UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICA

LUCRARE DE LICENȚA

Courier Assistant

propusă de

Dragoș-Sebastian Brașovianu

Sesiunea iulie, 2024

Coordonator științific

Lect. Dr. Oana Otilia Captarencu

Cuprins

Introducere

1. Funcționalitățile aplicației

1.1. Autentificare și/sau înregistrare

1.2. Gestionarea comenzilor, atât ca livrator, cat și administrator

1.3. Vizualizarea unei rute și crearea unui traseu

1.4. Gestionarea zonelor de livrare

1.5. Gestionarea utilizatorilor, drept administrator

1.6. Vizualizarea statisticilor

2. Arhitectura si implementare

2.1. Tehnologii utilizate

2.2. Arhitectura aplicației

2.3. Implementare

Concluzii

Bibliografie

Introducere

Pe măsură ce tehnologia evoluează într-un ritm alert, la fel se întâmplă și în cazul comerțului electronic astfel că domeniul livrărilor se confruntă cu o serie de provocări printre care se includ și următoarele: timpul de livrare redus (clienții din ziua de azi au așteptări tot mai mari, punând astfel presiune pe companiile de curierat pentru a reduce timpul de livrare), gestionarea eficientă a comenzilor (volumul masiv de comenzi poate suprasolicita sistemele și poate duce la erori de procesare) , a traseelor de livrare (orașele mai mari tind sa aibă un trafic mai dens astfel că reduc timpul de livrare dacă rutele nu ar putea fi schimbate), a curierilor, precum și evidența informațiilor și statisticilor. Evoluția rapidă a vânzărilor online a dus la o explozie a volumului de comenzi pe care firmele de curierat trebuie să-l gestioneze. Volumul masiv de comenzi pune presiune pe infrastructura companiilor de curierat, iar gestionarea eficientă a comenzilor este crucială pentru a evita erorile, întârzierile și pierderile. Pe lângă acest fapt, gestionarea comenzilor trebuie să se facă și într-un timp rapid astfel încât să satisfacă nevoile clienților de a-și primi produsele cât mai repede posibil. Pentru a se atinge acest scop, firmele de curierat trebuie să aibă în vedere și eficiența operațională a aplicațiilor pe care le dețin, astfel că utilizarea tehnologiilor avansate poate contribui semnificativ la optimizarea operațiunilor și la îmbunătățirea serviciilor de livrare.

Din acest motiv, am ales sa dezvolt o aplicație Android care urmărește să optimizeze operațiunile companiilor de curierat pentru a reduce atât timpul de livrare, cât și timpul gestionării comenzilor în depozitele de livrare. Funcționalitățile aplicației Courier Assistant sunt concepute pentru a răspunde nevoilor specifice lucrătorilor din companiile de curierat pentru a asigura o gestionare eficientă și organizată a activităților. Livratorii pot vizualiza detaliile comenzilor din zonele care le-au fost atribuite, actualiza statusul comenzilor și crea rute de livrare optime sau personalizate în funcție de preferințele și necesitățile lor. De asemenea, aplicația oferă curierilor posibilitatea de a-și vizualiza statisticile personale dintr-o perioadă specifică de timp, facilitand astfel monitorizarea performanței individuale. Pentru administratori, se prezinta aceleași funcționalități, însă include și cateva functionalitati adiționale. Administratorii pot crea noi comenzi, pot edita sau șterge comenzile existente. In plus, acestia au posibilitatea de a administra zonele de livrare și de a gestiona utilizatorii aplicației, asigurandu-se ca fiecare comandă este procesata corespunzător.

Aplicația Courier Assistant pune accentul pe actualizarea și sincronizarea automată a datelor în timp real, ceea ce contribuie la o experiență fluidă și la o gestionare eficienta a operațiunilor indiferent de tipul de utilizator, administrator sau curier.

Lucrarea este structurată în 2 capitole principale. Primul capitol capitol prezintă în detaliu principalele componente ale aplicației cu care utilizatorii pot interacționa, subliniind diferența de atribuții dintre livrator și administrator. Al doilea capitol descrie, în primul rand, tehnologiile utilizate pentru dezvoltarea aplicației, urmat apoi de prezentarea design pattern-ului utilizat ca punct de start, de integrarea serviciilor cloud și structurarea bazei de date, iar în final se exemplifică și se prezintă detalii tehnice de implementare și diferite părți relevante ale codului.

1. Funcționalitățile aplicației

Aplicația Courier Assistant prezintă funcționalități dedicate companiilor de curierat, mai exact livratorilor și administratorilor. Funcționalitățile aplicației sunt împărțite în două secțiuni:

- o secțiune destinată livratorilor, în care aceștia își pot gestiona într-un mod limitat o serie de comenzi (pot vizualiza detaliilor comenzilor din zonele atribuite, pot actualiza stadiul comenzilor), își pot crea o ruta de livrare optimă sau o ruta specifică, cu punctele de livrare alese și în ordinea dorită, și își pot vizualiza statisticile personale pe o perioada specifică de timp (numar de comenzi livrate, eșuate, ore lucrate, distanță parcursă);
- o secțiune destinată administratorilor, in care pot gestiona comenzile (pot crea noi comenzi, pot edita comenzile existente, iar stadiul comenzilor poate fi monitorizat în timp real), zonele de livrare, utilizatorii.

După cum se poate vede în Figura 1, am realizat o diagramă use case care reprezintă toate funcționalitățile aplicației și modul în care acestea sunt accesate de utilizatori.

Pentru o imagine mai clară a funcționalităților aplicației, voi incepe prin a le explica separat:

1.1. Autentificare și/sau înregistrare

Pentru această funcționalitate am ales să implementez o interfață simplă și indiferent de tipul de utilizator, autentificarea se realizeaza prin utilizarea unui e-mail și a unei parole pe care utilizatorul le-a ales în momentul înregistrării. Pentru partea de înregistrare se păstrează aceeași funcționalitate ca la partea de autentificare, doar că pe lângă câmpurile de e-mail și parola, se vor completa și trei câmpuri adiționale. Totodată, cu toate că înregistrarea este liberă pentru oricine, doar administratorul poate asigna zone specifice de livrare utilizatorilor.

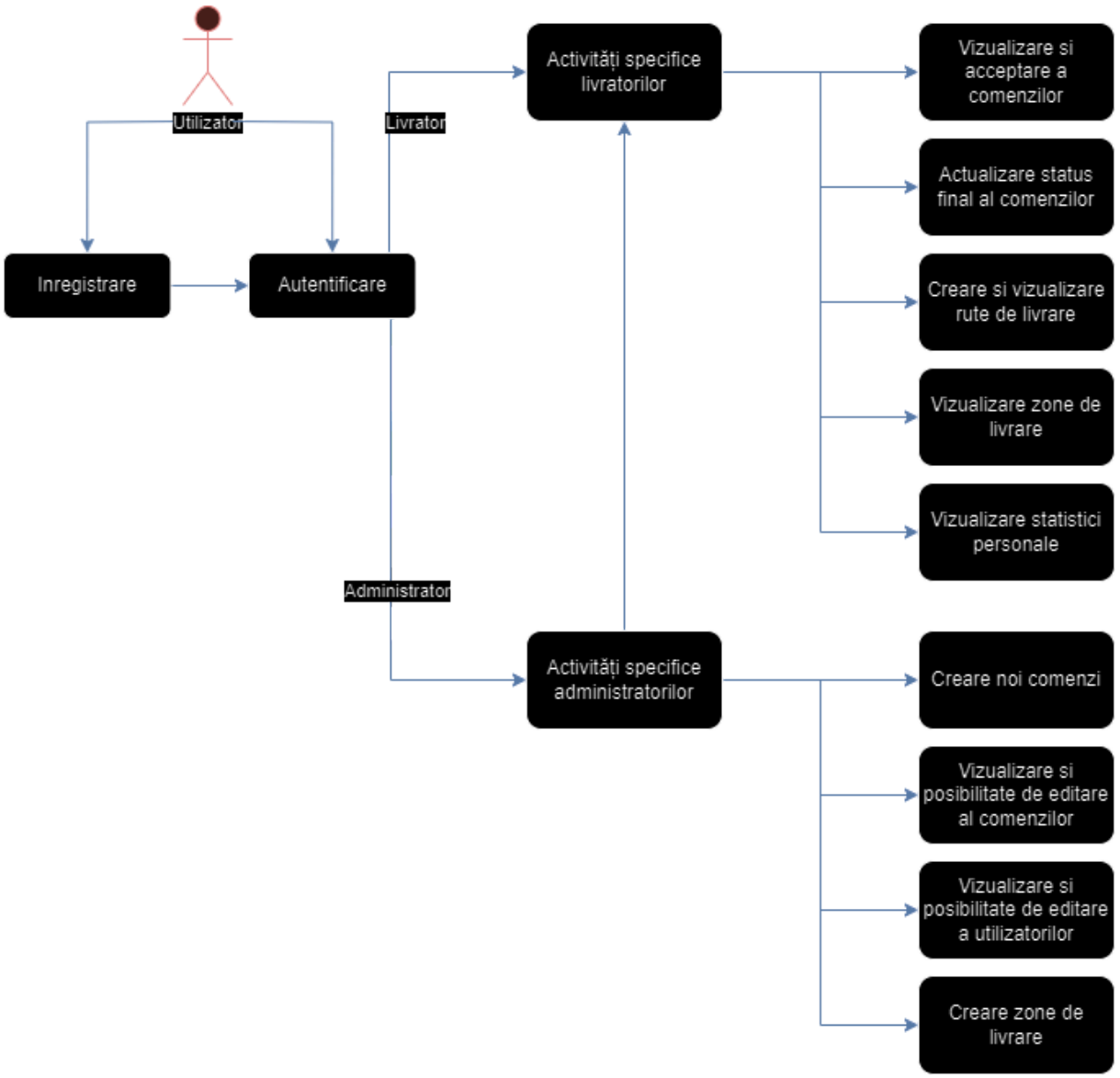


Figura 1: Diagrama use case

1.2. Gestionarea comenzilor

Cum spuneam la început, funcționalitățile diferă în funcție de utilizator, astfel că livratorii pot vizualiza comenzile gata de ridicare în timp real în meniul de notificari (Figura 2), unde sunt afișate detaliile comenzilor, precum ID-ul comenzilor, zona de livrare, adresa de livrare, data limită de livrare. Totodată, aceștia pot confirma ridicarea comenzii, statusul acesteia schimbându-se automat din “In progress” în “Accepted” in baza de date.

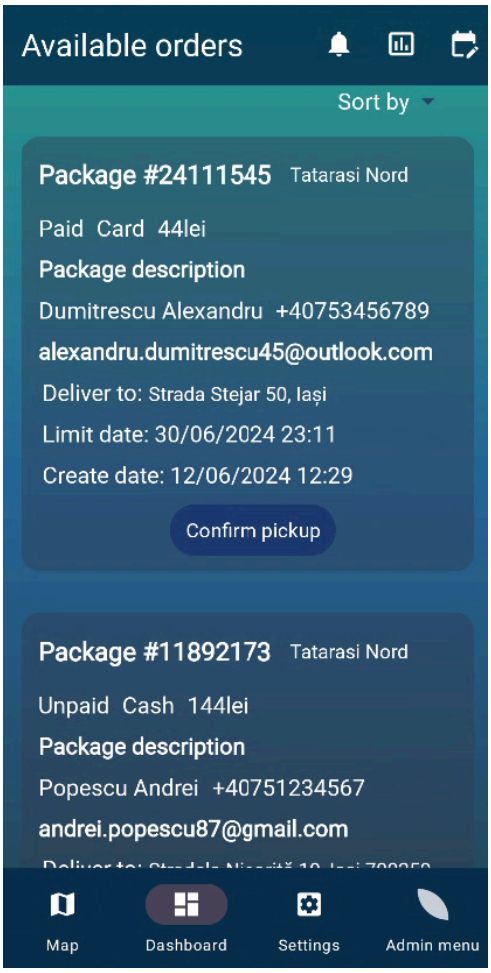


Figura 2: Meniul de notificări/
lista de comenzi disponibile în depozit

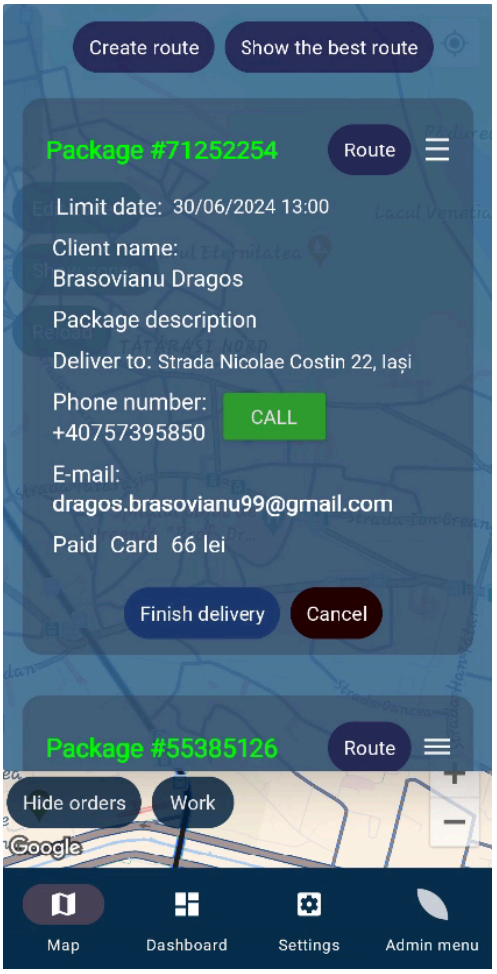


Figura 3: Lista de comenzi acceptate

Comenzile cu statusul “Accepted” apar într-o listă separată (Figura 3), în meniul ce conține si harta unde livratorii pot să-și stabilească traseul de livrare reordonând lista prin drag-and-drop și apasând butonul de creare de rută (personalizat sau optim). Tot in lista curentă, aceștia pot actualiza statusul final al comenzilor în funcție de rezultatul livrării (Figura 4). Statusul final poate fi setat ca

“Livrat” doar dacă livratorului i se comunica codul PIN corespunzător comenzii de către clientul care urmează sa ridice comanda. În caz contrar și în caz de finalizare a comenzii, livratorul trebuie sa aleagă un alt status și să introducă informații adiționale care explica statusul ales, daca este necesar.

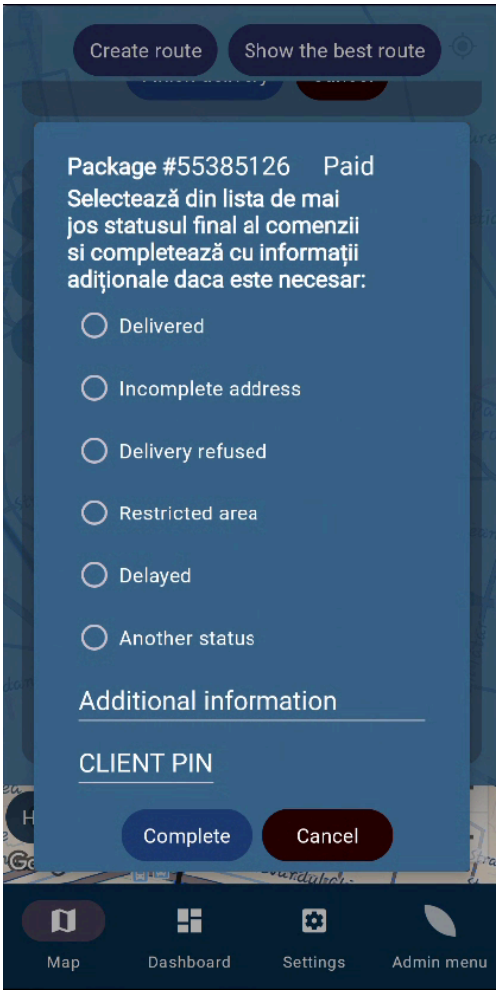


Figura 4: Finalizare comandă

Administratorii, pe de alta parte, dispun și de un meniu nou, dedicat acestora. Aici, aceștia au posibilitatea de a crea noi comenzi, care se adăugă după aceea în baza de date. Procesul de gestionare a comenzilor este flexibil si detaliat, permițând administratorilor să modifice câte o singură proprietate sau pe toate. Pot actualiza statusul de livrare al unei comenzi, pot asigna comenzi către curieri specifici (prin specificarea adresei de e-mail a curierului dorit) și actualiza diverse informații ale clienților, cum ar fi numele, adresa de livrare, numărul de telefon, adresă de e-mail. În momentul în care se dorește editarea și se utilizează butonul de

editare, datele deja existente sunt încărcate în câmpurile de editare, aceasta asigurând o experiență de utilizare intuitivă.

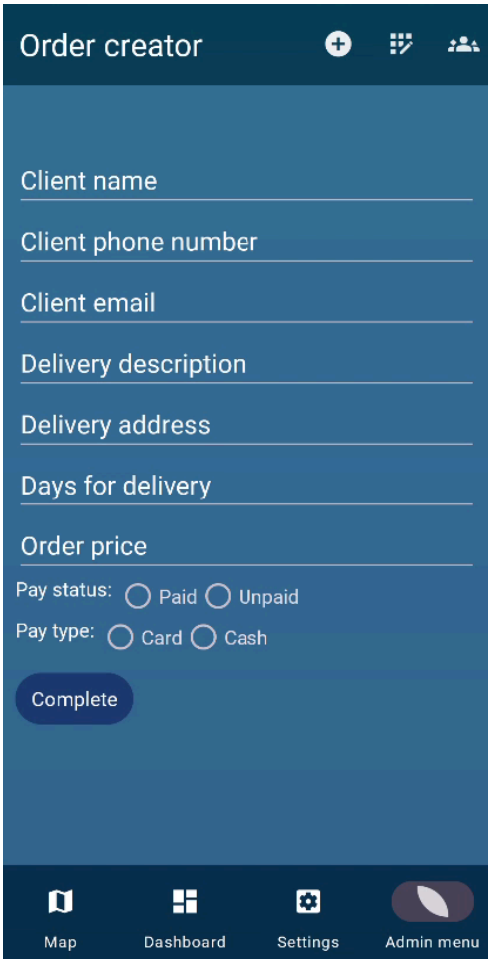


Figura 5: Meniu de creare comandă nouă

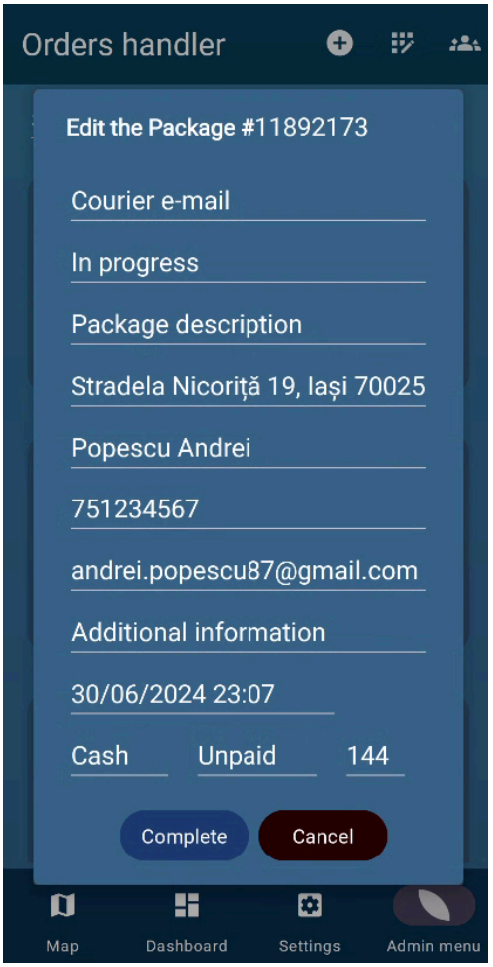


Figura 6: Meniu de editare al unei comenzi

De subliniat faptul că lista de comenzi din meniul dedicat administratorilor dispune și de cateva functionalități ajutătoare de căutare, de sortare, de filtrare care ar facilita gestionarea comenzilor din baza de date.(Figura 7).

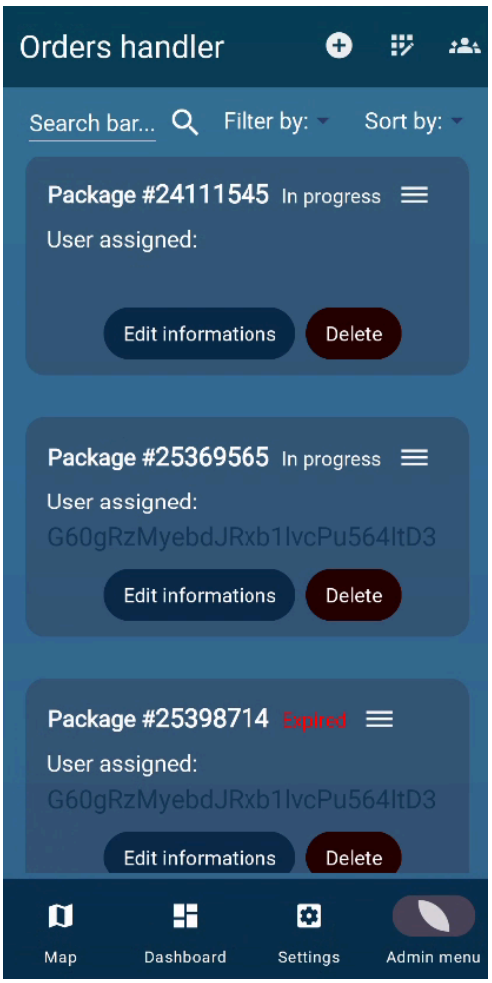


Figura 7: Lista de comenzi(Admin menu)

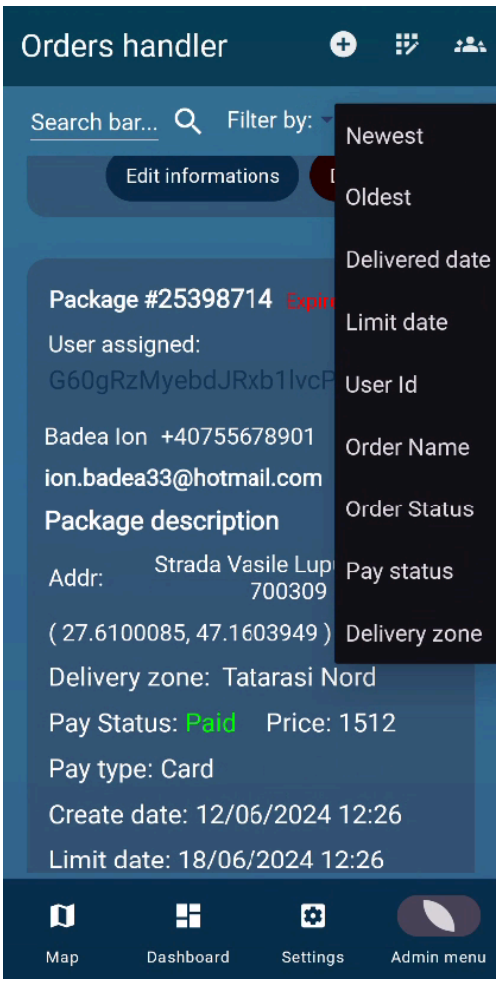


Figura 8: Posibilități de sortare

1.3. Vizualizarea unei rute și crearea unui traseu

Cum spuneam mai devreme, după ce comenzile sunt acceptate ajung într-o listă accesibilă în meniul ce conține și harta. De aici, livratorii beneficiază de posibilitatea de a crea rute pentru livrări într-un mod flexibil și eficient. Aceștia pot opta să creeze un traseu simplu de la locația lor actuală până la destinația comenzii pentru care au selectat să afișeze ruta. (Figura 9 & 10)

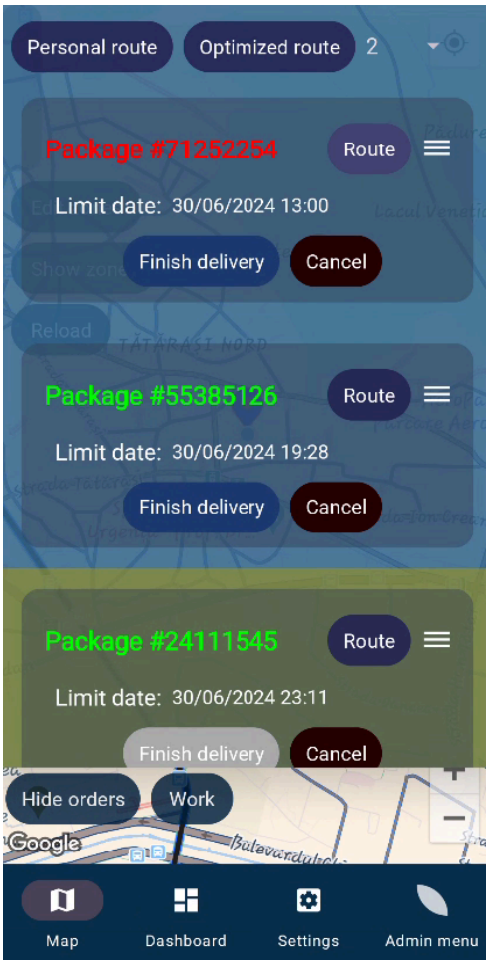


Figura 9: Selectare Route

->

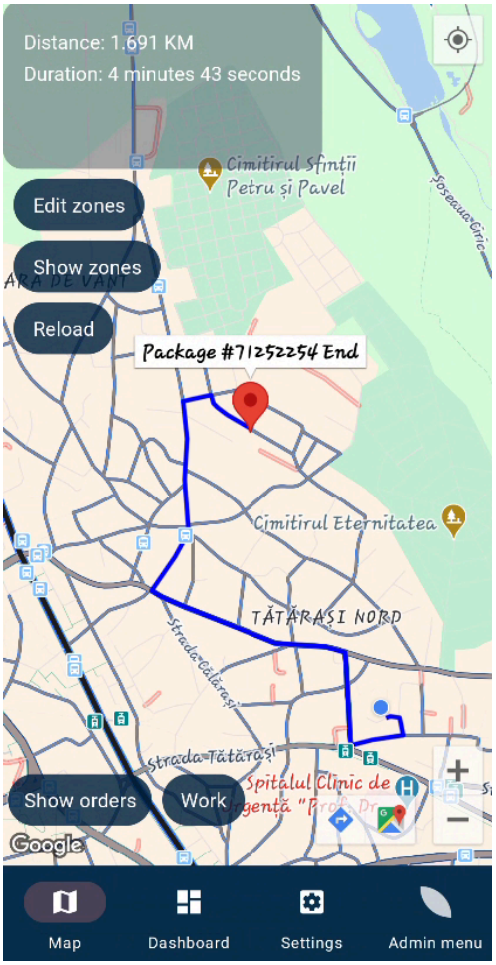


Figura 10: Rută creată

De asemenea, livratorii pot alege să-și creeze o rută personalizată sau optimă, selectând un număr specific de comenzi. Indiferent de tipul de rută ales, aplicația va crea traseul pentru primele *n* comenzi din listă, dar, utilizând funcționalitatea de drag-and-drop, aceștia pot reordona comenzile din listă pentru a stabili pentru ce comenzi se va crea ruta. În Figura 11 se poate observa lista care a fost transmisă pentru a se crea ruta optimă, iar în Figura 12 se poate observa lista care a fost creată după optimizare. Pentru a obține lista din Figura 12, aplic un algoritm prin care sortez mai întâi destinațiile comenzilor în funcție de proximitatea față de locația livratorului. In continuare, pentru a asigura o ordine optimă, aplic o a doua sortare care se concentrează pe minimizarea distanței dintre destinațiile sortate în primă fază, astfel obținând un traseu eficient de parcurs.

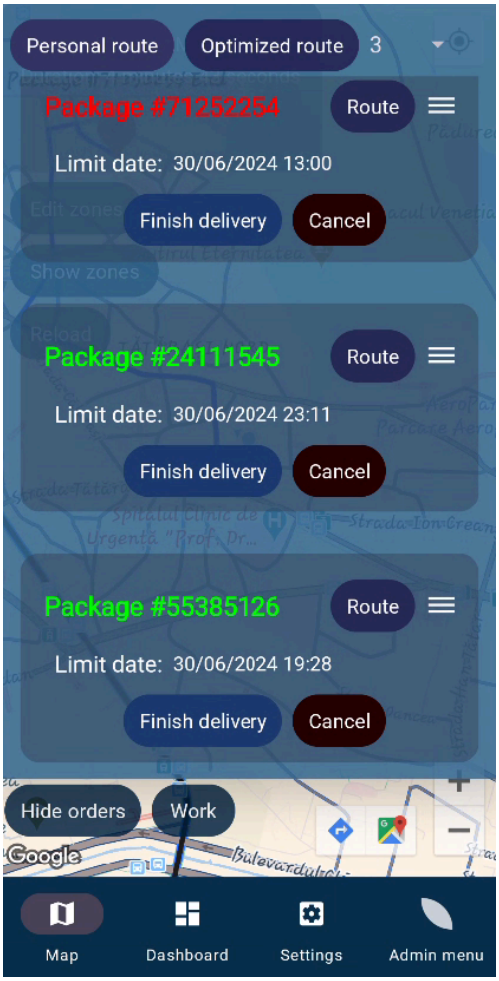


Figura 11: Selectare “Optimized Route"

->

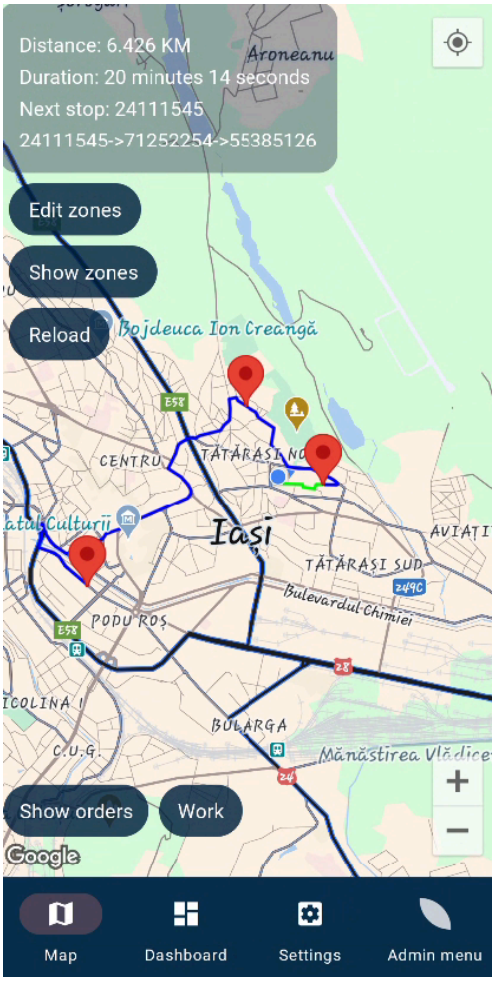


Figura 12: Rută optimă creată

Am ales această abordare deoarece implică flexibilitate și eficiență prin faptul că livratorii pot ajusta rapid rutele în funcție de preferințele personale, de trafic, etc.

1.4. Gestionarea zonelor de livrare

În calitate de livrator, utilizatorul are posibilitatea doar de a vizualiza zonele de livrare pe hartă (Figura 13). Totodată, zonele sunt folosite ca filtre pentru meniul ce conține lista de comenzi pregătite de ridicare. Fiecărui utilizator îi sunt asignate una sau mai multe zone de livrare, iar în meniul respectiv comenzile sunt afișate numai dacă zonele atribuite livratorului coincid cu zonele comenzilor.

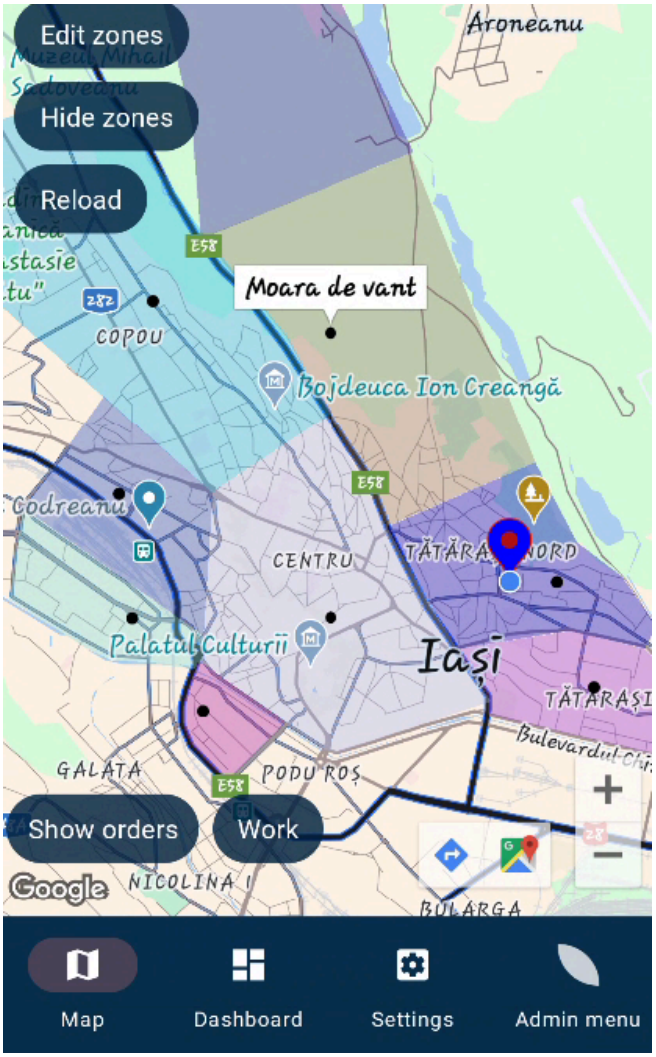


Figura 13: Zone de livrare

Cu toate acestea, in calitate de administrator, utilizatorului i se permite crearea unor noi zone prin utilizarea butonului “Edit zones”. Pentru a crea o nouă zonă de livrare, acesta utilizează o hartă interactivă integrată în aplicație în care trebuie sa plaseze markere. Fiecare marker definește un punct specific al poligonului pe care îl va crea (Figura 14). Odată ce forma poligonului este definită, administratorul va selecta butonul de salvare și va numi zona, moment în care aceasta se va salva în baza de date.

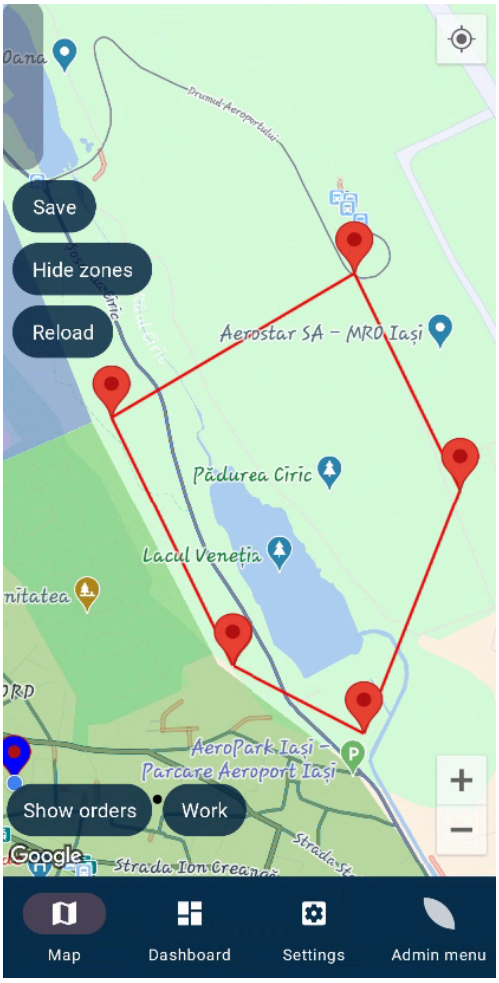


Figura 14: Crearea unei noi zone

1.5. Gestionarea utilizatorilor

Gestionarea utilizatorilor este posibilă doar în calitate de administrator și implică accesul și gestionarea detaliată a tuturor utilizatorilor prezenți în baza de date. Această funcționalitate este crucială pentru asigurarea securității, eficienței și organizării în aplicația Courier Assistant. Administratorii trebuie sa asigneze livratorilor una sau mai multe zone de livrare (Figura 15) astfel încât aceștia să poată prelua comenzi din zonele atribuite.

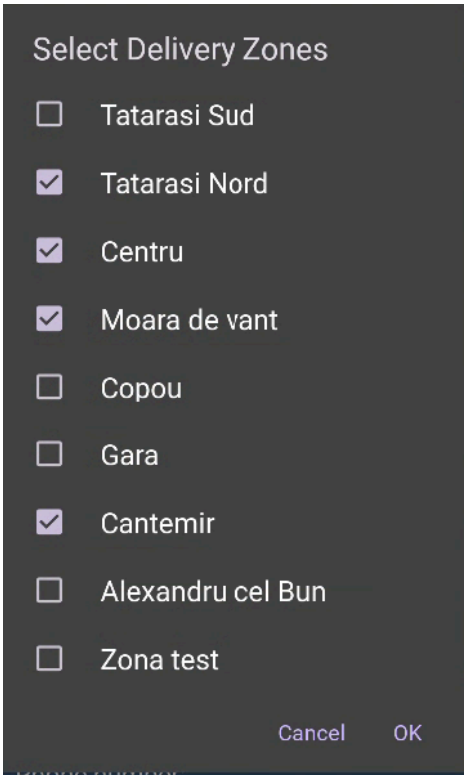


Figura 15: Zone de livrare - Asignare

1.6. Vizualizarea statisticilor

Funcționalitatea de vizualizare a statisticilor oferă livratorilor informații relevante despre progresul livrărilor. Livratorii au posibilitatea de a selecta un interval de timp specific pentru care doresc să vizualizeze statisticile. Statisticile se actualizează automat în funcție de statusul comenzilor. De fiecare dată când statusul unei comenzi se modifică (de exemplu, când o comandă este marcată ca livrată sau alt status), statisticile afișate în aplicație sunt actualizate, fără a fi necesară o intervenție manuală a utilizatorului.

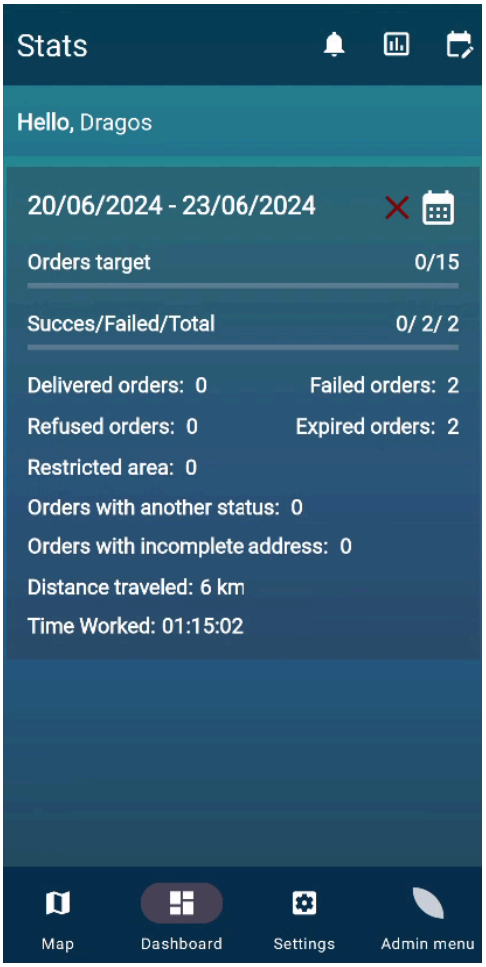


Figura 16: Statistici

2. Arhitectură și implementare

2.1. Tehnologii utilizate

La implementare aplicației Courier Assistant am folosit diverse tehnologii puse în aplicare prin ajutorul platformei de dezvoltare Android Studio, IDE oficial de la Google pentru dezvoltarea aplicațiilor Android. Am ales Android Studio deoarece am fost motivat de:

- **Instrumentele integrate**, care au facilitat procesul de dezvoltare și testare a aplicației. Android Studio permite rularea aplicației pe diferite versiuni și configurații Android prin utilizarea de dispozitive virtuale, gestionate de Android Virtual Device (AVD) care mi-a oferit posibilitatea sa creez diferite tipuri de emulatoare personalizate care au automatizat instalarea unui build de testare în momentul rulării programului. Prin ajutorul unei alte funcționalități din cadrul acestui IDE, Android Debug Bridge (ADB), am putut folosi telefonul personal în aceeași maniera pentru testarea și depanarea aplicației, pentru examinarea unor probleme legate de localizare.

- **Android Jetpack**, un set de biblioteci open-source care oferă funcții esențiale și ajută la dezvoltarea, construirea, gestionarea sarcinilor și facilitarea creării de interfețe de utilizator a aplicației. Din acest pachet au fost folosite o serie de biblioteci:

    - ViewModel, o clasă care este responsabilă pentru gestionarea datelor și logici de afișare a informațiilor din aplicație, utilizată pentru a gestiona datele într-un mod independent față de UI.

    - DataBinding, responsabilă pentru a gestiona componentele UI într-un mod declarativ. Aceasta facilitează actualizarea automată a interfeței atunci când datele se schimbă

- LiveData, o clasa speciala din librăria Jetpack care permite observarea modificărilor aduse datelor din ViewModel, și care a fost folosită pentru actualizarea automată a datelor în componentele UI (fragmente), permițând aplicației sa fie reactiva schimbărilor. Aspectul mai important al acestei clase e faptul ca tine cont de ciclul de viata a componentelor. Deci, gestionarea resurselor este optimizată prin asigurarea actualizărilor doar atunci când sunt necesare și doar pentru componentele relevante și active, contribuind astfel la o experiență de utilizare mai fluidă și mai eficientă.

- RecyclerView, o clasă proiectată pentru afișarea listelor de date într-un mod eficient prin reciclare, în cazul de față lista cu comenzi și/sau utilizatori (în cazul administratorului). Reciclarea se face automat și elimina nevoia de a crea și distruge elemente UI pentru fiecare obiect din listă. Permite gestionarea listelor lungi și variabile pentru o experiență fluentă de utilizare.

- Coroutines, o caracteristică generală a limbajului de programare Kotlin folosită pentru gestionarea fluxului de date în operațiunile asincrone fără a bloca actualizările interfeței de utilizator, asigurând o experiență receptivă. În aplicația Courier Assistant sunt folosite pentru a optimiza autentificarea utilizatorilor, reducând timpul de așteptare pentru conectare, dar și gestionarea datelor utilizatorilor, a comenzilor, a zonelor de livrare și a rutelor, oferind o actualizare mai rapidă.

Tehnologiile pe care le-am utilizat:

- **Kotlin,** este un limbaj de programare orientat obiect, special conceput pentru a facilita dezvoltarea de aplicații Android. Kotlin ajută la evitarea erorilor prin gestionarea clară a valorilor nule, care sunt o sursă comună de erori în alte limbaje de programare, precum Java.

- **Firebase Authenticator**, este o soluție de autentificare care oferă o modalitate sigură și convenabilă pentru ca utilizatorii sa se conecteze la aplicație.

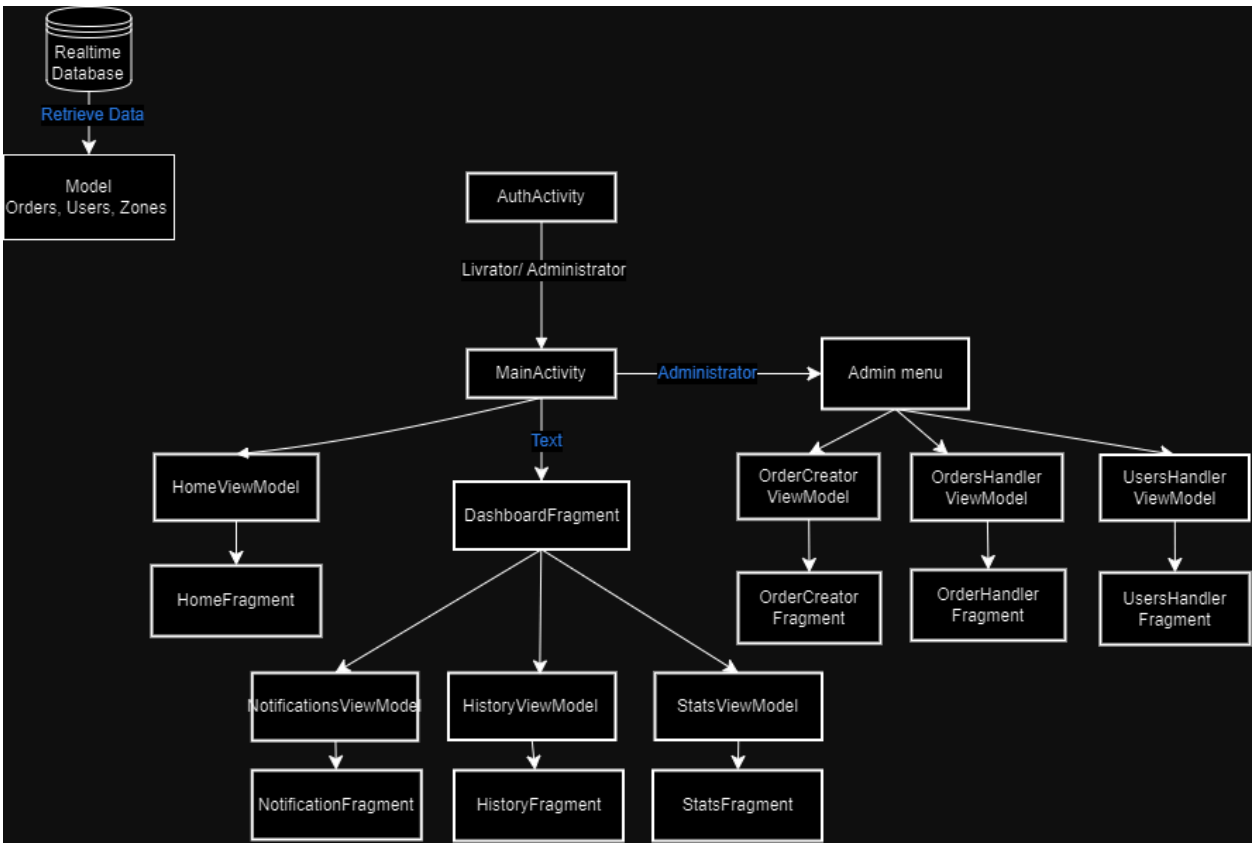
- **Firebase Realtime Database**, este o baza de date NoSQL care stochează și sincronizează datele în timp real. Este o soluție perfectă pentru aplicația Courier Assistant care urmărește să actualizeze instantaneu detaliile comenzilor sau ale utilizatorilor.

- **Google Maps API(Android SDK)**, folosit pentru a afișa harta, marcaje, folosite pentru a evidenția locații specifice pe hartă cum ar fi adresele de livrare, locațiile curierilor și afișarea punctelor de stop a rutelor create de aceștia. Pe lângă API-ul de hartă au fost folosite și:

    - **Geocoding API,** utilizat pentru a transforma adresele fizice introduse la crearea comenzilor în set de coordonate geografice (latitudine, longitudine)

    - **Routes API,** utilizat pentru a calcula rutele optime luând în considerare condițiile actuale de trafic și estimările de timp în consecință

- **Retrofit,** este o librărie ce facilitează interacțiunea cu API-urile web, cum ar fi Routes API si Geocoding API în cazul aplicației Courier Assistant. Permite convertirea automată a apelurilor API în metode Kotlin. Acesta face cereri către serverele definite și convertește răspunsurile de tip JSON în obiecte de tip data pentru rute și coordonate.



2.2. Arhitectura aplicației

După cum se poate observa în figura anterioara, pentru aplicația Courier Assistant am ales sa adopt o arhitectură modulară bazată pe modelul arhitectural Model-View-ViewModel (MVVM) (Figura 17) deoarece acest tip de model separă clar responsabilitățile în trei componente: După cum se observă, aplicației este împărțită în mai multe module, fiecare având o responsabilitate specifică.
In continuare, voi prezenta cele trei componente ale modelului Model-View-ViewModel :

- **Model**: reprezintă datele utilizate în aplicație și include clase precum Orders, Users, Zones, Routes. Aceste clase sunt structurate pentru a reprezenta informațiile stocate în baza de date.
- **View**: reprezintă interfața grafica și afișează datele relevante pentru utilizator. Interfața grafică a aplicației este alcătuită la randul ei din mai multe fragmente distincte fiecare avand un ViewModel personalizat pentru a gestiona logica specifică;

- **ViewModel**: asigură comunicarea dintre Model și View, gestionarea datelor și logica de afișare in View.

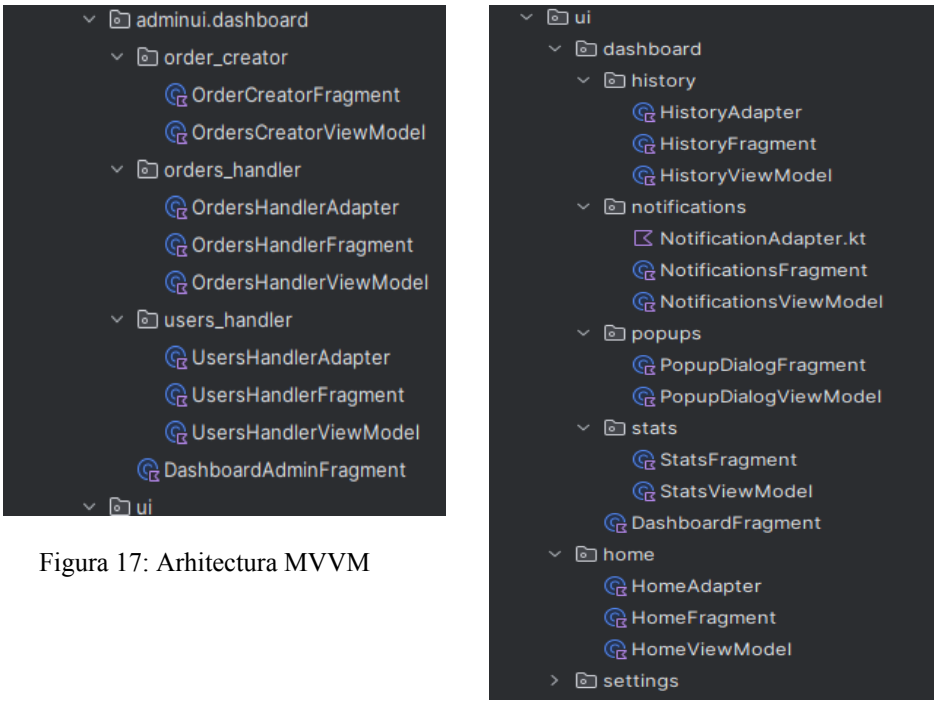


Figura 17: Arhitectura MVVM

Pe lângă arhitectura MVVM descrisă anterior, am ales să integrez diferite servicii din platforma Google Cloud pentru autentificarea utilizatorilor (Firebase Authenticator), stocarea datelor (Firebase Realtime Database) și integrarea hărții în aplicație (Google Maps API).

În cadrul aplicației Courier Assistant, structura bazei de date este creată utilizând o baza de date NoSQL, Firebase Realtime Database, fiind organizată într-o structură ierarhică formată din noduri (Orders, Users, Zones, Routes) și subnoduri, fiecare reprezentând o entitate distinctă.

În nodul “Orders” se stochează informațiile despre comenzile disponibile, fiecare comanda fiind reprezentată și identificată de un subnod unic format sub forma “Order _idUnic”. Acest subnod conține detalii specifice despre fiecare comandă precum numele clientului, informații de contact, detalii de plată, adresa de livrare, date de creare și limită, statusul comenzii, și alte informații adiționale. (Figura 18)

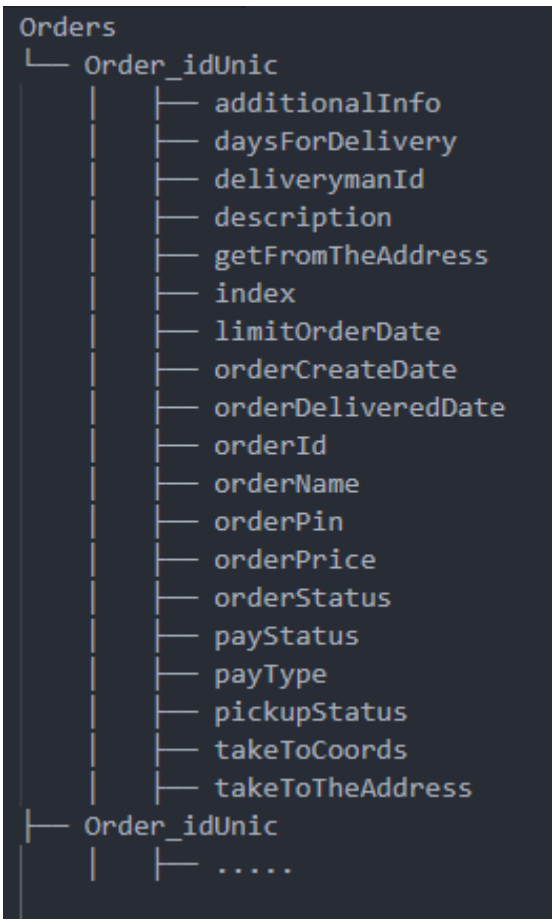


Figura 18: Structura nodului Orders

În nodul “Users” se salvează informații personale despre utilizatori și statistici ale acestora. La fel ca și la nodul “Orders”, utilizatorii sunt identificați de subnodul unic de forma “User_idUnic”, iar nodurile respective includ date personale ale utilizatorului și statistici ale acestuia. (Figura 19)

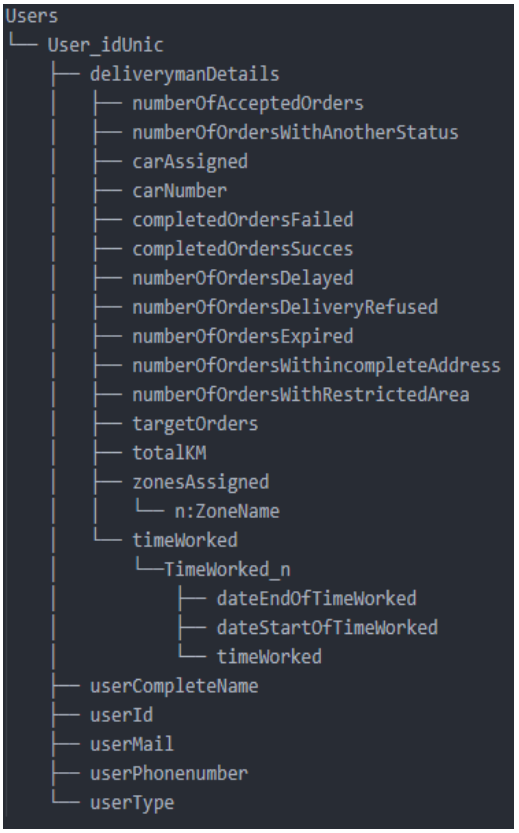


Figura 19: Structura nodului Users

În nodul "Zones" se salvează informații despre zonele de livrare create de administrator, mai exact numele zonei și punctele de delimitare ale poligonului. (Figura 20)

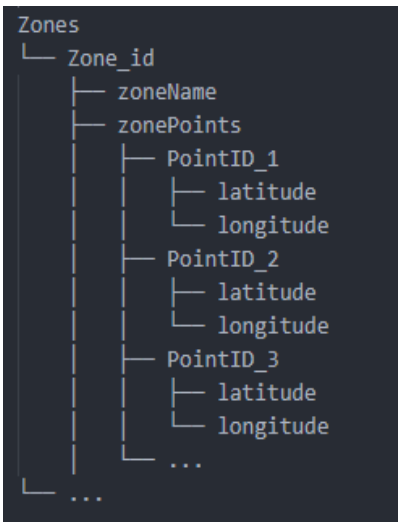


Figura 20: Structura nodului Zones

2.3. Implementarea aplicației

Acest subcapitol are ca scop prezentarea detaliată a procesului de implementare al aplicației. Cum spuneam în subcapitolul Arhitectura aplicației, implementarea aplicației a urmat modelul arhitectural Model-View-ViewModel (MVVM) care a permis o abordare structurată a fiecărui tip de componente, dar activitatea AuthActivity servește ca punct de intrare în aplicație. Aici, în funcție de succesul autentificării se decide dacă activitatea ulterioară (MainActivity) este pentru livratori sau pentru administratori. La inceput, in metoda onStart(), metodă ce ține de ciclul de viață al activității și care este apelată ori de câte ori activitatea devine vizibilă, se verifică dacă există deja un utilizator autentificat, iar dacă există, interfața este actualizată și utilizatorul este redirecționat către MainActivity prin utilizarea funcției updateUI, care are ca scop verificarea înregistrării sau a autentificării cu succes.

```
private val authManager: FirebaseManager = FirebaseManager()

override fun onStart() {
    super.onStart()
    val currentUser = authManager.getCurrentUser()
    if (currentUser != null) {
        updateUI(currentUser, null, isLoginAttempt = true,
isSignupAttempt = false)
    }
}
```

Figura 21: metoda onStart()

```
private fun updateUI(user: FirebaseUser?, exception: Exception?,
isLoginAttempt: Boolean, isSignupAttempt: Boolean) {
    hideLoadingIndicator()
    if(isSignupAttempt && exception == null){
        showAlertDialog("Succes", "The account was created! Now you
can use the account created to login")
    }
    if (user != null && isLoginAttempt) {
        val intent = Intent(this, MainActivity::class.java)
        startActivity(intent)
    }
```

```
    else {
        when (exception) {
            //Firebase exception types
            is FirebaseAuthWeakPasswordException ->
showAlertDialog("Error", "Weak password")
            is FirebaseAuthInvalidCredentialsException ->
showAlertDialog("Error", "Invalid credentials")
            is FirebaseAuthUserCollisionException ->
showAlertDialog("Error", "An account already exists with this email
address.")
            else -> showAlertDialog("Error", "Authentication
failed.")
        }
    }
}
```

Figura 22: metoda updateUI(....)

Apoi, in metoda onCreate() (Figura 23), metodă care este apelată o singură dată în momentul principal pentru a inițializa activitatea, inițializez componentele UI și setez ascultători pentru butonul de autentificare și înregistrare.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.auth_layout)
    usernameInput = findViewById(R.id.username_input)
    passInput = findViewById(R.id.password_input)
    loginBtn = findViewById(R.id.login_btn)
    signupBtn = findViewById(R.id.signup_btn)

    loginBtn.setOnClickListener {...}

    signupBtn.setOnClickListener {...}
}
```

Figura 23: AuthActivity onCreate()

În momentul în care butonul de autentificare sau înregistrare este apăsat, se verifică dacă datele introduse (e-mail și parolă) sunt valide, iar apoi se face apel la funcționalitatea oferită de Firebase Authenticator de autentificare/înregistrare. În

momentul înregistrării, salvez adițional in Realtime Database datele de contact, zone preferate și ID-ul utilizatorului pentru implementări ulterioare ce necesită aceste informații. Înainte de a trece mai departe la activitatea principală, vreau sa subliniez funcționalitățile clasei FirebaseManager care este responsabilă de operațiunile și interacțiunea în cadrul serviciului Firebase Authenticator si Realtime Database. Această clasa gestionează autentificarea utilizatorilor, ascultarea modificărilor din baza de date în timp real pentru Orders, Users, Zones, permițând aplicației sa reacționeze automat la modificările aduse bazei de date (creare, ștergere, actualizare), gestionează operațiunile efectuate asupra comenzilor, utilizatorilor, zonelor (creare, ștergere, actualizare).

In continuare, MainActivity (Figura 24) este de fapt activitatea principală a aplicației și gestionează navigarea între fragmente prin intermediul unui element de interfață de la Google "BottomNavigationView", și prin utilizarea unui "navHostFragment", care este de fapt un container special ce conține fragmentele și gestionează navigarea conform unui grafic de navigație (Figura 25). Tot aici verific dacă utilizatorul care este autentificat este administrator sau nu, eliminând butonul specific administratorului în cazul în care nu este.

```
private lateinit var binding: ActivityBinding
private val authManager: FirebaseManager = FirebaseManager()

override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   binding = ActivityBinding.inflate(layoutInflater)
   setContentView(binding.root)
   val navView: BottomNavigationView = binding.navView
   val navHostFragment =
supportFragmentManager.findFragmentById(R.id.nav_host_fragment_acti
vity_main) as NavHostFragment
   val navController = navHostFragment.navController
   navView.setupWithNavController(navController)
   val navMenu = navView.menu
   authManager.checkIfUserIsAdmin { isAdmin ->
       if (!isAdmin) {
           navMenu.removeItem(R.id.navigation_dashboard_admin)
       } else {
           Log.d("Auth", "Admin logged")
       }
   }
}
```

Figura 24: MainActivity

```
  <navigation
xmlns:android="http://schemas.android.com/apk/res/android"
     xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
      xmlns:tools="http://schemas.android.com/tools"
      android:id="@+id/mobile_navigation"
      app:startDestination="@id/navigation_dashboard">

      <fragment
          android:id="@+id/navigation_home"
          android:name="com.example.locapp.ui.home.HomeFragment"
          android:label="@string/title_map"
          tools:layout="@layout/fragment_home" />

      <fragment
          android:id="@+id/navigation_settings"

android:name="com.example.locapp.ui.settings.SettingsFragment"
          android:label="@string/title_settings"
          tools:layout="@layout/fragment_settings" />

      <fragment
          android:id="@+id/navigation_dashboard"

android:name="com.example.locapp.ui.dashboard.DashboardFragment"
          android:label="Dashboard"
          tools:layout="@layout/fragment_dashboard" >
      </fragment>

      <fragment
          android:id="@+id/navigation_dashboard_admin"

android:name="com.example.locapp.adminui.dashboard.DashboardAdminFra
gment"
          android:label="Dashboard Admin"
          tools:layout="@layout/fragment_dashboard_admin" >
      </fragment>

   </navigation>
```

Figura 25: mobile_navigation.xml

Fragmentele sunt utilizate pentru a separa și a organiza funcționalitățile în cadrul aplicației, respectând principiile design pattern-ului Model-View-ViewModel ales pentru implementare. Pentru dezvoltarea aplicației au fost folosite o serie de fragmente (HomeFragment, HistoryFragment, NotificationsFragment, StatsFragment, OrderCreatorFragment, OrdersHandlerFragment, etc), fiecare având funcționalități bine definite și câte un ViewModel particular care se va ocupa de logica de afișare a datelor în interfața utilizatorului. Pe tot parcursul aplicației, ca Model (clasă ce a definit entitățile utilizate în aplicație), au fost folosite clasele Order (Figura 26), Users (Figura 27), Zones, Routes, ce au definit structura unei comenzi, a unui user, respectiv a unei zone și au fost utilizate pentru a reflecta datele stocate in Firebase Realtime Database.

```
data class Order(
    val deliverymanId: String = "",
    val orderUniqueId: String = "",
    val orderPin: Int = 0,
    val orderId: Int = 0,
    val orderClientName: String = "",
    val orderPhoneNumber:  Int = 0,
    val orderClientEmail: String = "",
    val orderName: String = "",
    val description: String =" ",
    val payType: String = "",
    val payStatus: String = "",
    val orderPrice: Int = 0,
    val takeToTheAddress: String = "",
    val getFromTheAddress: String = "",
    val takeToCoords: Coordinates = Coordinates(),
    val getFromCoords: Coordinates = Coordinates(),
    val zoneDelivery: String =" ",
    val orderCreateDate: String = "",
    val orderDeliveredDate: String = "",
    val limitOrderDate: String = "",
    val daysForDelivery: Int = 0,
    val orderStatus: String = "",
    val additionalInfo: String = "",
    val pickupStatus: String = "",
    val orderNumber: Int = 0,
    var index: Int = 0
)

data class Coordinates(
    val latitude: Double = 0.0,
    val longitude: Double = 0.0
)
```

Figura 26: Clasa Order

```
data class Users(
    val userId: String = "",
    val userMail: String = "",
    val userCompleteName: String = "",
```

```
        val userPhonenumber: String = "",
        val userType: String = "delivery man",
        val deliverymanDetails: DeliverymanDetails =
DeliverymanDetails()
    )

    data class DeliverymanDetails(
        val carAssigned: String = "",
        val carNumber: String = "",
        val acceptedOrders: Int = 0,
        val deliveredOrders: Int = 0,
        val targetOrders: Int = 0,
        val totalKM: Int = 0,
        val totalTimeWorked: String = "",
        val incompleteAddress: Int = 0,
        val deliveryRefused: Int = 0,
        val restrictedArea: Int = 0,
        val delayed: Int = 0,
        val expired: Int = 0,
        val anotherStatus: Int = 0,
        val completedOrdersSucces: Int = 0,
        val completedOrdersFailed: Int = 0,
        val zonesAssigned: List<String> = emptyList(),
        val timeWorked: List<TimeWorked> = emptyList()
    )

    data class TimeWorked(
        val timeWorked: String = "",
        val dateStartOfTimeWorked: String ="",
        val dateEndOfTimeWorked: String = "",
        val distanceTraveled: Float = 0f
    )
```

Figura 27: Clasa Users

Fragmentul HomeFragment este unul din fragmentele importante ale aplicației. Acesta implementează funcționalități esențiale pentru afișarea hărții, planificarea rutelor și afișarea acestora pe hartă și gestionarea comenzilor, permițând vizualizarea și actualizarea statusului final. Gestionarea comenzilor se realizeaza utilizând un adaptor personalizat ce este utilizat pentru a afișa lista de

comenzi acceptate. Acest adaptor, HomeAdapter (Figura 28), după inițializare este conectat la RecyclerView și furnizează datele despre comenzi și gestionează viitoarele interacțiuni cu utilizatorul precum vizualizarea pe hartă a rutei către destinația comenzii respective, anularea comenzii, actualizarea statusului final, reordonarea listei (drag-and-drop).

```
homeAdapter = HomeAdapter(
    homeViewModel,
    onShowOnMapClickListener = { order ->
        viewModelScope.launch { this: CoroutineScope
            handleSingleRoute(order)
        }
        googleMap.clear()
    },
    onCancelClickListener = { order ->
        homeViewModel.cancelOrder(order.orderId)
    },
    onDeliverClickListener = { order ->
        showConfirmationDialog(order)
    },
    onCallButton = { order ->
        callTheClient(order)
    }
)

ordersRecyclerView.adapter = homeAdapter
ordersRecyclerView.layoutManager = LinearLayoutManager(requireContext())
```

Figura 28: Home Adapter

Expunerea comenzilor către interfața utilizatorului se face prin intermediul LiveData și MutableLiveData asigurându-se o actualizare continuă și în timp real al datelor afișate în fragment. Procesul se realizează prin observarea schimbărilor listei de tip LiveData (Figura 29: acceptedOrders - variabila inițializată în momentul în care se încarcă lista cu comenzi existente din baza de date prin intermediul SharedViewModel-ului). Lista este apoi sortată (sortare ce ține de indexul care este modificat în momentul în care se realizeaza o rearanjare prin drag-and-drop) și încărcată în adaptorul "homeAdapter" care afișează datele în interfața utilizatorului prin intermediul RecyclerView-ului.

```
sharedViewModel.acceptedOrders.observe(viewLifecycleOwner) { orders ->
    val sortedOrders = orders.sortedBy {it.index}
    homeAdapter.submitList(sortedOrders)
```

Figura 29: Observator acceptedOrders

Vizualizarea unei rute simple se face prin apelarea metodei handleSingleRoute (Figura 30) care creeaza o rută intre poziția actuala a utilizatorului si locația de destinație specificată. Pentru început se preiau coordonatele destinației din comanda order, iar apoi după ce se verifică dacă harta este încărcată se creează un request body conform documentației Routes API necesar pentru a solicita ruta.

```
      private fun handleSingleRoute(order: Order) {
         val endLat = order.takeToCoords.latitude
         val endLng = order.takeToCoords.longitude
         val orderId = order.orderId
         if (isMapReady) {
             val startLatLng = LatLng(startLat, startLng)
             val endLatLng = LatLng(endLat, endLng)
             val originLocation =
LocationWrapper(Location(MyLatLng(startLat, startLng)))
             val destinationLocation =
LocationWrapper(Location(MyLatLng(endLat, endLng)))

googleMap.addMarker(MarkerOptions().position(endLatLng).title("Packa
ge #${order.orderId}"))

             val requestBody = RoutesRequestBody(
                 origin = originLocation,
                 destination = destinationLocation,
                 travelMode = "DRIVE",
                 computeAlternativeRoutes = false,
                 units = "IMPERIAL"
             )
             //Fetch route
             val requestBodyType =
RoutesRequestBodyType.WithoutWaypoints(requestBody)
             val header =
"routes.duration,routes.distanceMeters,routes.polyline.encodedPolyli
ne"

homeViewModel.getRoutesResponse(sharedViewModel.getApiKey(),
requestBodyType, header, viewModelScope)
```

```
homeViewModel.routesResponse.observe(viewLifecycleOwner) { response
->
                response?.let {
                    homeViewModel.routesApiResponse(it, googleMap)
                    val distance =
it.routes.firstOrNull()?.distanceMeters?.toDouble() ?: 0.0
                    val duration =
it.routes.firstOrNull()?.duration ?: ""
                    updateRouteInfo(distance/1000, duration)
                    binding.routeOrderTextView.text =
orderId.toString()
                }
            }
        }
    }
```

Figura 30: Metodă handleSingleRoute(...)

Apelarea API-ului se face prin apelarea metodei “getRoutesResponse” (Figura 31) din HomeViewModel care primește ca parametri cheia api-ului, request body-ul, header-ul și un domeniu pentru corutină. Funcția lansează o corutină pe un fir de execuție optimizat pentru operațiuni de tip I/O (“Dispatchers.IO”). Această abordare asigură faptul că cererile API nu blochează firul principal al aplicației. In continuare, verific tipul de request body, dacă cererea este făcută pentru o rută simpla sau o rută cu o lista de coordonate. După această verificare, apelul API se realizeaza prin utilizarea Retrofit, o bibliotecă Android utilizată pentru a crea solicitări HTTP, în cazul prezent la Routes API și Geocoding API, și pentru a primi un răspuns de tip RoutesResponse. Răspunsul este verificat pentru a determina dacă este favorabil ca mai apoi corpul răspunsului sa fie extras și stocat într-o variabila apiResponse pentru a fi utilizat ulterior.

```
    private val _routesResponse =
MutableLiveData<RoutesResponse?>()
    val routesResponse: LiveData<RoutesResponse?> =
_routesResponse
```

```
      fun getRoutesResponse(apiKey:String, requestBody:
RoutesRequestBodyType, header: String, viewModelScope:
CoroutineScope){
         viewModelScope.launch(Dispatchers.IO) {
             try{
                 val response = when (requestBody) {
                     is RoutesRequestBodyType.WithWaypoints ->
RetrofitClient.routesServices.getRoutes(apiKey, header,
requestBody.body).execute()
                     is RoutesRequestBodyType.WithoutWaypoints ->
RetrofitClient.routesServices.getRoutes(apiKey, header,
requestBody.body).execute()
                 }
                 val apiResponse = if (response.isSuccessful)
response.body() else {
                     Log.d("API RESPONSE ERROR FOR MULTIPLE ROUTES",
"API error: ${response.errorBody()?.string()}")
                     null
                 }
                 withContext(Dispatchers.Main) {
                     _routesResponse.postValue(apiResponse)
                 }
             }
             catch(e: Exception) {
                 withContext(Dispatchers.Main) {
                     _routesResponse.value = null
                     Log.e("getRoutesResponse Error", "$e")
                 }
             }
         }
      }
```

Figura 31: getRoutesResponse(...)

Pentru a expune rezultatul cererii rutei în interfața utilizatorului, am schimbat domeniul corutinei pe “Dispatchers.Main” pentru că interfața poate fi modificată doar din firul de execuție principal, potrivit restricțiilor Android, și am salvat răspunsul rutei într-o variabilă de tip MutableLiveData, care mai apoi este observată în HomeFragment prin variabila publică de tip LiveData(Figura 32). În

cazul în care răspunsul nu este nul, verificare făcută prin expresia ?.let, se salvează și se actualizează interfața utilizatorului.

```
      homeViewModel.routesResponse.observe(viewLifecycleOwner) {
response ->
         response?.let {
             homeViewModel.routesApiResponse(it, googleMap)
             val distance =
it.routes.firstOrNull()?.distanceMeters?.toDouble() ?: 0.0
             val duration = it.routes.firstOrNull()?.duration ?: ""
             updateRouteInfo(distance/1000, duration)
             binding.routeOrderTextView.text = orderId.toString()
         }
      }
```

Figura 32: routeResponse - LiveData

Procesul prin care se vizualizează o rută formată din mai multe puncte de oprire se face aproximativ într-o manieră asemănătoare apelând metoda handleMultipleRoutes(figura 33). Pentru apelul acestei metode, în loc de a folosi o singură comandă, se folosește o lista de comenzi și o variabilă de tip Boolean care stabilește dacă se creează o rută optimă sau o rută personalizată. În continuare se parcurge lista de comenzi, în care se creează 2 tipuri de liste, una ce conține obiecte de tip LocationWrapper, necesare pentru a fi citite corespunzător de request body, o listă ce stochează id-urile comenzilor si un map ce stochează pentru fiecare orderId, obiectul format din coordonate de destinație, LocationWrapper, corespunzător comenzii. Acestea vor fi utilizate mai târziu pe parcursul aplicației. In continuare, după ce aceste variabile ajutătoare au fost create, verific ce tip de rută este cerut. Am ales ca în cazul în care variabila routeType este adevărată, acesta sa reprezinte indicatorul pentru a crea ruta optimă.

```
private fun handleMultipleRoutes(
   orders: List<Order>?,
   routeType: Boolean
) {
   val intermediates = mutableListOf<LocationWrapper>()
   val orderIdsList = mutableListOf<Int>()
   val intermediateMap = HashMap<Int, LocationWrapper>()
```

```
    orders?.forEach { order ->
        val locationWrapper =
LocationWrapper(Location(MyLatLng(order.takeToCoords.latitude,
order.takeToCoords.longitude)))
        intermediates.add(locationWrapper)
        orderIdsList.add(order.orderId)
        intermediateMap[order.orderId] = locationWrapper
        val endLatLng = LatLng(order.takeToCoords.latitude,
order.takeToCoords.longitude)
        if (isMapReady) {

googleMap.addMarker(MarkerOptions().position(endLatLng).title("Packa
ge #${order.orderId}"))

     }
    }
    if (routeType) {
        homeViewModel.sortNearestNeighb(startLat, startLng,
intermediates, intermediateMap, orderIdsList)
    }

    //Request body
    val endLat =
intermediates.lastOrNull()?.location?.latLng?.latitude ?: 0.0
    val endLng =
intermediates.lastOrNull()?.location?.latLng?.longitude ?: 0.0
    val originLocation = LocationWrapper(Location(MyLatLng(startLat,
startLng)))
    val destinationLocation =
LocationWrapper(Location(MyLatLng(endLat, endLng)))
    val requestBody = RoutesRequestBodyWithWaypoints(
        origin = originLocation,
        destination = destinationLocation,
        intermediates = intermediates,
        travelMode = "DRIVE",
        computeAlternativeRoutes = false,
        units = "IMPERIAL"
    )
    val requestBodyType =
RoutesRequestBodyType.WithWaypoints(requestBody)
    val header = "routes.duration,routes.distanceMeters,routes.legs"
```

```
    homeViewModel.getRoutesResponse(sharedViewModel.getApiKey(),
requestBodyType, header, viewModelScope)
    homeViewModel.routesResponse.observe(viewLifecycleOwner) {
response ->
        response?.let {
            homeViewModel.routesApiResponseWaypoints(it, googleMap,
orderIdsList)
            val distance =
it.routes.firstOrNull()?.distanceMeters?.toDouble() ?: 0.0
            val duration = it.routes.firstOrNull()?.duration ?: ""
            updateRouteInfo(distance/1000, duration)
            binding.routeOrderTextView.text =
orderIdsList.joinToString("->")
        }
    }
}
```

Figura 33: handleMultipleRoutes()

În cazul în care se dorește crearea unei rute optime, se apelează funcția de sortare sortNearestNeighb (Figura 34) din HomeViewModel, care este responsabilă pentru a sorta locațiile intermediare din lista "intermediates". În primă parte, se sortează lista "intermediates" folosind metoda sortBy pentru a ordona locațiile intermediare în funcție de distanța față de punctul de start, adică locația curentă a utilizatorului, iar apoi pentru fiecare locație din lista sortată "intermediates", se ia locația precedentă din iterația precedentă și se sortează submulțimea pe baza distanței față de locația precedentă. Această sortare trebuie să asigure ca punctele intermediare sunt sortate într-o ordine care minimizează distanța dintre ele. La final, asigur că lista orderIds reflectă corect ordinea destinațiilor intermediare după sortare.

```
fun sortNearestNeighb(
    startLat: Double, startLng: Double,
    intermediates: MutableList<LocationWrapper>,
    intermediateMap: HashMap<Int, LocationWrapper>,
    orderIds: MutableList<Int>
) {
    intermediates.sortBy { loc ->
        val lat = loc.location.latLng.latitude
        val lng = loc.location.latLng.longitude
```

```
            sqrt((startLat - lat).pow(2) + (startLng - lng).pow(2))
        }

        for (i in 1 until intermediates.size) {
            val prevLocation = intermediates[i - 1].location.latLng //
Previous location
            intermediates.subList(i, intermediates.size).sortBy { loc ->
                val dist = distanceHaversine(prevLocation.latitude,
prevLocation.longitude, loc.location.latLng.latitude,
loc.location.latLng.longitude)
                dist
            }
        }

        val reorderedOrderIds = mutableListOf<Int>()
        for (intermediate in intermediates) {
            val originalOrder = intermediateMap.filterValues { it ==
intermediate }.keys.firstOrNull()
            originalOrder?.let { reorderedOrderIds.add(it) }
        }
        orderIds.clear()
        orderIds.addAll(reorderedOrderIds)
}
```

Figura 34: Sortarea listei cu coordonate intermediare

Pentru calculul distanței am folosit formula Haversine (figura 35), care este o modalitate precisă de a calcula distanța dintre două puncte pe suprafața unei sfere folosind formula:

$$2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right)+\cos\varphi_1\cdot\cos\varphi_2\cdot\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right)$$

, unde:

φ_1, φ_2 - sunt punctele de latitudine pentru punctul 1, respectiv 2,

λ_1, λ_2 - sunt punctele de longitudine pentru punctul 1, respectiv 2,

r - fiind raza Pământului - 6378 KM

```
private fun distanceHaversine(lat1: Double, lon1: Double, lat2:
Double, lon2: Double): Double {
    val rad = 6378 // km
    val dLat = Math.toRadians(lat2 - lat1)
    val dLon = Math.toRadians(lon2 - lon1)
```

```
    val a = sin(dLat / 2) * sin(dLat / 2) + cos(Math.toRadians(lat1))
* cos(Math.toRadians(lat2)) * sin(dLon / 2) * sin(dLon / 2)
    val c = 2 * atan2(sqrt(a), sqrt(1 - a))
    return rad * c
}
```

Figura 35: Sortarea listei cu coordonate intermediare

Un alt fragment important din aplicația Courier Assistant îl reprezintă OrdersHandlerFragment, fragment ce ține de meniul administratorului. Aici sunt afișate toate comenzile disponibile din baza de date, folosindu-se, la fel ca în HomeFragment și în celelalte fragmente în care sunt disponibile comenzile, un RecyclerView în concordanță cu un adaptor. Aici am implementat dropdown-uri de sortare și de filtrare care permit utilizatorului să sorteze, respectiv sa filtreze comenzile în funcție de diferite criterii. Pe lângă această funcționalitate am adăugat și un buton de căutare (Figura 36) care filtrează comenzile pe baza textului introdus in caseta de cautare. Pentru a implementa asta am folosit un observator asupra listei de comenzi ca mai apoi sa filtrez lista folosind expresia when care va afișa orice comandă care va satisface una sau mai multe condiții.

```
private fun searchByName(adapter: OrdersHandlerAdapter) {
    sharedViewModel.orders.observe(viewLifecycleOwner) { allOrders ->
        val editText =
requireView().findViewById<EditText>(R.id.search_bar_edit)
        val searchedText = editText.text.toString()
        val filterOrdersBySearchedId = allOrders.filter {
            when{
                it.orderId.toString().contains(searchedText) -> true
                it.orderClientName.contains(searchedText, ignoreCase
= true) -> true
                it.orderClientEmail.contains(searchedText, ignoreCase
= true) -> true
                it.orderPhoneNumber.toString().contains(searchedText)
-> true
                it.payStatus.contains(searchedText, ignoreCase =
true) -> true
```

```
                it.zoneDelivery.contains(searchedText, ignoreCase =
true) -> true
                it.payType.contains(searchedText, ignoreCase = true)
-> true
                it.takeToTheAddress.contains(searchedText, ignoreCase
= true) -> true
                else -> false
            }
        }
        adapter.submitList(filterOrdersBySearchedId) {
            recyclerView.scrollToPosition(0)
        }
        Log.d("Observer", "OrdersHandlerFragment")
    }
}
```

Figura 36: Funcție de căutare

Pentru a edita comanda, am implementat afișarea unui formular care se auotocompletează în momentul apăsării butonului de editare. După completarea formularului de editare, comanda poate fi actualizată în baza de date dacă una sau mai multe linii de editare au fost modificate. În continuare, în momentul editării adresei, preiau coordonatele adresei respective folosind apelul la Geocoding API(Figura 37), ca mai apoi să-i asignez zona de livrare din care destinația ar trebui sa facă parte.

```
    private fun fetchCoordinates(address: String): Coordinates {
        val geocodingResponse =
RetrofitClient.geocodingServices.getAddressCoordinates(address,
apiKey).execute()
        if (geocodingResponse.isSuccessful &&
geocodingResponse.body()?.status == "OK") {
            val respLoc =
geocodingResponse.body()?.results?.get(0)?.geometry?.location
            if (respLoc != null) {
                return Coordinates(respLoc.lat, respLoc.lng)
            }
        }
        throw RuntimeException("Failed to fetch coordinates")
```

```
    }

    private fun checkTheZoneOfAddress(zones: List<Zones>,
addressCoordinates: Coordinates): String {
        val addressLatLng = LatLng(addressCoordinates.latitude,
addressCoordinates.longitude)

        for (zone in zones) {
            val polygonCoordinates = zone.zonePoints.map {
LatLng(it.latitude, it.longitude) }

            if (PolyUtil.containsLocation(addressLatLng,
polygonCoordinates, true)) {
                return zone.zoneName
            }
        }
        return "Unknown zone"
    }
```

Figura 37: Funcția de preluare coordonate
și funcția de verificare zonă

O altă funcționalitate cheie din aplicația mea este vizualizarea statisticilor care se realizează prin intermediul fragmentului StatsFragment, unde se utilizează o serie de observatori pentru a ține toate informațiile actualizate în concordanță cu ceea ce există în baza de date. Aici folosesc un buton care deschide un dialog ce permite alegerea a două date calendaristice care sunt trimise mai departe către StatsViewModel unde sunt folosite pentru a specifica intervalul de timp pentru care se vor calcula statisticile. După cum se poate observa în Figura 38, odată setate cele doua date calendaristice se apelează metoda startStatsListener (Figura 39) care este responsabilă pentru monitorizarea schimbărilor aduse comenzilor din baza de date.

```
    fun setDateRange(startDate: Date? = null, endDate: Date? =
null) {
        this.startDate = startDate
        this.endDate = endDate
        fetchDeliverymanDetails()
        startStatsListener()
```

```
    }

    private fun startStatsListener() {
        firebaseManager.setStatsListener(
            scope = viewModelScope,
            onStatsUpdated = {
                fetchDeliverymanDetails()
            },
            onCancelled = { error ->
                Log.e("StatsViewModel", "Stats listener cancelled:
${error.message}")
            },
            startDate = startDate,
            endDate = endDate
        )
    }
```

Figura 38: Setarea unui interval de timp

Atunci când se observă o schimbare a comenzilor, aceasta apelează updateUserStats() și updateUserWorkStats(), care calculează statisticile pentru utilizatorul ce este conectat. Folosind interogări firebase, updateUserStats() (figura 40) selectează comenzile din baza de date în funcție de statusul acestora și de intervalul de date, apoi numără câte comenzi îndeplinesc criteriile de filtrare și actualizează valorile corespunzătoare în nodul "deliverymanDetails". În aceeași manieră funcționează și updateUserWorkStats(), doar ca aceasta funcție calculează distanța totală parcursă și timpul total lucrat într-un interval de timp.

```
fun setStatsListener(
    scope: CoroutineScope,
    onStatsUpdated: () -> Unit,
    onCancelled: (DatabaseError) -> Unit,
    startDate: Date? = null,
    endDate: Date? = null
) {
    scope.launch(Dispatchers.IO) {
        try {
            ordersRef.addValueEventListener(object :
ValueEventListener {
                override fun onDataChange(snapshot: DataSnapshot) {
```

```
                    val currentUserId = auth.currentUser?.uid
                    updateUserStats(currentUserId, startDate, endDate)
                    updateUserWorkStats(currentUserId, startDate,
endDate)
                    onStatsUpdated.invoke()
                }
```

Figura 39: Metoda setStatsListener(....)

```
private fun updateUserStats(userId: String?, startDate: Date?,
endDate: Date?) {
    if (userId == null) return
    val statusNodeMap = mapOf(
        "Accepted" to "acceptedOrders",
        "Delivered" to "deliveredOrders",
        "Incomplete address" to "incompleteAddress",
        "Delivery refused" to "deliveryRefused",
        "Restricted area" to "restrictedArea",
        "Expired" to "expired",
        "Another status" to "anotherStatus"
    )
    val dateFormatter = SimpleDateFormat("dd/MM/yyyy HH:mm",
Locale.getDefault())
    val deliverymanRef =
usersRef.child("User_$userId").child("deliverymanDetails")
    statusNodeMap.forEach { (orderStatus, nodeName) ->
        val query =
ordersRef.orderByChild("orderStatus").equalTo(orderStatus)
        query.addListenerForSingleValueEvent(object :
ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                var orderStatusCount = 0
                snapshot.children.forEach { orderSnapshot ->
                    val order =
orderSnapshot.getValue(Order::class.java)
                    if (order?.deliverymanId == userId) {
                        val orderDeliveredDateString =
order.orderDeliveredDate
                        if (orderDeliveredDateString.isNotBlank()) {
                            val orderDeliveredDate =
dateFormatter.parse(orderDeliveredDateString)
```

```
                            if (orderDeliveredDate != null) {
                                val isInDateRange = when {
                                    startDate != null && endDate !=
null -> {
                                        // Case 1: Date range
specified
dateHelper.isDateInRange(orderDeliveredDate, startDate, endDate)
                                    }
                                    startDate != null && endDate ==
null -> {
                                        // Case 2: Single day
selection
dateHelper.isSameDay(orderDeliveredDate, startDate)
                                    }
                                    else -> true // No dates
specifid, return all orders
                                }
                                if (isInDateRange) {
                                    orderStatusCount++
                                }
                            }
                        }
                    }
                }

deliverymanRef.child(nodeName).setValue(orderStatusCount)
```

Figura 40: Metoda updateUserStats(...)

Concluzii

În concluzie, am dezvoltat această aplicației, Courier Assistant, pentru a optimiza operațiunile de curierat, urmărind să reduc timpul de livrare și să îmbunătățesc gestionarea

comenzilor în depozitele companiilor de livrare. In opinia mea, funcționalitățile sunt adaptate atât pentru livratori, cât și pentru administratori, asigurând o gestionare eficientă și organizată a activităților. Aplicația pune accent pe actualizare în timp real și sincronizarea automată a datelor prin utilizarea serviciilor cloud, dorind sa ofere o experiență fluidă și eficientă pentru toți utilizatorii. Cu toate acestea, consider că aplicația este suficient de scalabilă pentru a fi îmbunătățită cu ușurință și că are potențialul de a evolua pentru a răspunde nevoilor dinamice din industria livrărilor. Aplicația ar putea fi dezvoltată astfel:

- Extinderea unui meniu pentru clienți în care aceștia ar putea comunica direct cu livratorii, pot urmări activ poziția coletelor, pot gestiona preferințele de livrare.
- Analiză avansată a datelor și crearea de rapoarte personalizate
- Integrarea unor soluții AI de predicție
- Globalizare

Bibliografie

1. Kotlin: https://developer.android.com/courses/android-basics-compose/course, https://kotlinlang.org/docs/home.html
2. Courutines:https://kotlinlang.org/docs/coroutines-guide.html#additional-references
3. Android Jetpack: https://developer.android.com/jetpack

4. Jetpack Architecture Components in Android: https://www.geeksforgeeks.org/jetpack-architecture-components-in-android/
5. Maps SDK for Android: https://developers.google.com/maps/documentation/android-sdk
6. Routes API: https://developers.google.com/maps/documentation/routes
7. Geocoding API: https://developers.google.com/maps/documentation/geocoding/overview
8. Realtime Database: https://firebase.google.com/docs/database/android/start
9. What is NoSQL: https://www.mongodb.com/resources/basics/databases/nosql-explained
10. Livedata & MutableLiveData: https://developer.android.com/topic/libraries/architecture/livedata
11. Data Binding: https://developer.android.com/topic/libraries/data-binding
12. RecyclerView: https://developer.android.com/develop/ui/views/layout/recyclerview
13. Activity: https://developer.android.com/guide/components/activities/intro-activities
14. Fragments: https://developer.android.com/guide/fragments, https://www.javatpoint.com/android-fragments
15. Difference between activity and fragments: https://www.geeksforgeeks.org/difference-between-a-fragment-and-an-activity-in-android/
16. NavHostFragment: https://developer.android.com/reference/kotlin/androidx/navigation/fragment/NavHostFragment, https://www.geeksforgeeks.org/bottom-navigation-bar-in-android-jetpack-compose/?ref=ml_lbp
17. Retrofit, HTTP client for Android: https://square.github.io/retrofit/
18. Haversine formula: https://en.wikipedia.org/wiki/Haversine_formula, https://www.movable-type.co.uk/scripts/latlong.html
19. Work with Lists of Data: https://firebase.google.com/docs/database/android/lists-of-data