirea și scrierea geolocațiilor în baza de date Firebase. Crearea unui obiect de tip GeoFire se realizează prin atașarea la obiect, a unei referințe către baza de date Firebase cu ajutorul căreia acesta va ști unde să scrie sau să citească datele.

```
DatabaseReference ref =
    FirebaseDatabase.getInstance().getReference("path/to/geofire");
GeoFire geoFire = new GeoFire(ref);
```

Actualizarea în baza de date a coordonatelor geografice are loc în codul din „*LocationCallback*”, disponibil în subcapitolul 3.4, cod ce oferă periodic cea mai recentă locație a utilizatorilor, locație ce va fi salvată și actualizată în baza de date de către un obiect de tip GeoFire.

```
String userId = FirebaseAuth.getInstance().getCurrentUser().getUid();
DatabaseReference refAvailable =
    FirebaseDatabase.getInstance().getReference("driversAvailable");
GeoFire geoFireAvailable = new GeoFire(refAvailable);
geoFireAvailable.setLocation(userId, new GeoLocation(location.getLatitude(),
    location.getLongitude()));
```

După cum se poate observa în codul de mai sus, are loc extragerea utilizatorului logat în aplicație, utilizator căruia i se va actualiza noile coordonate geografice în baza de date prin intermediul metodei „*setLocation()*”, apelată prin intermediul obiectului GeoFire.

În momentul în care utilizatorul se deconectează are loc ștergerea acestor coordonate geografice din baza de date, cu scopul evitării acumulării de date reziduale după rularea aplicației. Acest lucru fiind posibil cu ajutorul metodei „*removeLocation()*”, apelată de obiectul GeoFire.

```
geoFire.removeLocation(userId);
```

3.6. Plasarea unei cereri de transport și găsirea unui șofer disponibil

Plasarea cererilor de transport constă în crearea unei secțiuni în baza de date în care vor fi stocate cererile de transport ale pasagerilor ce așteaptă a li se asocia un șofer care să îndeplinească condițiile selectate (tipul serviciului). O astfel de cerere va conține coordonatele geografice a utilizatorului ce dorește a se transporta (Figura 3.3). O cerere de transport va fi inițializată în momentul în care un utilizator de tip pasager va apăsa pe butonul „*Call Uber*” de pe interfața aplicației (Figura 3.4), moment în care se va activa listener-ul butonului.

```
mRequest.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        int selectId = mRadioGroup.getCheckedRadioButtonId();
        final RadioButton radioButton = (RadioButton)findViewById(selectId);
        if(radioButton.getText() == null){
            return;
        }
    }
});
```



```

requestService = radioButton.getText().toString();
requestBol = true;
String userId = FirebaseAuth.getInstance().getCurrentUser().getUid();
DatabaseReference ref =
FirebaseDatabase.getInstance().getReference("customersRequest");
GeoFire geoFire = new GeoFire(ref);
geoFire.setLocation(userId, new
Geolocation(mLastLocation.getLatitude(),mLastLocation.getLongitude()));
pickupLocation= new LatLng(mLastLocation.getLatitude(),
mLastLocation.getLongitude());
pickupMarker = mMap.addMarker(new
MarkerOptions().position(pickupLocation).title("Pickup Here"));
mRequest.setText("Getting your driver...");
getClosestDriver();
}
});

```

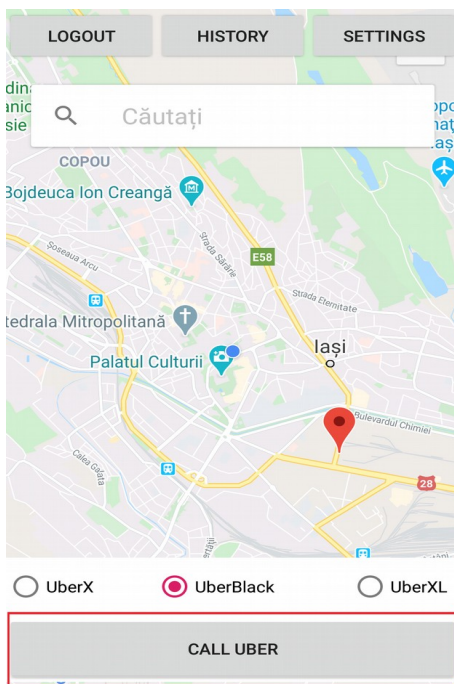


Figura 3.4: Interfața aplicației
aferentă utilizatorilor de tip pasager

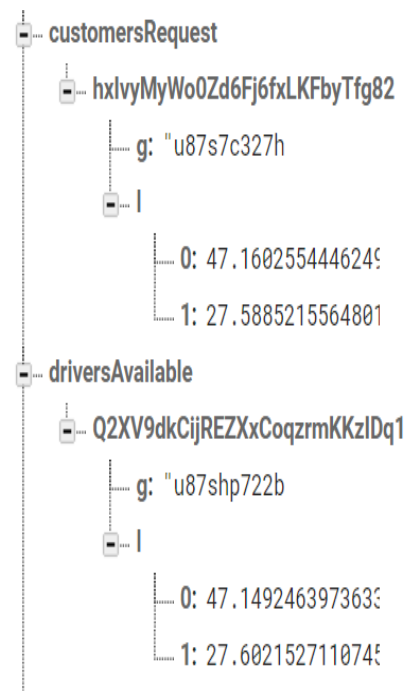


Figura 3.3: Baza de date ce conține
cererile pasagerilor și șoferii
disponibili

În codul de mai sus are loc identificarea pe baza id-ului propriu, pasagerul autentificat, id ce va fi stocat în variabila „userId”. Cu ajutorul acestui id și a unui obiect GeoFire ce primește ca referință în baza de date, secțiunea „customersRequest” (loc unde sunt stocate toate cererile de transport ce exista în acel moment, Figura 3.3), se vor stoca coordonatele geografice ale utilizatorilor ce au inițiat respectivele cereri.

Tot în codul de mai sus, are loc salvarea într-o variabilă numită „requestServices”, a tipului de serviciu selectat de către pasager prin selectarea unuia din butoanele de tip radio. Serviciu ce va fi folosit ca și criteriu în filtrarea șoferilor disponibili.

Pentru afișarea utilizatorului pe harta interfeței, se crează un obiect de tip „LatLng”, ce

primește ca și parametri, cele mai recente coordonate geografice disponibile, obiect prin intermediul căruia se va crea un marker pe hartă, aferent utilizatorului.

Apelarea metodei implementate „*getClosestDriver()*”, inițialiază procesul de căutare a șoferului. Metodă ce va conține o referință către baza de date, mai exact către șoferii disponibili în acel moment („*driversAvailable*”), aceștia având stocate coordonatele geografice (vezi Figura 3.3) ce vor fi interogate pentru a afla șoferii disponibili pe o anumită rază aplicată unui punct ce va reprezenta pasagerul ce a inițiat cererea de transport.

```
GeoQuery geoQuery;
private void getClosestDriver(){
    DatabaseReference driverLocation =
    FirebaseDatabase.getInstance().getReference().child("driversAvailable");
    GeoFire geoFire = new GeoFire(driverLocation);
    geoQuery = geoFire.queryAtLocation(new
    GeoLocation(pickupLocation.latitude,pickupLocation.longitude), radius);
    geoQuery.removeAllListeners();
}
```

Pe baza referinței „*driverLocation*”, se va crea un obiect de tip *GeoFire* cu ajutorul căruia se va crea un obiect *GeoQuery*, ce va interoga toate cheile dintr-o anumită zonă geografică. Pentru acest lucru este apelată metoda „*queryAtLocation()*”, prin intermediul obiectului *GeoFire* ce va primi ca parametri, cele mai recente coordonate ale utilizatorului („*pickupLocation*”), cât și raza aplicată respectivelor coordonate pentru care se va face scanarea de șoferi.

Ca și rezultate, acest *GeoQuery* va returna obiecte ce vor conține id-ul („*key*”) șoferilor găsiți pe respectiva rază, cât și coordonatele geografice ale acestora sub forma unor obiecte de tip *GeoLocation*. Astfel pentru procesarea acestor date returnate este necesară aplicarea unui listener („*GeoQueryEventListener*”) ce se va ocupa de evenimentele ce pot avea loc în momentul în care se face interogarea. Fiind posibilă producerea a cinci tipuri de evenimente:

- Key Entered – locația șoferului respectă criteriul interogării
- Key Exited – locația șoferului nu mai respecta criteriul interogării
- Key Moved – locația șoferului s-a schimbat dar în continuare respectă criteriul interogării
- Query Ready – toate datele din baza de date/server au fost încărcate
- Query Error – apariția unei erori în timpul interogării, ex: nerespectarea regulilor de securitate

Principala metodă ce a fost suprascrisă este reprezentată de „*onKeyEntered()*”, metodă ce răspunde de obiectele ce vor fi introduse în *GeoQuery*.

```
geoQuery.addGeoQueryEventListener(new GeoQueryEventListener() {
    @Override
    public void onKeyEntered(String key, GeoLocation location) {
        if(!driverFound && requestBol) {
            DatabaseReference mCustomerDatabase =
            FirebaseDatabase.getInstance().getReference().child("Users").child("Drivers").child(key);
            mCustomerDatabase.addListenerForSingleValueEvent(new ValueEventListener()
            {
                @Override
                public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                    if(dataSnapshot.exists() && dataSnapshot.getChildrenCount()>0){
```

```

        HashMap<String, Object> driverMap = (HashMap<String, Object>)
dataSnapshot.getValue();
        if(driverMap.get("service").equals(requestService)){
            driverFound = true;
            driverFoundId = dataSnapshot.getKey();
            DatabaseReference driverRef =
FirebaseDatabase.getInstance().getReference().child("Users").child("Drivers").child(
driverFoundId).child("customerRequest");
            String customerId =
FirebaseAuth.getInstance().getCurrentUser().getUid();
            HashMap map = new HashMap();
            map.put("customerRideId", customerId);
            map.put("destination", destination);
            map.put("destinationLat", destinationLatLng.latitude);
            map.put("destinationLng", destinationLatLng.longitude);
            driverRef.updateChildren(map);

            getDriverLocation();
            getDriverInfo();
            getHasRideEnded();
            mRequest.setText("Looking for Driver location..");
        }}
    @Override
    public void onCancelled(@NonNull DatabaseError databaseError) {
    }
    });} }

```

În codul de mai sus, în primul if al metodei „onKeyEntered()”, se testează dacă s-a găsit sau nu un șofer ce să corespundă cererii, astfel încât în cazul în care s-a găsit, să nu se mai efectueze căutarea unui alt șofer, deoarece o cerere de transport poate fi asociată doar unui singur șofer. De asemenea, se verifică și starea butonului de inițializare a unei cereri de transport (Figura 3.4) prin intermediul variabilei „requestBol”, testându-se dacă acesta a fost apăsat sau nu.

Dacă acest if este îndeplinit, ceea ce înseamnă că încă nu s-a găsit un șofer corespunzător, se va crea o referință sub denumirea „mCustomerDatabase”, la baza de date, către profilul șoferului ce tocmai a fost introdus în GeoQuery. Acestei referințe aplicându-i-se un listener de tip „SingleValueEvent”, deoarece este necesară execuția acestui listener doar o singură dată, pentru citirea unor informații statice ce corespund unui șofer.

Metoda „OnDataChange()”, generată de acest listener va avea un „dataSnapshot” ce va conține datele din locația dată ca referință în baza de date Firebase. Date ce vor fi introduse într-un map²¹ pentru o manipulare mai ușoară a acestora. Moment în care are loc verificarea tipului de serviciu prestat de către respectivul șofer, aceasta reprezentând și singura condiție pe care trebuie să o respecte un șofer pentru a fi asociat unei cereri.

Dacă această condiție nu este îndeplinită, se va trece la căutarea altui șofer, altfel se va crea o referință în baza de date pentru respectivul șofer, căruia i se va adăuga un nou câmp sub denumirea „customerRequest”, în care vor fi introduse datele privind id-ul pasagerului, numele

21 Map – reprezintă un obiect care mapează chei pe valori. Într-o astfel de structură nu pot exista chei duplicate. Fiecare cheie este mapată la exact o valoare.

destinației, cât și coordonatele aferente destinației, prin folosirea unui hashmap²². Acest lucru fiind necesar în partea de cod ce privește șoferul, pentru a realiza serviciul de transport.

De asemenea, în continuare are loc apelarea a trei metode implementate și anume:

- `GetDriverLocation()`: ce este responsabilă de preluarea coordonatelor geografice ale șoferului și crearea cu acestea a unui marker pe hartă, afișarea distanței dintre șofer și pasager, cât și afișarea unui mesaj în momentul în care șoferul se apropie de pasager.
- `GetDriverInfo()`: preia informațiile șoferului (nume, telefon, mașină, imagine de profil) și le afișează în aplicație pe interfața pasagerului.
- `TheRideIsOver()`: care are rolul de a reseta toate variabilele pe valorile inițiale și a pregăti interfața pentru o nouă rulare (ștergerea markerelor, eliminarea informațiilor privind șoferul de pe interfață).

În momentul în care obiectul `GeoQuery` își termină execuția și nu s-a găsit un șofer ce să corespundă cererii de transport, are loc incrementarea razei de acțiune și apelarea recursivă a metodei `„getClosestDriver()”`, în metoda `„onGeoQueryReady()”`, moment în care se va relua întregul proces de căutare, dar acum pentru noua rază.

```
@Override
public void onGeoQueryReady() {
    if(!driverFound){
        radius++;
        getClosestDriver();
    }
}
```

3.7. Salvarea și extragerea informațiilor din baza de date Firebase

În baza de date Firebase, informațiile sunt stocate în format JSON²³. Ca și metodă de salvare a datelor în baza de date Firebase este folosirea unui Map. Datele introduse de către utilizatori pe interfața aplicației sunt preluate de către elementele grafice XML. Informații ce sunt introduse mai apoi într-un Map cu ajutorul metodei `„put()”`. Map ce va fi trimis bazei de date prin intermediul unei referințe la o anumită locație din baza de date și a metodei `„updateChildren()”`, ce va primi ca parametru map-ul cu informații pe care le va scrie într-un anumit nod fără a suprascrie ceilalți fii ai nodului.

```
private DatabaseReference mCustomerDatabase;
private void saveUserInformation(){
    mName= mNameField.getText().toString();
    mPhone = mPhoneField.getText().toString();
    Map userInfo = new HashMap();
    userInfo.put("name", mName);
    userInfo.put("phone", mPhone);
}
```

22 HashMap – reprezintă una din implementările pentru Map. HashMap fiind neordonat și nesortat.

23 JSON(JavaScript Object Notation) – este un format de reprezentare și interschimb de date între aplicații informatice. Este un format text, inteligibil pentru oameni, utilizat pentru reprezentarea obiectelor și a altor structuri de date și este folosit în special pentru a transmite date structurate prin rețea, procesul purtând numele de serializare.

```

        userID = mAuth.getCurrentUser().getUid();
        mCustomerDatabase =
        FirebaseDatabase.getInstance().getReference().child("Users").child("Customers").
        child(userID);
        mCustomerDatabase.updateChildren(userInfo);
    }

```

De asemenea, introducerea datelor se mai poate realiza prin intermediul metodei „*setValue()*”, metodă ce suprascrive datele dintr-o anumită locație incluzând și nodurile de tip *child*.

```

User user = new User(name, email);
mDatabase.child("users").child(userID).setValue(user);

```

Pentru extragerea informațiilor din baza de date Firebase sunt folosite doua metode ce trebuie a fi atașate la o referință din baza de date:

- *addValueEventListener()*: execută metoda „*onDataChange()*” pentru citirea datelor ori de câte ori o informație sau un set de date, suferă schimbări.
- *addListenerForSingleValueEvent()*: execută o singură dată metoda „*onDataChange()*” pentru citirea datelor, după care se oprește din a se mai executa indiferent de starea datelor.

Metoda „*addValueEventListener()*” a fost folosită în aplicație pentru a extrage informațiile despre coordonatele geografice ale locației utilizatorilor, atât șoferi cât și pasageri, deoarece aceste coordonate sunt actualizate în baza de date din secunda în secunda, ceea ce este necesar ca citirea acestor date să se facă de asemenea concomitent cu aceste schimbări, pentru a beneficia de cele mai recente coordonate geografice pentru care se generează markere pe hartă.

Metoda „*addListenerForSingleValueEvent()*” este folosită în aplicație pentru citirea datelor statice, ce nu suferă modificări, precum numele utilizatorilor, numărul de telefon, mașina etc.

Ambele metode odată apelate vor genera două metode de tip override, mai exact „*onDataChange()*” și „*onCancelled()*”. Metoda ce se ocupă cu citirea datelor din baza de date este reprezentată de „*onDataChange()*” ce primește ca și parametru un obiect de tip „*DataSnapshot*”, ce va fi populat cu datele ce le găsește în locația din baza de date, locație dată prin referință.

Datele returnate de acest obiect „*DataSnapshot*”, pot fi introduse într-un map pentru o manipulare mai ușoară, sau pot fi folosite direct prin intermediul obiectului „*DataSnapshot*”.

```

mCustomerDatabase =
FirebaseDatabase.getInstance().getReference().child("Users").child("Customers").ch
ild(userID);
mCustomerDatabase.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
        if (dataSnapshot.exists() && dataSnapshot.getChildrenCount()>0){
            Map<String, Object> map = (Map<String, Object>)
dataSnapshot.getValue();
            if(map.get("name")!=null){

```

```

        mName = map.get("name").toString();
        mNameField.setText(mName);
    }
    if(map.get("phone")!=null){
        mPhone = map.get("phone").toString();
        mPhoneField.setText(mPhone);
    }
}
@Override
public void onCancelled(@NonNull DatabaseError databaseError) {
}
});

```

Respectivele informații fiind apoi afișate pe interfața utilizatorilor prin setarea elementelor grafice cu datele extrase din map sau „*DataSnapshot*” cu ajutorul metodei „*get()*”.

Metoda „*onCancelled()*” este activată în momentul în care citirea datelor este blocată. Acest lucru poate apărea în momentul în care utilizatorul nu are permisiunea de citire a datelor din respectiva bază de date sau dintr-o anumită locație din aceasta. Metoda va returna un obiect de tip „*DatabaseError*” ce va indica motivul pentru care nu s-a reușit citirea datelor.

3.8. Funcția de Place AutoComplete

Funcția de „*Place AutoComplete*” este un widget²⁴ reprezentat de o bară de căutare afișată pe interfața aplicației (vezi Figura 3.5) prin care utilizatorii, mai exact pasagerii, pot căuta o anumită locație în care doresc să se deplaseze, moment în care acest widget le va oferi sugestii de rezultate pe baza cuvintelor sau grupurilor de cuvinte sau litere pe care utilizatorii le-au introdus (vezi Figura 3.6).

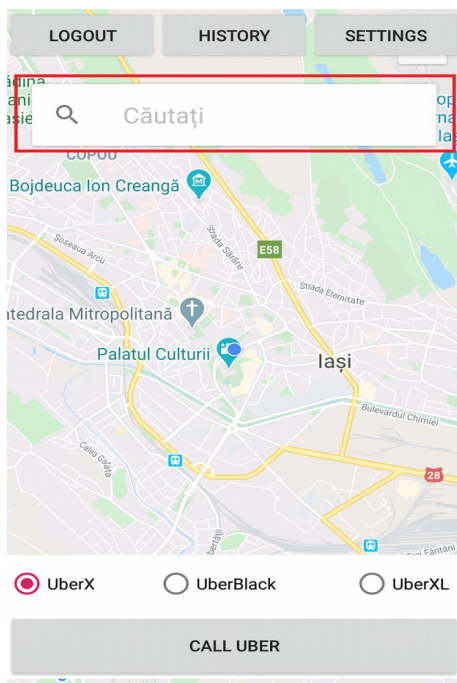


Figura 3.5: Interfața aplicației asociată pasagerului

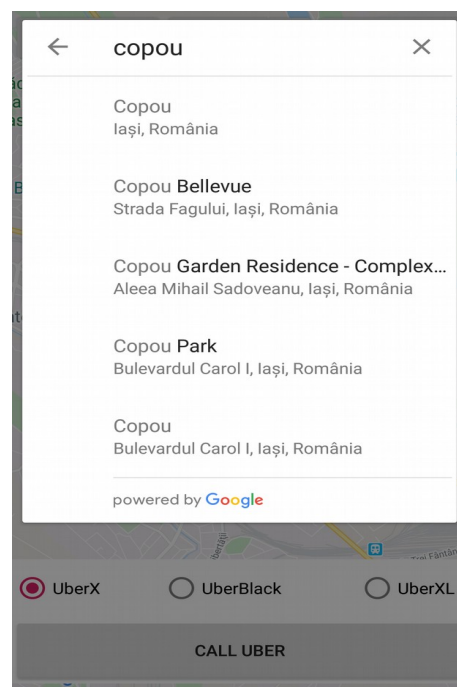


Figura 3.6: Place AutoComplete

²⁴ Widget – reprezintă o aplicație sau o componentă a interfeței, ce permite utilizatorului de a efectua o acțiune sau a accesa un serviciu.

Pentru a adăuga widget-ul pe interfața aplicației, este necesară adăugarea în fișierul XML aferent interfeței pasagerilor, un fragment de cod ce este disponibil în documentația oferită de cei de la Google pentru implementarea Place AutoComplete-ului.

```
<fragment android:id="@+id/autocomplete_fragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:name="com.google.android.libraries.places.widget.AutoCompleteSupportFragment"/>
```

Codul ce va implementa funcționalitatea widget-ului este adăugat în metoda „onCreate()” al interfeței pasagerului, astfel încât în momentul în care interfața este încărcată, va avea loc și crearea acestui widget.

```
AutocompleteSupportFragment autocompleteFragment = (AutocompleteSupportFragment)
    getFragmentManager().findFragmentById(R.id.autocomplete_fragment);

autocompleteFragment.setPlaceFields(Arrays.asList(Place.Field.ID,
    Place.Field.NAME, Place.Field.LAT_LNG));
autocompleteFragment.setOnPlaceSelectedListener(new PlaceSelectionListener() {
    @Override
    public void onPlaceSelected(Place place) {
        // TODO: Get info about the selected place.
        destination = place.getName().toString();
        destinationLatLng = place.getLatLng();
    }
    @Override
    public void onError(Status status) {
        // TODO: Handle the error.
    }
});
```

În codul de mai sus are loc inițializarea unui obiect de tip „AutocompleteSupportFragment”, obiect ce este inițializat cu elementul XML introdus în fișierul XML al interfeței, astfel făcându-se conexiunea dintre front-end²⁵ și back-end²⁶. Setarea câmpurilor ce se doresc a fi returnate se realizează cu ajutorul metodei „setPlaceFields()”, ce primește ca parametru, tipul formatului în care vor fi returnate datele (listă), listă ce poate conține date returnate cu ajutorul metodelor *ID*, *NAME*, *LAT_LNG*, *getWebsiteUri()*, *getOpeningHour()*, *getPhoneNumber()*, *getRating()* etc.

În cazul de față, datele returnate fiind reprezentate de id-ul locației, numele locației și de un obiect de tip „LatLng”, ce va conține coordonatele geografice ale destinației selectate de utilizator, coordonate folosite pentru a afișa pe harta șoferului, locația selectată de pasager, cât și crearea unei rute până la respectiva destinație.

După setarea câmpurilor ce vor fi returnate, obiectului „AutocompleteSupportFragment”, îi este aplicat un listener ce va genera două metode de tip

25 Front-end – reprezintă acea parte a unui site, program sau aplicație pe care utilizatorii o pot vedea și interacționa (meniuri, butoane, tranziții, formulare de contact etc.). Acesta are două părți: design-ul (partea creativă) și dezvoltarea interfeței (partea de cod sau implementare HTML, CSS).

26 Back-end – partea de code a unei aplicații sau a unui program, ce realizează managementul conținutului. Este responsabilă de gestionarea tuturor informațiilor și a structurilor pe care utilizatorul obișnuit nu le poate vedea. De obicei constă în implementarea a trei părți: un server, o aplicație de interfață și o baza de date.

override. Metoda „onPlaceSelected”, ce va fi activată în momentul în care utilizatorul selectează una din locațiile ce le primește ca și sugestii. Și metoda „onError()”, ce va fi activată în momentul în care se va genera o anumită eroare.

După ce widget-ul cât și funcționalitatea lui au fost create, este necesară activarea din consola Google a unui API, mai exact „Places API”, ce va oferi acces la baza lor de date ce conțin locațiile ce vor fi afișate pe interfața widget-ului Place AutoComplete. Acest serviciu fiind accesat de către aplicație prin intermediul unei chei API (vezi subcapitolul 3.3. Setarea API-ului Google Maps pentru crearea și setarea unei chei API).

De asemenea, acest serviciu oferit de „Places API” este limitat la un număr de cereri, altfel în momentul în care aplicația ar fi publicată în app stores, numărul de cereri poate depăși această limită. Pentru a dispune de un număr nelimitat de cereri este necesară crearea unui cont de plată în consola celor de la Google, prin introducerea unui cont bancar, astfel încât în momentul în care are loc depășirea limitei standard de cereri oferite gratuit de Google, se va realiza un cost per număr cereri.

3.9. Crearea rutelor și afișarea acestora pe interfața aplicației

Această funcționalitate este oferită utilizatorilor de tip șofer și este folosită în aplicație în două cazuri. În momentul în care un șofer efectuează un serviciu de transport, acestuia îi va fi oferită pe interfața aplicației o rută către pasagerul pe care urmează să-l preia. Al doilea caz este reprezentat de momentul în care șoferul a preluat pasagerul, aplicația oferindu-i o rută către destinația selectată de pasager, astfel șoferul mereu va avea o ghidare exactă pe hartă.

Pentru implementarea acestui serviciu este necesară adăugarea unui dependency și mai exact „com.github.jd-alexander:library” în fișierul „build.gradle”.

Pentru a realiza afișarea rutelor trebuie ca în clasa asociată interfeței careia dorim să-i introducem această funcționalitate, să implementăm interfața „RoutingListener”, interfață ce va conține metodele abstracte „onRoutingFailure()”, „onRoutingStart()”, „onRoutingSuccess()” și „onRoutingCancelled()”, metode ce se vor ocupa de starea rutelor.

```
public class DriverMapActivity extends FragmentActivity implements
    OnMapReadyCallback, RoutingListener
```

Codul responsabil de crearea rutelor:

```
private void getRouteToMarker(LatLng pickupLatLng){
    Routing routing = new Routing.Builder()
        .key(getString(R.string.google_maps_key))
        .travelMode(AbstractRouting.TravelMode.DRIVING)
        .withListener(this)
        .alternativeRoutes(false)
        .waypoints(new LatLng(mLastLocation.getLatitude(),
            mLastLocation.getLongitude()), pickupLatLng)
        .build();
    routing.execute();
}
```

Această metoda „getRouteToMarker()”, este apelată în momentul în care un șofer trebuie să efectueze o cursă de transport. Aceasta primește ca și parametru un obiect de tip „LatLng”, ce conține coordonatele geografice ale utilizatorului ce a inițiat cererea de transport. În interiorul ei

are loc loc crearea unui obiect de tip „*Routing*” și setarea acestuia. Pentru crearea rutelor este necesară crearea unei cereri către server-ul Google pentru folosirea serviciului oferit de API-ul „*Direction API*”, ce va oferi date privind direcțiile, necesare creării rutelor. Această cerere efectuându-se prin intermediul unei chei API, prin intermediul căreia aplicația va fi recunoscută de către Google și prin care va solicita folosirea acestui serviciu (vezi subcapitolul 3.3. Setarea API-ului Google Maps). De asemenea după cum se poate observa în codul de mai sus, se selectează modul de parcurgere al respectivei rute, care dispune de mai multe opțiuni cum ar fi: *DRIVING*, *BIKING*, *TRANSIT* și *WALKING*. Prin câmpul „*alternativeRoutes()*”, aplicația poate dispune de afișarea mai multor alternative de rute, prin setarea acestuia pe „*true*”. Iar un ultim lucru ce mai necesită a fi setat este reprezentat de punctele între care se va crea ruta, în codul de mai sus primind ca și parametri două obiecte de tip „*LatLng*”, ce vor conține coordonatele șoferului, respectiv ale pasagerului.

În momentul în care obiectul „*Routing*” este executat prin metoda „*execute()*”, are loc crearea unui număr de rute, care poate fi și zero, rute ce vor fi preluate de către metoda „*onRoutingSuccess()*”.

Dacă crearea rutelor a avut loc cu succes, metoda de tip override „*onRoutingSuccess()*” este activată, metodă ce răspunde de desenarea propriu-zisă a rutelor pe interfața utilizatorilor.

```
private List<Polyline> polylines;
private static final int[] COLORS = new int[]
{R.color.primary_dark_material_light};
@Override
public void onRoutingSuccess(ArrayList<Route> route, int shortestRouteIndex) {
    if(polylines.size()>0) {
        for (Polyline poly : polylines) {
            poly.remove();
        }
    }
    polylines = new ArrayList<>();
    for (int i = 0; i < route.size(); i++) {
        int colorIndex = i % COLORS.length;
        PolylineOptions polyOptions = new PolylineOptions();
        polyOptions.color(getResources().getColor(COLORS[colorIndex]));
        polyOptions.width(10 + i * 3);
        polyOptions.addAll(route.get(i).getPoints());
        Polyline polyline = mMap.addPolyline(polyOptions);
        polylines.add(polyline);
    }
}
```

Această metodă primește ca și parametri un vector ce va conține rutele generate de metoda „*getRouteToMarker()*”, cât și id-ul rutei cele mai optime. Aceste rute fiind mai apoi preluate printr-un for cu ajutorul cărora se vor crea obiecte de tip „*PolylineOptions*”, fiindu-le asociate o culoare și o grosime și de asemenea punctele cardinale ce vor crea respectiva rută.

Aceste obiecte fiind folosite la rândul lor în crearea unor obiecte de tip *Polyline*, reprezentând poliliniile ce vor fi afișate pe harta interfeței utilizatorilor și stocate într-o listă de polilinii, listă ce este resetată la fiecare rulare al acestei metode „*onRoutingSuccess()*”.

Ștergerea rutelor de pe hartă se realizează prin apelarea metodei implementate „*erasePolylines()*”, ce va reseta variabila globală de tip listă cu ajutorul metodelor „*remove()*” și „*clear()*”.


```
private void erasePolylines(){
    for(Polyline line : polylines){
        line.remove();
    }
    polylines.clear();
}
```

3.10. Crearea istoricului curselor efectuate și popularea acestuia

Pentru a crea o interfață în care va fi afișat istoricul curselor efectuate de către utilizatori este necesară crearea unui element XML de tip „*RecyclerView*” și setarea poziției acestuia pe interfața aplicației (Figura 3.7). Acest lucru se efectuează în fișierul XML aferent interfeței unde se dorește a fi implementat.

```
<androidx.recyclerview.widget.RecyclerView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/historyRecyclerView"
    android:scrollbars="vertical">
</androidx.recyclerview.widget.RecyclerView>
```

Este necesară crearea unui layout ce va reprezenta forma înregistrării (item list) ce va fi afișată în istoric. Acest layout fiind reprezentat de elemente XML pentru afișarea textelor, imaginilor, checkpoint-urilor etc, în funcție de informațiile ce se doresc a fi afișate (Figura 3.8).

Fiecărei înregistrări din istoric îi este asociat un obiect care va conține informațiile necesare populării layout-ului aferent respectivei înregistrări. Prin urmare trebuie creată o clasă ce va conține variabilele necesare reținerii informațiilor ce trebuie introduse în layout-ul precizat mai sus (Figura 3.8).

Pentru introducerea și afișarea elementelor din istoric este necesară crearea unui așa numit „*Adapter*”. Acesta are rolul de a converti un obiect de pe o anumită poziție a listei (ce conține obiecte cu informațiile pentru toate înregistrările ce trebuie afișate în istoric), într-o înregistrare afișată în istoric prin intermediul layout-ului ce conține forma de afișare a înregistrării. Cu alte cuvinte, acesta realizează transportul datelor extrase din baza de date, ce le primește sub forma unui vector sau listă prin intermediul constructorului intern, și pe care le afișează în istoric prin intermediul înregistrărilor. De asemenea adapter-ul mai are rolul de a produce un număr optimizat de layout-uri pentru afișarea înregistrărilor, atât cât permite display-ul dispozitivului utilizatorului. Mai exact în momentul în care un device poate afișa doar un număr n de înregistrări din cauza dimensiunilor acestuia, acest adapter va crea doar un număr m de layout-uri care va respecta condiția: $m > n$, $m - n < 5$ (nu exista o valoare standard, acesta valoare („cinci”) a fost pusă pentru a înțelege conceptul). Diferența dintre m și n nefiind una foarte mare. Acest fapt fiind principala diferență dintre „*ListView*” și „*RecyclerView*” și principalul motiv pentru care „*RecyclerView*” este apreciat. Spre deosebire de „*ListView*”, acesta încarcă doar înregistrările ce le poate afișa dispozitivul, celelalte înregistrări ce vor fi afișate în momentul în care se efectuează scroll-ul, vor fi reciclate (vezi Figura 3.9). În acest mod se evită supraîncărcarea memoriei ce s-ar produce în cazul în care pentru fiecare înregistrare din listă ar fi trebuit crearea unui layout propriu.

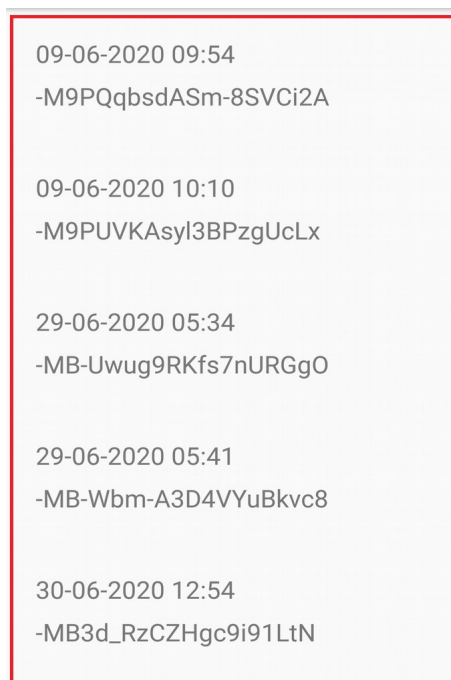


Figura 3.7: Recycle View

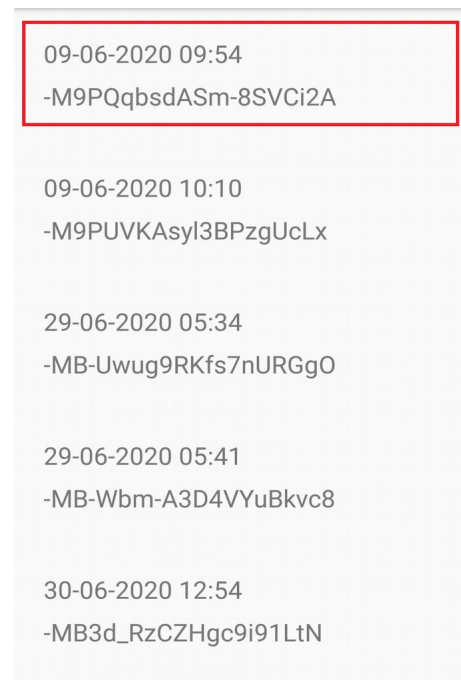


Figura 3.8: Forma unei înregistrări din istoric

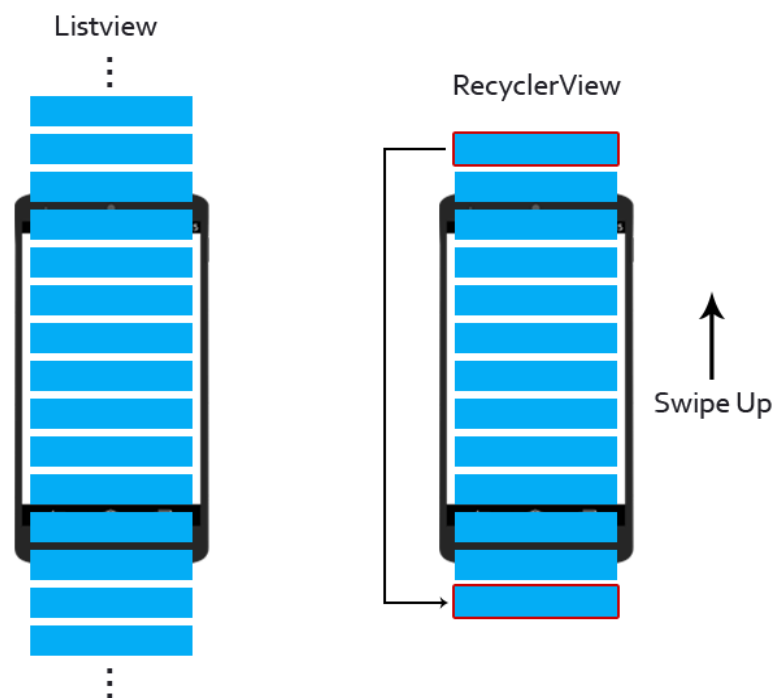


Figura 3.9: Diferența dintre RecyclerView și ListView [12]

Implementarea clasei „*Adapter*”, necesită existența unui obiect de tip „*ViewHolder*” care are ca scop crearea referințelor către toate view-urile aferente layout-ului înregistrării. Pentru crearea acestor obiecte s-a implementat clasa „*HistoryViewHolders*”, clasă ce va moșteni clasa „*ViewHolder*” din librăria „*RecyclerView*”, căreia îi este necesară crearea unui constructor în care

se va realiza referințele către elementelor din fișierul XML al înregistrării (în cazul de față, id-ul cursei și timpul).

```
public class HistoryViewHolders extends RecyclerView.ViewHolder {
    public TextView rideId;
    public TextView time;
    public HistoryViewHolders(@NonNull View itemView) {
        super(itemView);

        rideId = (TextView)itemView.findViewById(R.id.rideId);
        time = (TextView)itemView.findViewById(R.id.time);
    }
}
```

Clasa adapter-ului creat, va moșteni clasa „*Adapter*”, din librăria „*RecyclerView*”, clasă ce va folosi la rândul ei, clasa „*HistoryViewHolders*”, creată mai sus. Pentru această clasă trebuie implementate trei metode override: „*onCreateViewHolder()*”, „*onBindViewHolder()*” și „*getItemCount()*”.

```
public class HistoryAdapter extends RecyclerView.Adapter<HistoryViewHolders> {
    private List<HistoryObject> itemList;
    private Context context;
    public HistoryAdapter(List<HistoryObject> itemList, Context context){
        this.itemList= itemList;
        this.context = context;
    }
    @NonNull
    @Override
    public HistoryViewHolders onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
        View layoutView =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_history, null,
false);
        HistoryViewHolders rcv = new HistoryViewHolders(layoutView);
        return rcv;
    }
    @Override
    public void onBindViewHolder(@NonNull HistoryViewHolders holder, int position)
{
        holder.rideId.setText(itemList.get(position).getRideId());
        holder.time.setText(itemList.get(position).getTime());
    }
    @Override
    public int getItemCount() {
        return itemList.size();
    }
}
```

După cum se poate observa în codul de mai sus, se declară o listă cu obiectele ce vor conține informații privind înregistrările, listă ce va fi parsată prin intermediul constructorului.

Metoda „*onCreateViewHolders()*”, are rolul de a încărca layout-ul înregistrării din fișierul XML prin metoda „*inflate()*” și de a crea și a returna un obiect de tip „*ViewHolder*” ce

va face referințe la elementele XML din layout încărcat.

Metoda „*onBindViewHolder()*”, are ca și scop popularea cu date a layout-ului încărcat, prin intermediul „*viewholder*”-ului creat anterior și a datelor ce sunt extrase din lista de obiecte, obiecte ce sunt extrase prin intermediul parametrului „*position*”.

Cea din urmă metodă, „*getItemCount()*”, returnează numărul total de obiecte din lista de obiecte, listă ce este dată ca parametru în momentul în care se apelează adapter-ul.

După crearea tuturor acestor clase și metode, este necesară instanțierea și apelarea acestora în fișierul java ce va implementa interfața ce va conține istoricul.

```
public class HistoryActivity extends AppCompatActivity {
    private RecyclerView mHistoryRecyclerView;
    private RecyclerView.Adapter mHistoryAdapter;
    private RecyclerView.LayoutManager mHistoryLayoutManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_history);

        mHistoryRecyclerView = findViewById(R.id.historyRecyclerView);
        mHistoryRecyclerView.setHasFixedSize(true);

        mHistoryLayoutManager = new LinearLayoutManager(HistoryActivity.this);
        mHistoryRecyclerView.setLayoutManager(mHistoryLayoutManager);
        mHistoryAdapter = new HistoryAdapter(getDataSetHistory(),
        HistoryActivity.this);
        mHistoryRecyclerView.setAdapter(mHistoryAdapter);
    }
}
```

Astfel în codul de mai sus are loc crearea unei referințe de tip „*RecyclerView*” către elementul de același tip din fișierul XML, acesta reprezentând locul unde vor fi introduse înregistrările privind cursele efectuate de către utilizatori. De asemenea, se creează un obiect de tip adapter ce va popula „*recycleview*”-ul cu înregistrări, cât și a unui obiect de tip „*LayoutManager*”, ce este responsabil de măsurarea și poziționarea elementelor vizuale cât și pentru determinarea politicii de reciclare al acestor elemente în momentul în care nu mai sunt vizibile pentru utilizator. Cele două obiecte din urmă (*LayoutManager* și *Adapter*) fiind aplicate „*recycleview*”-ului prin intermediul metodele „*setLayoutManager()*” și „*setAdapter()*”.

Unul din parametri primiti de adapter în momentul instanțierii acestuia este reprezentată de o metodă creată de noi, „*getDataSetHistory()*”, ce va extrage informațiile din baza de date, ce le va returna sub forma unei liste de obiecte.

Pentru o performanță mai bună al „*recycleview*”-ului, este necesară setarea acestuia de a avea o dimensiune exactă, indiferent de numărul de înregistrări pe care va trebui să le afișeze. Pentru acest lucru are loc apelarea metodei „*setHasFixedSize()*”, ce va primi ca parametru valoarea booleană *true*.

Popularea istoricului cu date se realizează prin crearea unei liste cu obiecte ce conțin câmpurile unei înregistrări, în cazul de față, id-ul cursei și timpul când aceasta a fost efectuată, informații ce sunt extrase din baza de date. Adapter-ul fiind informat de încărcarea obiectelor în listă prin intermediul metodei „*notifyDataSetChanged()*”, ca mai apoi adapter-ul să realizeze încărcarea respectivelor date.

```
mHistoryAdapter.notifyDataSetChanged();
```

3.11. Implementarea sistemului de plată PayPal

3.11.1. Setarea contului Paypal

Pentru a implementa sistemul de plată PayPal, este necesară crearea unui cont de tip dezvoltator, accesând site-ul PayPal. Cont prin intermediul căruia se va putea crea un proiect din meniul „DashBoard”. Un lucru important de menționat este tipul de proiect selectat în momentul creării acestuia, site-ul oferind două opțiuni: „Sandbox” și „Live”. Tipul de proiect „Sandbox” are ca și scop testarea aplicației pe partea de tranzacții financiare prin intermediul unor conturi PayPal virtuale ce pot fi create sau generate de către site-ul PayPal, tranzacțiile efectuându-se cu bani virtuali din conturile setate anterior, astfel evitând efectuarea de tranzacții reale cu sume de bani reale. Tipul de proiect „Live” este folosit în momentul în care dezvoltatorul dorește să își publice aplicația într-un app store, cu scopul obținerii de bani reali, în care tranzacțiile cât și conturile sunt reale. Aplicației de față a fost asociată tipului de proiect, „Sandbox”.

Setarea proiectului constă în asocierea acestuia a unui nume și selectarea unui cont virtual PayPal de tip „Business”, oferit de către site-ul PayPal sau ce poate fi creat și personalizat din meniul „Accounts”, butonul „Create Account”, unde se pot crea două tipuri de conturi: „Business” și „Personal”. Contul „business” reprezentând locul unde se vor realiza tranzacțiile financiare realizate de către pasageri, ca mai apoi șoferii prin intermediul unor cereri http, își vor expune dorința de a le fi virată suma aferentă cursei efectuate de către aceștia.

Proiectul creat și setat în PayPal, îi va fi asociat un id unic, id prin intermediul căruia se va realiza legătura dintre proiectul PayPal și aplicație.

3.11.2. Implementarea sistemului de plată PayPal în aplicație

Pentru implementarea sistemului de plată PayPal, în aplicație este necesară importarea dependency-ului celor de la PayPal, prin intermediul căruia vom dispune de serviciile acestora. Acest lucru efectuându-se în fișierul „build.gradle” în care se va importa ultima versiune de SDK PayPal disponibilă.

```
implementation 'com.paypal.sdk:paypal-android-sdk:2.15.3'
```

Este necesară crearea unui obiect de tip „PayPalConfiguration”, ce are rolul de a configura tranzacția PayPal, setări ce vor fi făcute în concordanță cu cele ale proiectului creat pe site-ul PayPal. Aceste setări constau prin setarea id-ului generat de proiectul creat pe site-ul Paypal, astfel făcându-se legătura între respectivul proiect și aplicație, cât și setarea tipului de tranzacție, unde sunt oferite trei opțiuni:

- environment_sandbox – pentru stadiul de testare al aplicației, tranzacțiile având opțiunea de a fi vizualizate din meniul „dashboard” al site-ului PayPal.
- environment_no_network – folosit pentru tranzacții false, nereale, tranzacțiile neputând fi vizualizate deoarece tranzacțiile nu vor interacționa cu serverele PayPal.
- environment_production – în momentul în care aplicația este lansată într-un magazin de aplicații, tranzacțiile financiare fiind reale.

```
private static PayPalConfiguration config = new PayPalConfiguration()
    .environment(PayPalConfiguration.ENVIRONMENT_SANDBOX)
    .clientId(PayPalConfig.PAYPAL_CLIENT_ID);
```

În codul de mai sus, „*PayPalConfig*”, reprezentând o clasă creată de noi, în care se află variabila ce conține id-ul proiectului creat pe site-ul PayPal.

Informațiile privind plata sunt introduse într-un obiect „*PayPalPayment*”, ce va primi ca parametri suma ce urmează a fi plătită, cât și doi parametri de tip string prin care se specifică moneda în care se efectuează plata cât și serviciul pentru care are loc plata.

De asemenea, are loc crearea și inițializarea unui nou intent, intent ce are scop redirectionarea către o noua activitate (interfață). Acesta va redirectiona utilizatorul către o interfață din exteriorul aplicației (ce nu a fost implementată în aplicație), numită „*PaymentActivity*”, generată de SDK-ul PayPal, ce va reprezenta interfața PayPal prin care utilizatorii vor efectua plata (Figura 3.11).

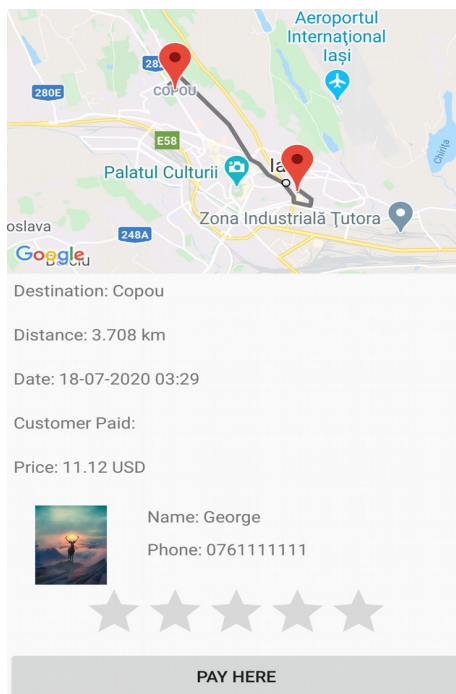


Figura 3.10: Inițializarea plății PayPal

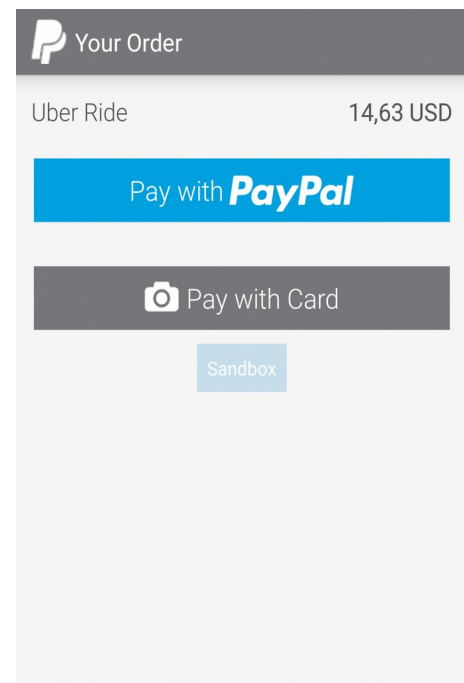


Figura 3.11: Payment Activity

Pentru setarea acestei activități este necesară transmiterea configurărilor privind plata și tranzacția, configurări ce au fost enumerate mai sus. Acest lucru fiind posibil cu ajutorul metodei „*putExtra()*”, ce are rolul de a transmite informațiile de la un intent la altul.

```
private int PAYPAL_REQUEST_CODE= 1;
private void paypalPayment() {
    PayPalPayment payment = new PayPalPayment(new BigDecimal(ridePrice), "USD",
    "Uber Ride", PayPalPayment.PAYMENT_INTENT_SALE);

    Intent intent = new Intent(this, PaymentActivity.class);

    intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    intent.putExtra(PaymentActivity.EXTRA_PAYMENT, payment);
```



```
startActivityResult(intent, PAYPAL_REQUEST_CODE);
}
```

Apelarea intent-ului este realizat prin intermediul metodei „*startActivityResult()*”, ce primește ca parametri intent-ul ce se dorește a fi încărcat, cât și un cod prin care se identifică ce intent va returna rezultatele. Cod ce ajută în momentul în care aplicația lansează mai multe intent-uri, prin intermediul codului, se va identifica intent-ul ce a returnat anumite rezultate.

Datorită încărcării intent-ului (activității) prin intermediul metodei „*startActivityResult()*”, se pot verifica rezultatele primite de către acest intent, rezultate ce vor conține informații privind plata, mai exact dacă aceasta s-a efectuat cu succes sau a intervenit o eroare în procesarea sa. Pentru acest lucru s-a folosit metoda de tip override, „*onActivityResult()*”, ce este activată de către metoda „*startActivityResult()*”, în momentul în care aceasta returnează rezultate. Această metodă „*onActivityResult()*” are trei parametri ce ofera informațiile următoare:

- requestCode – identifică ce intent a returnat rezultatele. Acesta este definit de dezvoltator în momentul în care un intent este apelat prin metoda „*startActivityResult()*” și ajută în identificarea intent-urilor, în cazul în care avem apelate mai multe intent-uri prin aceeași metodă. Metoda „*onActivityResult()*”, gestionează toate răspunsurile primite de la intent-uri, astfel fiecărui intent îi este asociat un cod unic prin intermediul căruia va rula o anumită bucată de cod din metodă.
- resultCode – informează asupra statusului rulării intent-ului, dacă aceasta s-a efectuat cu succes sau au intervenit erori.
- data – conține datele ce sunt returnate de către intent.

Pentru preluarea informațiilor trimise ca și răspuns de către intent, s-a creat un obiect de tip „*PaymentConfirmation*” în care se va adauga prin metoda „*getParcelableExtra()*”, datele ce au fost puse drept răspuns de către activitatea „*PaymentActivity*”, prin intermediul metodei „*putExtra()*”. Obiect ce va fi transformat într-un obiect de tip „*JSONObject*”, din care se testează statusul răspunsului, ce informează dacă plata a fost aprobată sau nu, iar în funcție de acesta se va afișa un mesaj corespunzător.

În momentul în care plata s-a efectuat cu succes, se dezactivează butonul prin intermediul căruia se face plata, astfel se evită posibilitatea ca un utilizator să plătească din greșală de mai multe ori aceeași cursă de transport.

De asemenea, este necesară inițializarea serviciilor PayPal în momentul în care activitatea ce deține aceste servicii este încărcată pe interfața dispozitivului, astfel încât aplicația să fie pregătită pentru încărcarea intent-ului PayPal. Pentru acest lucru fiind introdus codul de mai jos în metoda „*onCreate()*” al activității aferente serviciului de plată.

```
Intent intent = new Intent(this, PayPalService.class);
intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
startService(intent);
```

Serviciu ce trebuie oprit în momentul în care activitatea ce a inițiat acest serviciu nu mai este în rulare, prin linia de cod ce se va plasa în metoda „*onDestroy()*” al activității.

```
stopService(new Intent(this, PayPalService.class));
```


3.12. Implementarea cereri http al șoferului către PayPal

3.12.1. Implementarea PayPal – setarea și configurarea host-ului și a funcțiilor

Pentru implementarea sistemului de plată prin care utilizatorii de tip șofer, vor putea primi suma de bani aferentă cursei de transport efectuate, s-a realizat prin intermediul tool-ului „*firebase-functions*”, oferit de către cei de la Firebase. Acest lucru se va realiza prin crearea unei cereri http către server-ul Firebase prin care șoferii își vor primi banii. Firebase permite rularea codului pe care un dezvoltator îl crează, pe partea lor de server, acest lucru asigurând securitatea aplicației privind plățile.

Pentru a folosi funcțiile din Firebase, trebuie instalat tool-ul pentru Firebase command line. Pentru acest lucru trebuie accesat consola Firebase, în meniul functions, unde va fi afișată o comandă ce va trebui rulată în *node.js* deoarece *firebase-functions* folosește în back-end limbajul JavaScript. Este recomandată crearea unui folder special unde se vor importa scripturile JavaScript. După care se setează calea *node.js*-ului, către fișierul creat unde se va rula comanda afișată în Firebase:

```
npm install -g firebase-tools
```

După care se va inițializa host-ul asupra folder-ului ce va conține script-urile JavaScript, prin următoarea comandă:

```
firebase init hosting
```

Moment ce va trebui să selectăm numele proiectului din Firebase pe care dorim să îl asociem cu folderul menționat mai sus.

Importarea funcțiilor și al SDK-urilor, ce vor conține script-urile JavaScript se realizează prin comanda ce va fi introdusă în *node.js*:

```
firebase init functions
```

În urma rulării acestei comenzi, se va alege limbajul de programare prin care vom realiza scrierea funcțiilor necesare pentru realizarea plății șoferilor, server-ul oferind opțiunile de *JavaScript* și *TypeScript*. De asemenea, este necesară acceptarea instalării dependențelor Firebase.

În urma rularii tuturor acestor comenzi, în folder-ul creat la început se va găsi un fișier sub denumirea „*functions*”. Pentru ultima etapă a configurării tool-ului *firebase-functions* este necesară schimbarea cailor din *node.js*, în folderul *functions*, prin comanda „*cd functions*”, fișier în care se va realiza importarea SDK-ului pentru serviciile Paypal. Acest lucru fiind posibil prin comandă:

```
npm install paypal-rest-sdk
```

În final se realizează trimiterea tuturor acestor configurări, import-uri și funcții create, către serverul Firebase, prin comandă:

```
firebase deploy
```

Un lucru important de reținut este acela de a nu se șterge folder-ul creat mai sus, deoarece odată șters, se pierde accesul la toate funcțiile create cât și a configurațiilor făcute, iar Firebase nu

dispune de nici o opțiune de a le recupera, fiind necesară implementarea acestora de la zero.

3.12.2. Implementarea cererii http în JavaScript

Implementarea cererii http se va realiza în folderul *functions*, fișierul *index.js*. Acest fișier, *index.js*, va reprezenta locul unde se vor crea funcțiile JavaScript ce vor fi încărcate pe server-ul Firebase. În acest fișier se crează o funcție, în cazul de față, aceasta numindu-se „*payout*”, ce va realiza plata propriu-zisă a șoferilor, dar pentru ca această funcție să fie apelată prin intermediul link-ului de host, va trebui introdus în fișierul *firebase.json*, redirecționarea către această funcție. Acest lucru fiind posibil prin modificarea atribut-ului *rewrite*.

```
{
  "hosting": {
    "public": "public",
    "ignore": ["firebase.json", "**/.*", "**/node_modules/**"],
    "rewrite": [{ "source": "/payout", "function": "payout" } ]
  }
}
```

Este necesară crearea a trei variabile cu ajutorul cărora vom inițializa aplicația și vom dispune de serviciile necesare creării plății șoferilor.

```
const functions = require('firebase-functions');
const paypal = require('paypal-rest-sdk');
const admin = require('firebase-admin');
admin.initializeApp();
```

În codul de mai sus, variabila *functions* va încărca modulul cu funcțiile firebase, variabila *paypal* va încărca SDK-ul pentru serviciul PayPal, iar în variabila *admin* se va încărca SDK-ul ce asigură interacțiunea cu server-ul Firebase și totodată oferă un mediu privilegiat prin care putem executa anumite acțiuni precum citirea și scrierea în baza de date cu drepturi absolute etc.

Este necesară configurarea variabilei *paypal*, prin setarea câmpului „*mode*”, cu tipul aferent proiectului creat pe site-ul PayPal și anume „*sandbox*” în cazul de față, împreună cu setarea câmpurilor „*client_id*” și „*client_secret*”, id-uri unice ce sunt generate și asignare fiecarui proiect creat pe site-ul PayPal.

```
paypal.configure({
  mode: 'sandbox',
  client_id: functions.config().paypal.client_id,
  client_secret: functions.config().paypal.client_secret
})
```

Odată configurat serviciul PayPal, trebuie încărcat pe server-ul Firebase variabilele „*paypal.client_id*” și „*paypal.client_secret*”, pentru a conecta server-ul Firebase cu proiectul ce se afla pe site-ul PayPal. Acest lucru se realizează în terminalul *node.js*, care va avea ca și cale, fișierul ce conține configurările și funcțiile Firebase. Unde se vor executa una câte una comenzile:

```
firebase functions:config:set paypal.client_id="id-ul aferent proiectului de pe site-ul paypal"
```

```
firebase.functions.config.set paypal.client_secret="client_secret-ul aferent  
proiectului de pe site-ul paypal"
```

Este necesară crearea unei funcții ce va calcula suma de bani pe care pasagerii trebuie să o plătească și pe care firma trebuie să o aloce șoferului ce a creat cererea de plată, din care se scade un eventual comision. Acest lucru fiind posibil prin crearea unei referințe către baza de date Firebase în secțiunea „*history*”, ce deține informațiile tuturor curselor efectuate de către toți șoferii, iar în momentul în care va avea loc crearea unei noi înregistrări a unei curse în această secțiune, această funcție va fi apelată și va prelua distanța cursei pe baza căreia se va calcula prețul. Urmând ca mai apoi să se creeze un câmp în respectiva cursă ce se va numi „*price*”, unde se va afișa prețul (vezi Figura 3.12). Astfel, se asigură securitatea privind calculul sumei de plată, acesta fiind realizată de către server-ul Firebase și nu în codul back-end al aplicației.

```
exports.newRequest =  
functions.database.ref('/history/{pushId}').onCreate((snapshot, context) => {  
  var distance = snapshot.child('distance').val();  
  var price = distance * 3;  
  var pushId = context.params.pushId;  
  return snapshot.ref.parent.child(pushId).child('price').set(price);  });
```



Figura 3.12: Exemplu de înregistrare a unei curse de transport în baza de date

În momentul în care o cursă de transport a fost efectuată, în baza de date, respectivei curse îi va fi asociată o cheie unică prin care poate fi identificată, cheie ce va fi stocată atât în

secțiunea „History” (Figura 3.12), locul unde sunt stocate toate informațiile cursei, cât și în baza de date aferentă șoferului ce a efectuat respectiva cursă (Figura 3.13), unde i se va asocia valoarea *true*. Cheile care au asociată această valoare, identifică cursele pentru care șoferul încă nu a solicitat plata. Din acest fapt este necesară crearea unei funcții JavaScript care va crea un vector cu toate cursele unui șofer, ce au valoarea *true*. Acest lucru a fost realizat în codul de mai jos, funcție care primește ca și parametru, id-ul unui șofer și returnează vectorul ce conține cursele neplătite de către firmă, șoferului.

```
function getPayoutsPending(uid){
    return admin.database().ref('Users/Drivers/' + uid +
    '/history').once('value').then((snap) =>{
        if(snap === null){
            throw new Error("profile doesn't exist");
        }
        var array = [];
        if(snap.hasChildren()){
            snap.forEach(element => {
                if(element.val() === true){
                    array.push(element.key);
                }
            });
        }
        return array;
    }).catch((error) => {
        return console.error(error);
    });
}
```

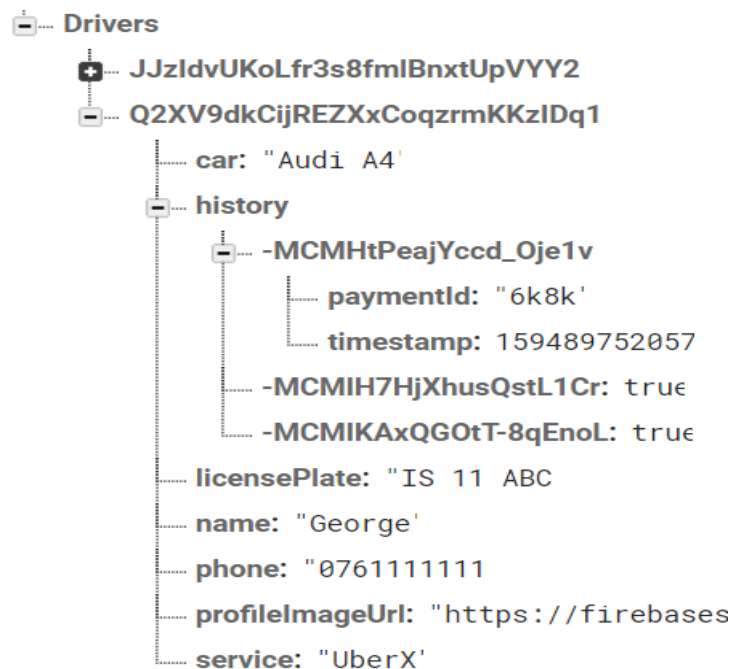


Figura 3.13: Exemplu de înregistrare a unui șofer în baza de date

Vectorul returnat de funcția de mai sus, este dat ca și parametru unei alte funcții ce caută în secțiunea principală „History” (Figura 3.12), respectivele id-uri ale curselor ce se găsesc în vector, curse cărora le sunt extrase prețul generat de funcția „newRequest” de mai sus. Aceste prețuri fiind totalizate într-o variabilă ce va reprezenta suma totală pe care firma este nevoită de a o plăti șoferului pentru cursele efectuate de acesta.

```
function getPayoutsAmount(array){
    return admin.database().ref('history').once('value').then((snap) =>{
        var value = 0.0;
        if(snap.hasChildren()){
            snap.forEach(element => {
                if(array.indexOf(element.key) > -1){
                    if(element.child('price').val() !== null){
                        value += element.child('price').val();
                    }
                }
            });
        }
        return value;
    }).catch((error) => {
        return console.error(error);
    });
}
```

După ce șoferul este plătit, în baza de date se efectuează niște actualizări prin care se evidențiază acest lucru. Pentru asta a fost creată o funcție ce în momentul în care o cursă a fost plătită, aceasta va crea un câmp în secțiunea „History”, sub denumirea „driverPaidOut”, ce va fi setată cu valoarea *true* (vezi Figura 3.12). De asemenea în cadrul profilului șoferului ce a fost plătit, în secțiunea „history” al acestuia, se va adăuga la cursa ce a fost plătită, id-ul aferent plății, cât și data când s-a efectuat plata.

```
function updatePaymentsPending(uid, paymentId){
    return admin.database().ref('Users/Drivers/' + uid +
    '/history').once('value').then((snap) =>{
        if(snap === null){
            throw new Error("profile doesn't exist");
        }

        if(snap.hasChildren()){
            snap.forEach(element => {
                if(element.val() === true){
                    admin.database().ref('Users/Drivers/' + uid + '/history/' +
                    element.key).set({
                        timestamp: admin.database.ServerValue.TIMESTAMP,
                        paymentId: paymentId
                    });
                    admin.database().ref('history/' + element.key +
                    '/driverPaidOut').set(true);
                }
            });
        }
    });
}
```

```

    });
}
return null;
}).catch((error) => {
    return console.error(error);
});
}

```

Toate aceste funcții fiind mai apoi conectate în cererea http trimisă server-ului de către șoferii ce doresc a li se vira sumele de bani de către firmă (vezi Anexa 4.). Cerere inițiată în momentul în care șoferul apasa butonul „Payout” (Figura 3.14) după ce și-a introdus contul de PayPal.

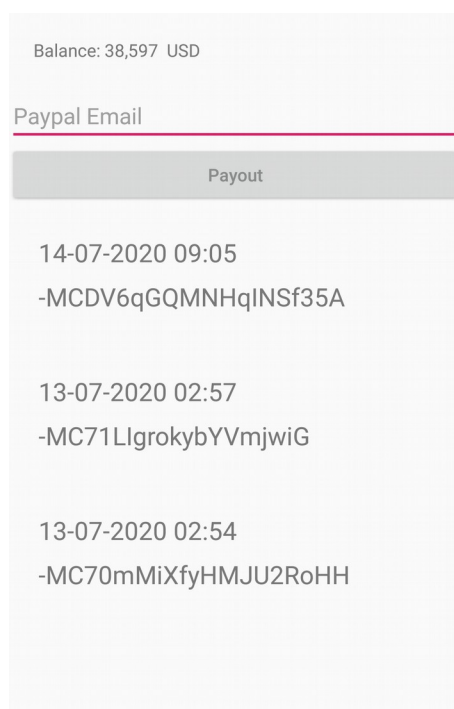


Figura 3.14: Plata șoferului

Unde în Figura 3.14 câmpul „Balance” reprezintă suma pe care șoferul trebuie să o primească de la firmă și totodată acesta reprezintă indiciul că un pasager a plătit transportul, în caz contrar *Balance* ar fi avut valoarea zero.

În Figura 3.14, sub butonul „Payout”, sunt afișate cursele pentru care firma încă nu i-a virat banii șoferului sau șoferul încă nu a făcut o cerere de plată pentru ele. Aceste curse dispar de pe interfață în momentul în care firma efectuează plata șoferului.

3.12.3. Apelul în aplicație a cererii http pentru realizarea plății șoferului

Inițializarea unei cereri http prin care șoferul solicită plata se realizează prin apăsarea butonului „Payout” din Figura 3.14, ce va apela metoda „*payoutRequest()*”. Metodă în care se crează un obiect de tip „*OkHttpClient*” prin intermediul căruia se va trimite cererea http.

```
final OkHttpClient client = new OkHttpClient();
```

Se crează și un obiect de tip „*JSONObject*” ce va conține parametri necesari efectuării plății, mai exact id-ul șoferului ce realizează cererea, cât și email-ul aferent contului său de PayPal.

```
JSONObject postData = new JSONObject();

try {
    postData.put("uid", FirebaseAuth.getInstance().getUid());
    postData.put("email", mPayoutEmail.getText().toString());
} catch (JSONException e) {
    e.printStackTrace();
}
```

Pentru crearea cererii este necesar existența unui obiect „*RequestBody*”, ce va reprezenta conținutul cererii. Prin urmare acesta va primi ca parametri obiectul JSON ce conține datele șoferului cât și un obiect „*MediaType*”, ce descrie tipul conținutului ce îl deține cererea http.

```
public static final MediaType MEDIA_TYPE = MediaType.parse("application/json");
RequestBody body = RequestBody.create(MEDIA_TYPE, postData.toString());
```

În final are loc crearea cererii prin intermediul obiectului de tip „*Request*”:

```
final Request request = new Request.Builder()
    .url("https://us-central1-uberapp-b72f6.cloudfunctions.net/payout")
    .post(body)
    .addHeader("Content-Type", "application/json")
    .addHeader("cache-control", "no-cache")
    .build();
```

În codul de mai sus *url*-ul reprezentând URL-ul țintă ce se va ocupa de aceste cereri, url ce este afișat în Firebase în meniul *functions*. În crearea cererii fiind setat și obiectul „*RequestBody*” creat mai sus.

După crearea și setarea cererii, nu mai rămâne decât să fie trimisă prin intermediul clientului http.

```
client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) throws IOException {
        int responseCode = response.code();
        if (response.isSuccessful()){
            switch (responseCode){
                case 200:
                    Snackbar.make(findViewById(R.id.layout), "Payout Successful!",
                    Snackbar.LENGTH_LONG).show();
                    break;
                case 500:
                    Snackbar.make(findViewById(R.id.layout), "Error: Could not
                    complete Payout", Snackbar.LENGTH_LONG).show();
            }
        }
    }
})
```



```
                break;
            default:
                Snackbar.make(findViewById(R.id.layout), "Error: Could not
complete Payout", Snackbar.LENGTH_LONG).show();
                break;
        }}
    });
```

Clientul Http realizând după execuția cererii, un callback ce va conține răspunsul cererii și în funcție de acest răspuns afișându-se un mesaj corespunzător

Capitolul 4. Testarea aplicației și rezultate experimentale

Testarea reprezintă o etapă extrem de importantă în crearea unei aplicații software deoarece aceasta pune în evidență eventualele erori ce ar putea apărea pe parcursul rulării aplicației. Scopul acesteia este de a evidenția prezența erorilor și nu absența lor. Cu ajutorul procesului de testare se poate realiza mai apoi procesul de depanare ce constă în a localiza erorilor, crearea unor ipoteze asupra comportamentului programului, corectarea defectelor și apoi retestarea programului [13].

4.1. Testarea aplicației Android

Testarea aplicației Android s-a realizat cu ajutorul IDE-ului Android Studio care pune la dispoziție un meniu numit *“Android Virtual Device Manager”*, de unde se pot crea dispozitive mobile virtuale cu anumite configurări hardware, prin intermediul cărora se pot testa aplicațiile. Aceste dispozitive virtuale având opțiunea de a fi configurate astfel încât să ruleze o anumită versiune de Android și care să simuleze un anumit tip de telefon.

De asemenea, aplicațiile Android pot fi testate și pe dispozitive reale, fiind necesară activarea pe dispozitivul mobile, a opțiunii *“USB Debugging mode”*, ce poate fi accesat din setările acestuia, ca mai apoi fiind conectat la laptop/pc prin intermediul unui cablu USB, fapt ce va duce la recunoașterea respectivului dispozitiv de către Android Studio și astfel fiind posibilă instalarea aplicațiilor în mod direct pe acestea.

Android Studio are la bază IDE-ul Intelij IDEA, una din cele mai performante platforme din momentul actual ce conține unelte pentru debugging și măsurarea parametrilor (memorie, CPU, network, cantitatea de energie, etc.) a unei aplicații.

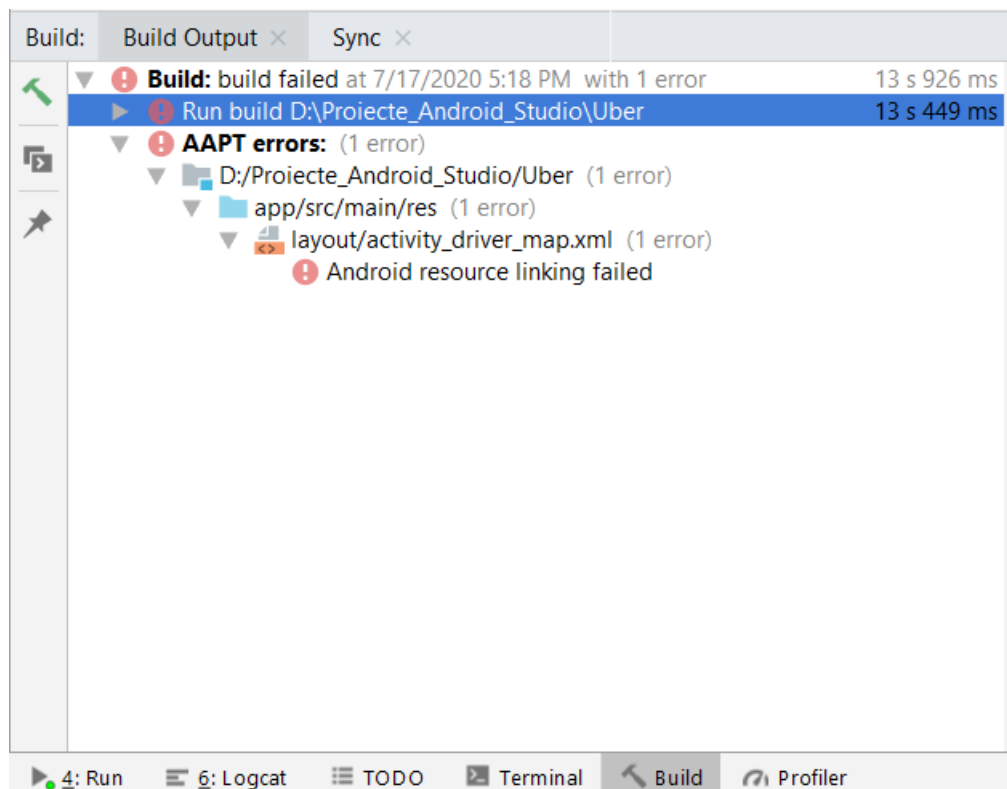


Figura 4.1: Testarea corectitudinii sintaxei de cod pentru Java și XML

În Figura 4.1, secțiunea “Build” al compilatorului asigură că sintaxa fișierelor Java și XML să fie corect implementată, prin afișarea erorilor și a locațiilor problemă, în eventualitatea existenței acestora. Această testare este efectuată de către compilator în momentul în care se dorește a fi instalată aplicația pe emulator sau pe un dispozitiv real. Detaliile privind erorile sunt afișate prin efectuarea unui dublu click pe acestea.

În Figura 4.2, secțiunea “Logcat”, este responsabilă de afișarea eventualelor erori privind logica implementării aplicației în timpul rulării acesteia pe dispozitivele mobile sau emulatoare, cât și a fișierelor Java, unde s-au detectat respectivele erori. Această secțiune este afișată în momentul în care construirea (“build”) și instalarea aplicației a avut loc cu succes. Aici sunt afișate toate procesele ce rulează în background ale dispozitivului mobile sau al emulatorului, chiar și cele care nu fac parte din aplicație. Această secțiune având opțiunea de a selecta dispozitivul pentru care se urmărește rularea aplicației.

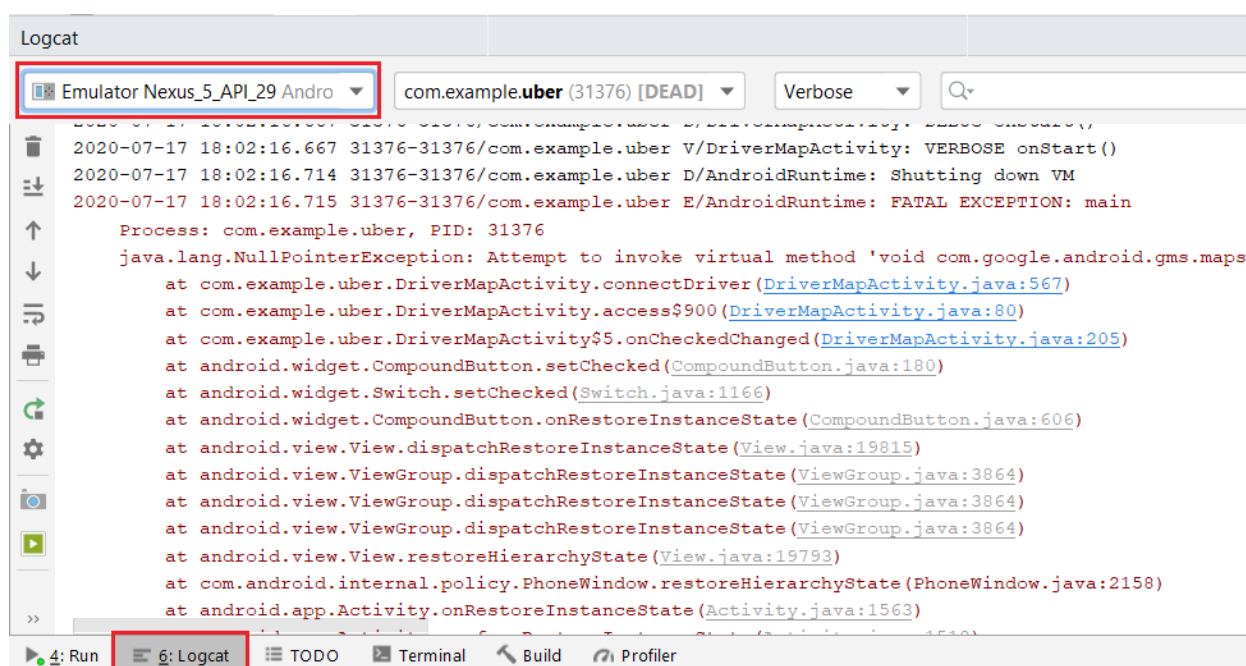


Figura 4.2: Testarea privind logica implementării aplicației

De asemenea, a fost necesară instalarea unei aplicații din app store, pe dispozitivele pe care a fost rulată aplicația, această aplicație numindu-se “Fake GPS”, ce are rolul de a schimba poziția geografică a dispozitivului, cu una nereală și pentru a simula mișcarea utilizatorului. Din acest motiv, această aplicație a fost folosită pentru a simula transportul propriu-zis al persoanelor de către șoferi, prin schimbarea poziției geografice a acestora și pe baza căreia are loc calculul distanței din punctul în care a fost preluat pasagerul, până în punctul în care a fost transportat.

Testarea sistemului de plată reprezentată de PayPal, este realizată de serverul Firebase. Mesajele privind realizarea cu succes al plăților, cât și eventualele erori ce pot interveni, sunt afișate în consola Firebase, ce poate fi accesată printr-un browser web, în meniul *functions*, secțiunea *Logs* (Figura 4.3).

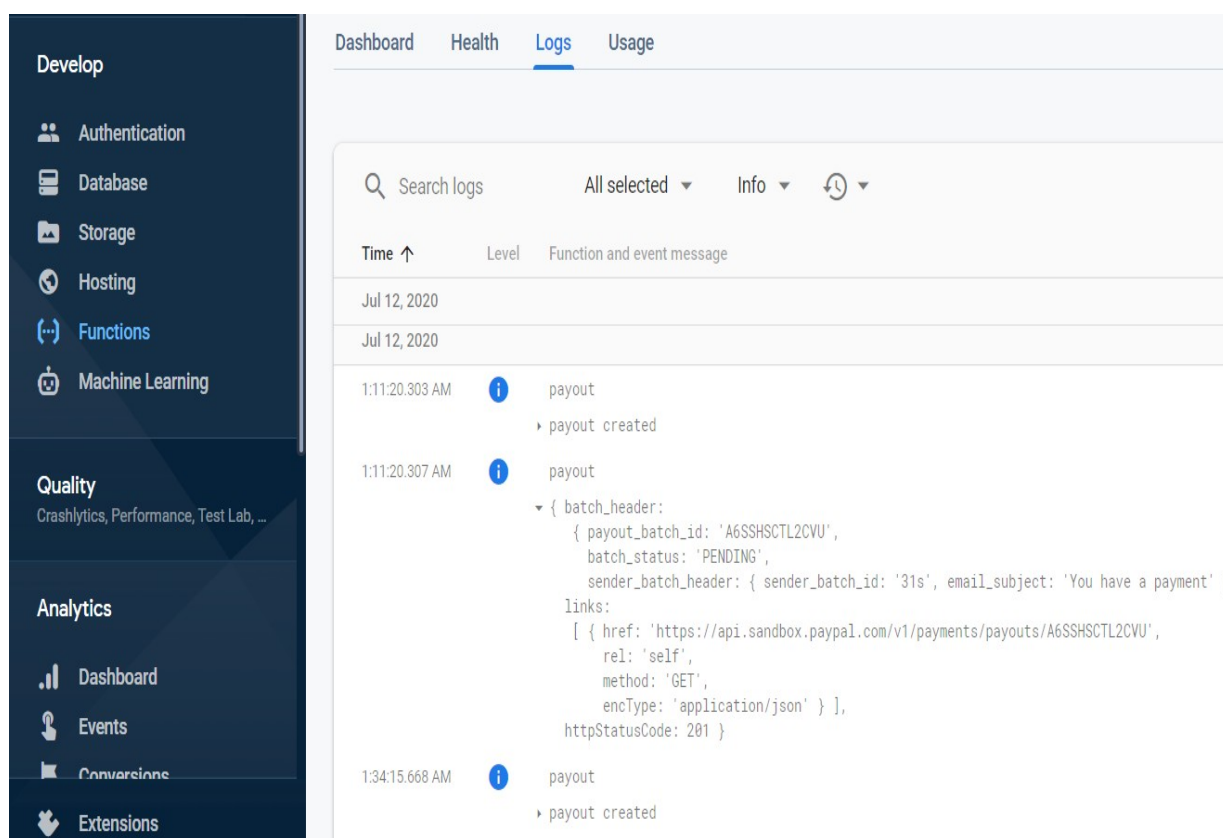


Figura 4.3: Testarea sistemului de plată PayPal

Totodată, plățile ce se realizează cu succes, sunt înregistrate pe site-ul PayPal în secțiunea “Dezvoltator”, meniul “Notifications” al proiectului creat pe site. Aici fiind afișate informații cu privire la suma de bani ce a fost tranzacționată, către și de la cine, data când a fost efectuată, etc. Toate acestea sub forma unui formular (Figura 4.5).

To	Subject	Date
driver112@gmail.com	You have a payment	16 Jul 2020 15:20:12
driver112@gmail.com	You have a payment	16 Jul 2020 15:15:06
customer112@gmail.com	Receipt for Your Payment to John Doe's Test Store	16 Jul 2020 15:13:25
sb-jhc7r1125137@business.example.com	Money is waiting for you	16 Jul 2020 15:13:15
customer112@gmail.com	Receipt for Your Payment to John Doe's Test Store	16 Jul 2020 15:09:25
sb-jhc7r1125137@business.example.com	Notification of payment received	16 Jul 2020 15:09:23
sb-jhc7r1125137@business.example.com	Here's the receipt for your payout	16 Jul 2020 14:06:26
sb-jhc7r1125137@business.example.com	Money is waiting for you	13 Jul 2020 17:24:55

Figura 4.4: Înregistrările plăților utilizatorilor pe site-ul PayPal



Jul 16, 2020 04:04:22 PDT
Transaction ID: 75N47363H27958730

Hello Maria Gheorghe,

**You sent a payment of \$37.97 USD to John Doe's Test Store
(sb-jhc7r1125137@business.example.com)**

It may take a few moments for this transaction to appear in your account.

Merchant
John Doe's Test Store
sb-jhc7r1125137@business.example.com

Instructions to merchant
You haven't entered any instructions.

Description	Unit price	Qty	Amount
Uber Ride	\$37.97 USD	1	\$37.97 USD
Subtotal			\$37.97 USD
Total			\$37.97 USD
Payment			\$37.97 USD
Payment sent to sb-jhc7r1125137@business.example.com			
Payment sent from customer112@gmail.com			
Funding Sources Used (Total)			
CREDIT UNION 1 x-0410			\$37.97 USD

Issues with this transaction?

You have 180 days from the date of the transaction to open a dispute in the Resolution Center.

? Questions? Go to the Help Center at www.paypal.com/help.

Figura 4.5: Exemplu de formular ce conține datele unei tranzacții

În figura de mai sus, fiind prezentat un exemplu de formular ce conține datele unei tranzacții financiare virtuale, realizată de un utilizator de tip pasager, către firma ce deține aplicația serviciului de transport public. Prin acest lucru se verifică corectitudinea datelor transmise în momentul efectuării plăților.

4.2. Rezultate experimentale

Ca și rezultate experimentale, aplicația a fost testată în vederea cantității de resurse hardware consumate în timpul rulării acesteia. Pentru efectuarea acestui lucru fiind folosită așa numitul tool din Android Studio, „Profiler”.

În Figura 4.6 se poate observa cantitatea de memorie RAM ce este necesară pentru rularea în condiții optime a aplicației. Aplicația după autentificare, având nevoie de aproximativ 145 MB de memorie RAM, apogeul ce îl atinge fiind reprezentat de momentul în care utilizatorul intră în istoricul curselor efectuate și dă click pe acestea pentru a vedea datele cu privire la cursele efectuate. În acel moment, cantitatea de memorie RAM necesară ajungând la

aproximativ 250 MB. Acest lucru datorându-se faptului că pe această interfață are loc apelarea atât a serviciilor Google și PayPal, cât și interogarea bazei de date Firebase, ce pune la dispoziție datele cu privire la cursa ce a fost selectată.

Din graficul de mai jos, se observă că procesorul este solicitat în momentul în care utilizatorul se loghează în aplicație și în momentul în care accesează istoricul curselor și efectuează plata lor, ajungând la o limită maximă de 35% din puterea procesorului, pentru o perioadă scurtă de timp. Acest rezultat fiind returnat de către un procesor cu 6 nuclee, ce au o frecvență de 1.4 GHz.

Iar cu privire la rețeaua de internet, aplicația nu este o mare consumatoare de internet, acesta fiind necesară în momentul în care se efectuează plata curselor, moment în care server-ul trimite răspunsul cu privire la starea plății efectuate, astfel ajungând la maximul de utilizare a rețelei de internet de aproximativ 1.2 MB/s, pentru o scurtă perioadă de timp.

Aceste rezultate fiind asemănătoare atât pentru interfața pusă la dispoziție utilizatorilor de tip pasager, cât și a celor de tip șofer.

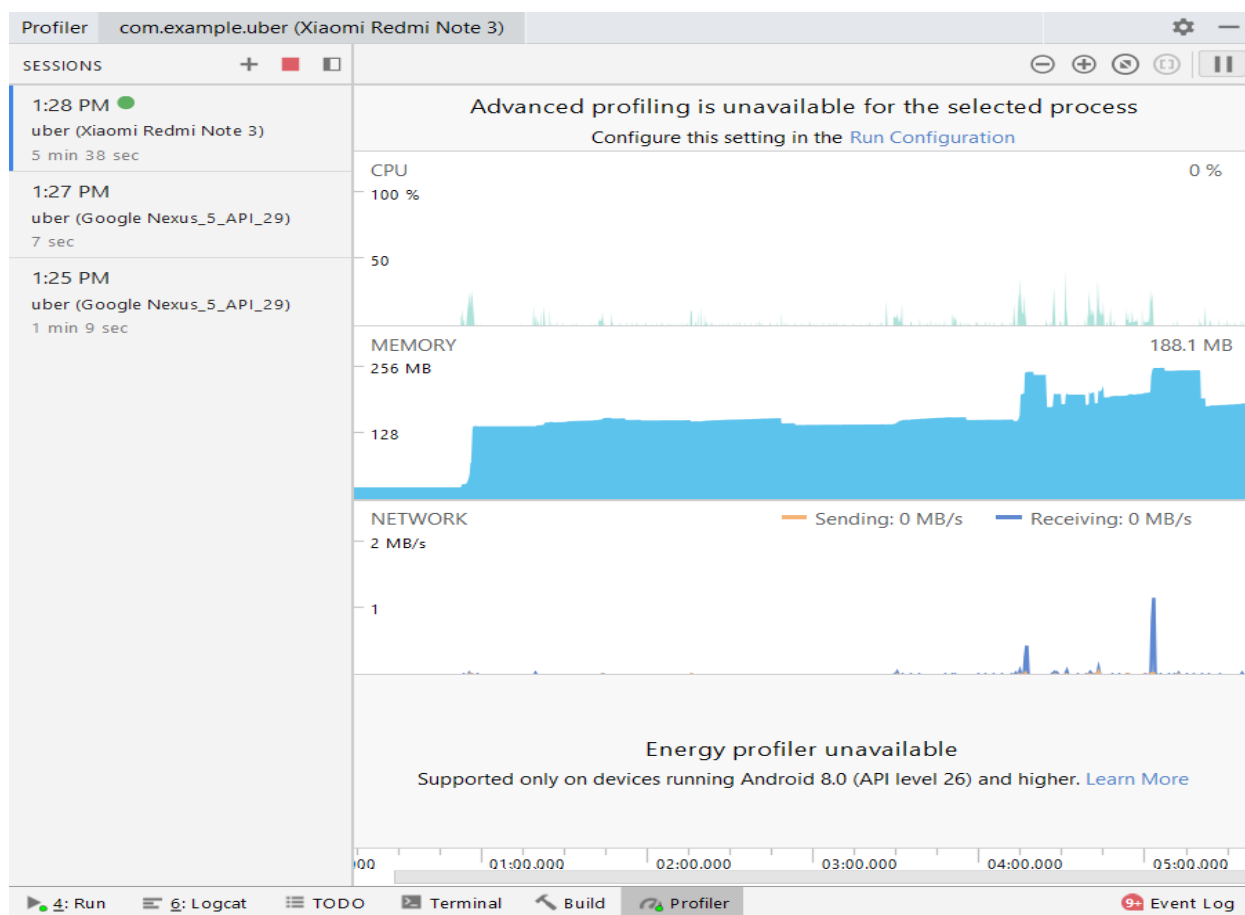


Figura 4.6: Consumul de resurse în timpul rulării aplicației

Concluzii

Principala concluzie ce trebuie a fi subliniată este faptul că în momentul actual, stadiul în care aplicația se află, face din aceasta să reprezinte doar un prototip al unei aplicații cu adevărat destinate transportului public. Acest lucru se datorează sistemului de plată implementat ce acceptă momentan doar tranzacții financiare virtuale, pentru care testarea acestuia s-a realizat cu ajutorul unor conturi PayPal virtuale. Un alt aspect ce face din această aplicație să fie doar un prototip este reprezentat de numărul limitat de dispozitive și emulatoare prin intermediul cărora aceasta a fost rulată și testată, aplicația necesitând un număr mare de dispozitive ce ar fi trebuit să o ruleze concomitent, pentru scoaterea în evidență a unor eventuale probleme.

În momentul de față, aplicația are implementat nucleul de funcționalități necesare unei aplicații menite transportului public, prin intermediul cărora utilizatorii dispun de serviciile și instrumentele necesare efectuării cât și oferirii de transport. Un astfel tip de aplicație fiind destul de complex și delicat din punct de vedere al funcționării acestuia deoarece în „joc” sunt puse anumite sume financiare, cât și starea spirituală a utilizatorilor ce doresc a se simți în siguranță în momentul în care folosesc un astfel tip de aplicație și care doresc servicii de calitate.

Privind simularea și testarea aplicației, aceasta a fost rulată pe maxim trei dispozitive mobile, datorită limitării ca și număr al acestora și în care rularea aplicației și a funcționalităților acestuia s-au realizat fără probleme.

Pentru ca aplicația să poată fi pusă într-un magazin de aplicații, aceasta necesită doar mici modificări în codul actual legat de efectuarea plăților, pentru ca aceasta să fie capabilă de efectuarea tranzacțiilor de plăți reale. Modificări ce constau în schimbarea configurării plății, din tipul „Sandbox” ce reprezintă modul demo al plăților (plăți virtuale), în tipul „Live”, ce permite tranzacționări financiare reale.

De asemenea, pentru ca aplicația să fie completă din orice punct de vedere, mai sunt necesare implementarea multor alte funcționalități, printre care cele mai importante fiind: serviciul de asistență, serviciul de raportare al utilizatorilor, serviciul de penalizare al utilizatorilor, serviciul de plată cu mai multe variante de plată (momentan doar prin PayPal se poate plăti) etc. Prin intermediul cărora, siguranța datelor utilizatorilor să fie mai sporită iar calitatea serviciilor să fie mai ridicată. Fapt ce va duce automat la creșterea încrederii utilizatorilor în aplicație.

Ca și concluzie finală, consider ca acest proiect și-a atins scopul și anume acela de a crea o aplicație menită transportului public ce să conțină doar minimumul de funcționalități necesare utilizatorilor de a crea cereri de transport și a oferi servicii de transport, toate acestea fiind oferite prin intermediul unor interfețe simple, intuitive și interactive.

Bibliografie

- [1] Andrew Hoog, „Android Forensics Investigation, Analysis and Mobile Security for Google Android”, Syngress, 2011.
- [2] Wikipedia, Android (operating system) [Online], Disponibil la adresa: https://en.wikipedia.org/wiki/Android_%28operating_system%29, Accesat: 2020.
- [3] Tomas Katysovas, „A first look at Google Android”, 2008.
- [4] Catalin B, Android tutorial - concepte, activitati si resurse ale unei aplicatii Android [Online], Disponibil la adresa: <http://www.itcsolutions.eu/ro/2011/09/08/android-tutorial-concepte-activitati-si-resurse-ale-unei-aplicatii-android/>, Accesat: 2011.
- [5] ocw.cs.pub.ro, Laborator 02. Structura unei Aplicații [Online], Disponibil la adresa: <https://ocw.cs.pub.ro/courses/eim>, Accesat: 2020.
- [6] Anupam Chugh, Android Studio Project Structure, Compiler, ProGuard [Online], Disponibil la adresa: <https://www.journaldev.com/9319/android-studio-project-structure-compiler-proguard>, Accesat: 2016.
- [7] iCode Academy , „Java for Beginners”, CreateSpace Independent Publishing Platform, 2017.
- [8] Andrei Ciobanu, JavaScript - Noțiuni de bază [Online], Disponibil la adresa: <https://web.ceiti.md/lesson.php?id=16>, Accesat: 2020.
- [9] Mihai Gabroveanu, Introducere in XML (eXtensible Markup Language) [Online], Disponibil la adresa: <http://inf.ucv.ro/~mihaiug/courses/xml/IntroducereInXML.html>, Accesat: 2006.
- [10] Sorin Popa, UML – Unified Modeling Language [Online], Disponibil la adresa: http://cadredidactice.ub.ro/sorinpopa/files/2018/10/L1_diagrame_use_case.pdf, Accesat: 2018.
- [11] Parmit Singh, Android OS Arhitecture [Online], Disponibil la adresa: <http://www.techplayon.com/android-os-architecture>, Accesat: 2017.
- [12] Kishore C.S, ListView vs RecyclerView [Online], Disponibil la adresa: <https://medium.com/@kish.imss/listview-vs-recyclerview-2965d50b363>, Accesat: 2018.
- [13] Florin Leon, Faza de testare (I) [Online], Disponibil la adresa: http://florinleon.byethost24.com/Curs_IP/IP12_Testarea1.pdf, Accesat: 2020.

Anexe.

Anexa 1. Crearea și configurarea proiectului în Android Studio și conectarea la serverul Firebase

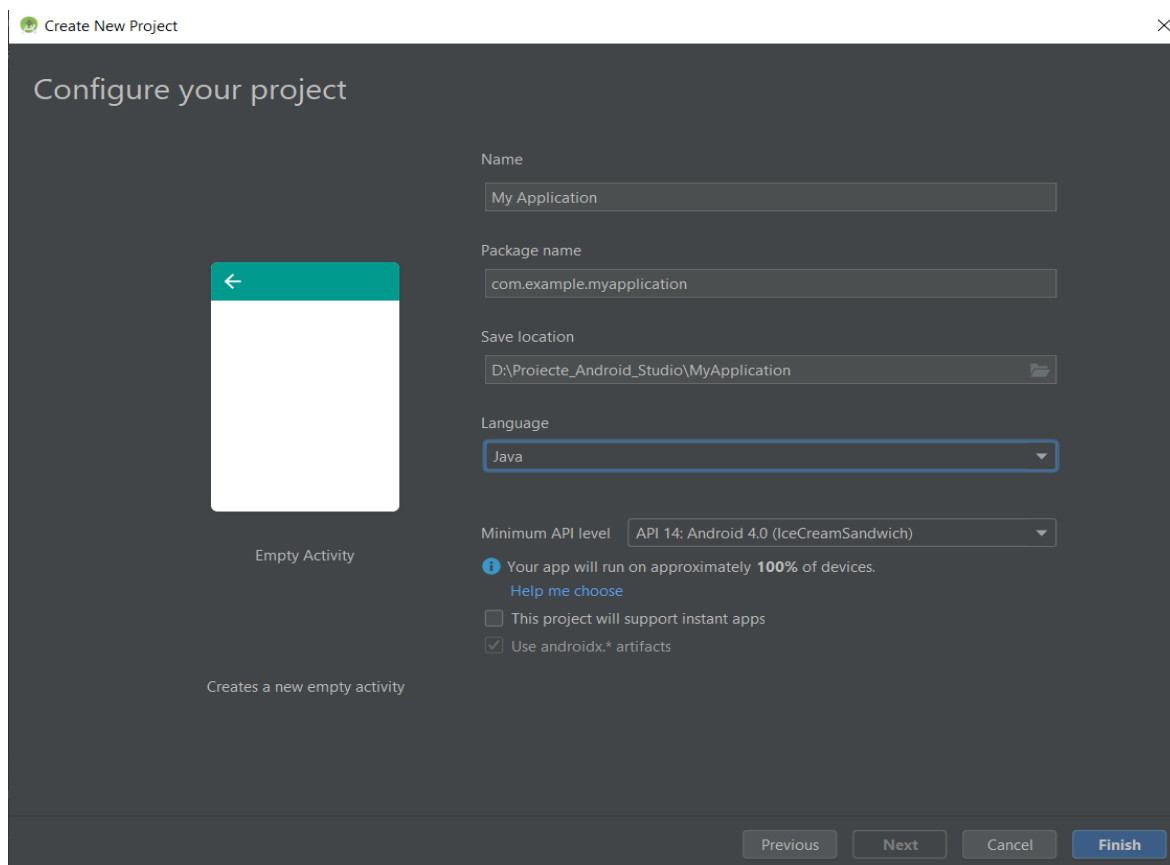


Figura 1: Fereastra de creare și configurare a proiectului

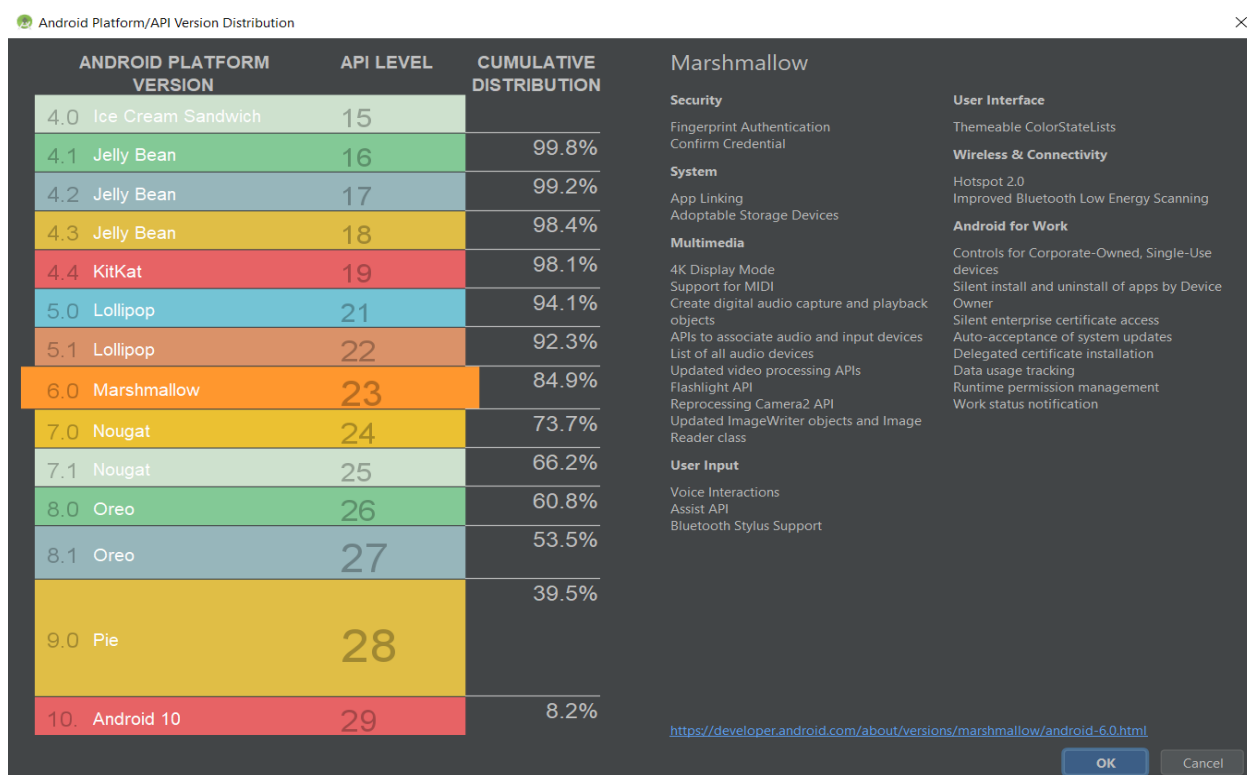


Figura 2: Versiunile Android și API levels

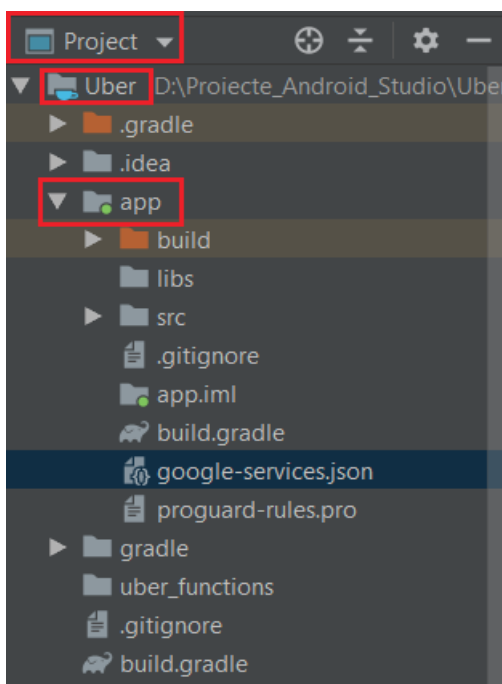


Figura 3: Inserare fișierului JSON generat de către Firebase

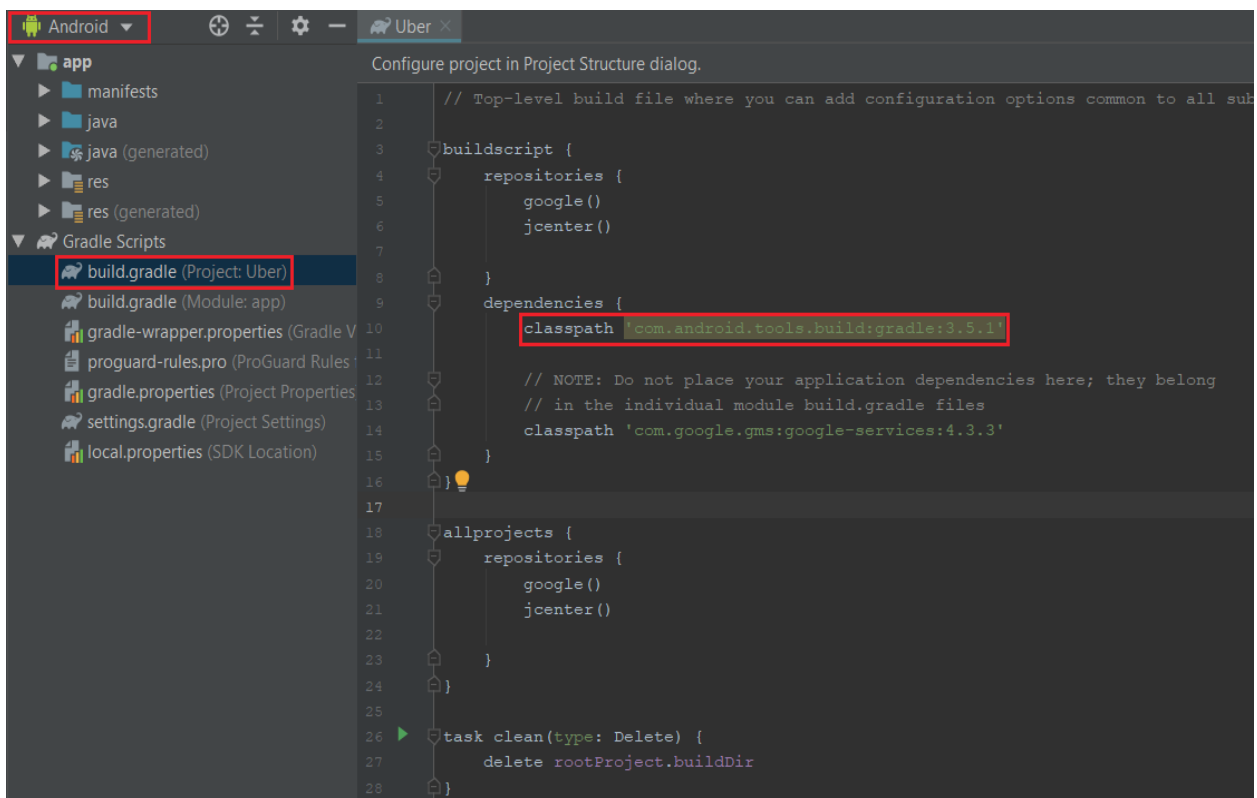


Figura 4: Introducerea dependency-ului generat de Firebase

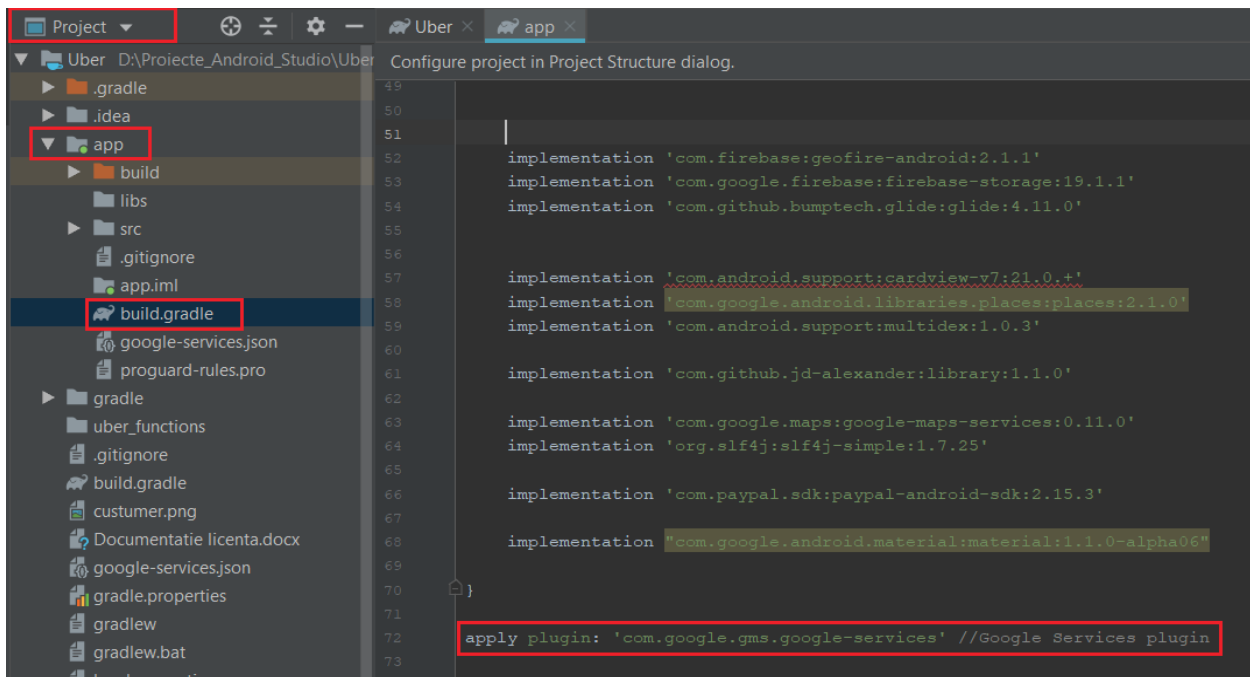


Figura 5: Introducerea plugin-ului generat de Firebase

Anexa 2. Activarea metodei de înregistrare a utilizatorilor din consola Firebase

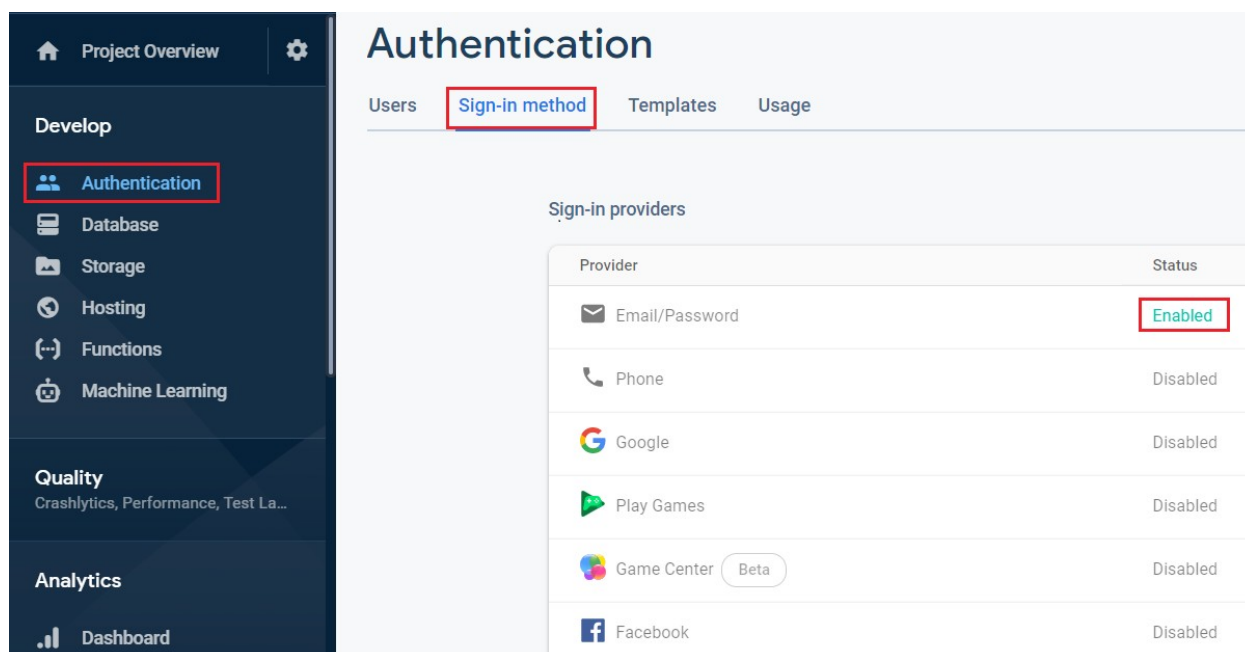


Figura 6: Selectarea metodei de înregistrare a utilizatorilor

Anexa 3. Crearea unei chei API și importarea acesteia în proiect

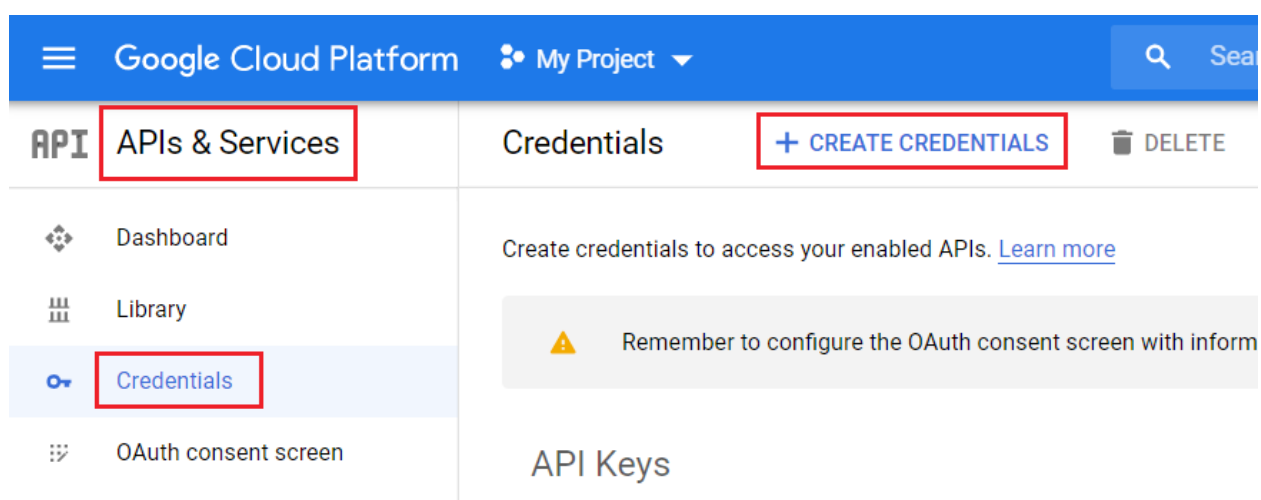


Figura 7: Crearea unei chei API Google

Anexa 4. Codul cererii http pentru realizarea plății șoferului

```

exports.payout = functions.https.onRequest((request, response) => {
  getPayoutsPending(request.body.uid).then((array) => {
    getPayoutsAmount(array).then((value) => {

      var valueTrunc = parseFloat(Math.round(value * 100) / 100).toFixed(2);
      const sender_batch_id = Math.random().toString(36).substring(9);
      const sync_mode = 'false';
      const payReq = JSON.stringify({
        sender_batch_header: {
          sender_batch_id: sender_batch_id,
          email_subject: "You have a payment"
        },
        items: [
          {
            recipient_type: "EMAIL",
            amount: {
              value: valueTrunc,
              currency: "USD"
            },
            receiver: request.body.email,
            note: "Thank you.",
            sender_item_id: "item_3"
          }
        ]
      });
      paypal.payout.create(payReq, sync_mode, (error, payout) => {
        if(error){
          console.warn(error.response);
          response.status('500').end();
          throw error;
        }else{
          console.info("payout created");
          console.info(payout);
          updatePaymentsPending(request.body.uid,
sender_batch_id).then(() =>{
            response.status('200').end();
            return;
          }).catch((error) => {
            return console.error(error);
          })
        }
      });
      return null;
    }).catch((error) => {
      return console.error(error);
    })

    return null;
  }

```

```
}).catch((error) => {  
    return console.error(error);  
})  
});
```