

UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Aplicație mobilă pentru socializare

Coordonator științific
Lect.dr. Ștefan Popescu

Absolvent
Cioboață Mihai-Cătălin

Bucureşti, iunie 2021

Rezumat

Domeniul aplicațiilor de socializare este relativ nou, fiind responsabil cu oferirea modalităților de comunicare prin intermediul internetului. Datorită popularității în plină creștere, dezvoltarea unei astfel de aplicații este o alegere inspirată dacă se poate identifica corect o nevoie concretă a pieței. Pe această premisă este bazată și prezența lucrarei, fiind descrise pe parcursul primului capitol nevoia identificată, respectiv motivația rezolvării acestei “probleme”.

Capitolul 2 va prezenta funcționalitatea aplicației asociată acestei teze, pentru conturarea unei imagini generale a produsului final, care va servi ca suport practic în capitolul 3, unde vor fi abordatele aspectele tehnice ale acestei lucrări. Pe lângă funcționalitățile prezentate în acest capitol, se oferă și un manual de utilizare al aplicației în care se regăsesc explicații și descrieri ale ecranelor principale, alături de imagini sugestive cu părțile lor componente.

În capitolul 3 se vor aborda principalele modalitățile de dezvoltare ale aplicației, unde vor avea loc discuții despre tehnologiile folosite, dezbatere pentru alegerea celei mai bune soluții în contextul unei probleme și prezentarea unor dificultăți de implementare, alături de rezolvările acestora. Accentul va fi pus pe tehnologiile moderne aferente ecosistemului Android, nefiind discutate și cazurile clasice de utilizare a librăriilor, precum schimbarea unui text sau tratarea apăsării unui buton.

Ultima parte va centraliza progresul obținut pe parcursul realizării acestei lucrări și se va sfârși cu sintetizarea unor concluzii legate de schimbările și îmbunătățirile ce pot fi aduse aplicației în eventualitatea unei continuări a procesului de dezvoltare de după redactarea acestei lucrări.

Abstract

Social media applications are a relatively new field that deals with providing communication capabilities over the internet. Due to its growing popularity, the development of such an application is an inspired choice if a concrete market need can be correctly identified. The present paper is based on this premise, where the identified market need and the motivation to solve this “problem” will be described during the first chapter.

Chapter 2 will present the functionality of the application associated with this thesis, in order to outline a general image of the final product, which will serve as a practical support in Chapter 3, where the technical aspects of this paper will be addressed. In addition to the features presented in this chapter, there is also a user manual of the application which contains explanations and descriptions of the main screens, along with suggestive images about their component parts.

Chapter 3 will address the main ways of developing the application, where there will be discussions about the technologies used, debates on choosing the best solution in the context of a given problem. There will also be a presentation about implementation difficulties and their found solutions. The focus will be on the modern technologies related to the Android ecosystem and there will not be mentions about the classic cases of library usage, such as changing a text or dealing with a button press.

The last part will centralize the progress obtained during the realization of this paper, and it will end with the synthesis of some conclusions related to the changes and improvements that can be brought to the application, in the context of a possible continuation of the development process.

Cuprins

1. Introducere	1
1.1 Motivație	1
1.2 Context	2
2. Prezentare	3
2.1 Punctele cheie	3
2.2 Funcționalitățiile de bază	4
2.3 Ghid de utilizare	6
2.3.1 Autentificare	6
2.3.2 Înregistrare	8
2.3.3 Meniul principal	8
2.3.4 Vizualizare meet-uri	9
2.3.5 Adăugare meet	10
2.3.6 Profilul meu	12
2.3.7 Adăugare autovehicul	14
2.3.8 Căutare	16
3. Dezvoltarea aplicației	19
3.1 Platforma de dezvoltare	19
3.2 Mediul de dezvoltare	21
3.3 Analiza procesului de dezvoltare	22
3.3.1 Alegerea versiunii minime de Android	23
3.3.2 Alegerea arhitecturii	24
3.3.3 Aplicarea arhitecturii în proiect	28
3.3.4 Limbajul de programare folosit	32
3.3.5 Librăriile Android Jetpack	35
3.3.6 Probleme și rezolvări	38
4. Concluzii	46

Lista figurilor

Figura 2.1. Ecranul de autentificare	7
Figura 2.1. Ecranul de înregistrare	8
Figura 2.3. Meniul principal	9
Figura 2.4. Ecranul cu toate meet-urile	10
Figura 2.5. Ecranul unui meet individual	10
Figura 2.6. Alegerea locului	11
Figura 2.7. Adăugarea unei descrieri	11
Figura 2.8. Alegerea datei și a orei	12
Figura 2.9. Rezumatul evenimentului	12
Figura 2.10 Profilul utilizatorului	13
Figura 2.11 Autovehiculele utilizatorului	14
Figura 2.12. Introducerea informațiilor despre autovehicul	15
Figura 2.13. Încărcarea pozei de profil a autovehiculului	16
Figura 2.14. Tastarea numărului de înmatriculare	16
Figura 2.15. Tastarea numărului de înmatriculare	17
Figura 2.16. Ecranul de căutare	18
Figura 2.17. Reselectarea opțiunii curente	18
Figura 2.18. Căutarea unui utilizator	19
Figura 2.19. Căutarea unui autovehicul	19
Figura 2.20. Profilul utilizatorului căutat	19
Figura 3.1. Arhitectura sistemului de operare Android	20
Figura 3.2. Android Studio - Editor vizual + analizator performanță	23
Figura 3.3. Alegerea versiunii minime a SDK-ului	24
Figura 3.4. Arhitectura MVC	25
Figura 3.5. Arhitectura MVP	26
Figura 3.5. Ciclul de viață al ViewModel-ului	28
Figura 3.6. Structura fișierelor Model	30
Figura 3.7. Comparație între două modele	30
Figura 3.8. Gruparea funcționalităților	32

Figura 3.9. Fișierele unei funcționalități	32
Figura 3.10. Kotlin în ecosistemul Android	34
Figura 3.11. Graful de navigare al aplicației	38
Figura 3.12. Graful de navigare între destinații	41

1. Introducere

Auto Club este o aplicație mobilă de socializare destinată posesorilor și pasionaților de automobile, disponibilă pentru sistemul de operare Android. Oferă posibilitatea creării unei identități prin intermediul careia se va interacționa cu alți utilizatori ai aplicației. Conținutul aplicației este orientat către ideea de a găsi alte persoane cu preferințe auto comune, alături de care se vor putea împărtăși impresii și momente în stilul altor aplicații de socializare.

1.1 Motivație

În zilele noastre industria auto este în continuă creștere datorită nevoii cât mai mare de mobilitate, pe care lumea modernă o impune. Printre posesorii de autovehicule se află și un număr considerabil de persoane pasionate de acest domeniu, pentru care autovehiculul personal nu este decât o simplă mașinărie cu scop utilitar. Pentru acești pasionați autovehiculul reprezintă mai degrabă un obiect cu valoare sentimentală, prin intermediul căreia își pot exprima personalitatea și creativitatea.

De asemenea, această evoluție a industriei auto a dat naștere și unui nou fenomen, relativ recent, cunoscut sub numele de “tuning”, care se traduce prin libertatea de modificare a unui autovehicul după bunul plac al posesorului, atât din punct de vedere stilistic, cât și din punct de vedere al performanțelor tehnice. Din cauza lipsei de promovare a acestui trend, foarte mulți pasionați nu dispun de posibilitatea formării unei comunități închegate, în care să se cunoască și să se ajute reciproc, fiind vorba despre un domeniu care nu a trecut încă de stadiul acceptării sociale.

Auto Club a fost creată special atât pentru utilizatorii care deja iau parte acestui fenomen și care doresc cunoașterea altor persoane cu valori comune, cât și pentru utilizatorii care încă nu sunt inițiați, dar care doresc să să se alăture comunității, aplicația servind ca un mijloc de inspirație și de dezvoltare a lor.

1.2 Context

În era digitalizării, aplicațiile mobile de socializare reprezintă un pilon important din viața noastră, fiind un mediu excelent pentru interacționarea cu prietenii, dar și pentru cunoașterea de noi persoane. Peste jumătate din populația lumii folosește aplicații de socializare, dintre care aproximativ 98% aleg să utilizeze un dispozitiv mobil pentru această activitate. Mai mult decât atât, există o creștere consistentă de la an la an a numărului de utilizatori, în 2021 fiind de 13% (490 milioane) față de anul precedent [1].

Când vine vorba despre domeniul auto, există foarte puține aplicații de socializare care pot aduce la un loc toți posesorii de automobile. În căutările făcute de mine pe Google Play, cel mai folosit manager de aplicații pentru telefoanele Android, am descoperit un număr relativ mic de aplicații care să ofere un mediu prietenos și ușor de folosit care să satisfacă nevoile acestei categorii de utilizatori. Cele mai notabile aplicații mobile care aparțin acestei nișe sunt:

- *RoadStr*

Această aplicație mobilă este orientată pe organizarea de evenimente și competiții locale cu specific auto, oferind și posibilitatea utilizatorilor de a se alătura unor cluburi de pasionați, cum ar fi cele destinate posesorilor unei anumite mărci auto. Are 100.000+ instalări.

- *Throdle*

Pe lângă funcționalitățile regăsite și la aplicația *RoadStr*, oferă posibilitatea creării unui “parc auto” unde utilizatorii își pot face cunoscute automobilele lor prin adăugarea de fotografii și detalierea specificațiilor tehnice. Are 500+ instalări.

Cele două argumente de mai sus, alături de motivația prezentată anterior, stau la baza prezentei teze, care își propune dezvoltarea unei aplicații mobile menită să ofere o alternativă la soluțiile existente pe piață.

2. Prezentare

În acest capitol se vor prezenta punctele cheie ale aplicației, alături de funcționalitățile de bază și un ghid de utilizare.

2.1 Punctele cheie

În urma studiului de piață realizat în capitolul anterior, se pot extrage câteva concluzii cu privire la fezabilitatea dezvoltării aplicației *Auto Club*, care se vrea a fi un concurent direct cu cele două aplicații prezentate mai sus. Pentru a decide dacă este relevantă încă o aplicație care se adresează aproximativ aceluiași public țintă, trebuie să se analizeze ce funcționalități vine în plus sau ce probleme rezolvă față de celelalte aplicații deja disponibile pe piață.

Așadar, pentru o poziționare favorabilă pe piață aplicațiilor de socializare destinată entuziaștilor acestui domeniu, *Auto Club* își propune rezolvarea unor probleme semnalate la aplicațiile concurente și anume:

- *Dificultatea găsirii unui anumit autovehicul când utilizatorul dispune de foarte puține informații.*

Probabil că s-a întâmplat să vezi în trafic o mașină care să-ți stârnească interesul, fie prin aspectul unic al acesteia, fie prin performanțele tehnice, după care să dispară prin sutele de participanți la trafic, rămânând în minte doar cu numărul de înmatriculare sau doar cu marca și modelul mașinii respective. *Auto Club* pune la dispoziția utilizatorilor săi căutarea unui autovehicul pe baza anumitor criterii cum ar fi marca, modelul sau numărul de înmatriculare, oferind pe lângă detaliile legate de mașina respectivă și întreg profilul auto al proprietarului, cu care poți interacționa ulterior.

- *Dificultatea organizării unor întâlniri cu ceilalți utilizatori.*

Deși posibilitatea organizării de evenimente este deja disponibilă și în alte aplicații existente, implementarea acestei funcționalități nu este foarte intuitivă, fiind specificate doar coordonatele GPS pentru localizarea evenimentului. Exprimarea poziției în acest mod, presupune o oarecare dificultate pentru utilizatorul de rând, fiind nevoie de un efort destul de mare pentru a găsi locul respectiv. *Auto Club* propune folosirea locurilor disponibile prin Google Maps, ca

formă de localizare, avand în vedere faptul că majoritatea evenimentelor se organizează în locuri publice, înregistrate deja în sistemul de navigare oferit de Google. Metoda folosită va fi foarte ușor de folosit, deoarece sistemul de alegere al locației funcționează aproape la fel cu cel din aplicațiile populare de navigație, cu care majoritatea utilizatorilor au contact frecvent.

2.2 Funcționalitățile de bază

- **Înregistrare:**

La prima interacționare cu aplicația, utilizatorul este rugat să se autentifice, având de ales între folosirea unui cont de Google sau înregistrarea unui cont local. După autentificare urmează alți pași prin care utilizatorul își va crea profilul pe care îl va folosi pe parcursul aplicației.

- **Funcția de căutare:**

Datorită nevoii de cunoaștere a celorlalți membrii ai comunității auto, s-a creat și funcționalitatea de vizualizare a altor profile sau a altor autovehicule. Căutarea este de două feluri:

- *Căutarea unui utilizator după numele acestuia*

Această căutare se va realiza automat în momentul introducerii succesive de caractere, permitând astfel găsirea unor utilizatori după un subșir conținut în numele acestora. Exemplu: Introducerea caracterelor “And” va afișa utilizatorii: Andrei, Andreea și Andrew.

- *Căutarea unei mașini după numărul de înmatriculare*

Asemănătoare cu căutarea de mai sus, în sensul că utilizatorul nu va trebui să tasteze întreg numărul de înmatriculare. Cele mai bune rezultate vor fi afișate în funcție de numărul parțial introdus. Exemplu: Introducerea caracterelor “B555” va afișa numerele de înmatriculare: “B555AUD” și “B555HND”.

Cele două metode de căutare sunt foarte practice pentru cazurile în care nu se cunosc complet detaliile de contact ale unui utilizator, sau numărul de înmatriculare ale unei anumite mașini, fiind sugerate cele mai potrivite rezultate. După terminarea căutării, utilizatorul are posibilitatea să vizualizeze toate detaliile profilelor sau a autovehiculelor găsite.

- **Meet:**

Socializarea efectivă pe care Auto Club o oferă este concentrată în jurul acestei funcționalități. Utilizatorii au posibilitatea creării unor întâlniri fizice într-un loc ales de ei cu ajutorul Google Maps. Astfel, oricine dorește să participe la o anumită întâlnire, are toate detaliile necesare pentru a putea ajunge acolo. Organizatorii de întâlniri pot adăuga și descrieri specifice prin care pot exprima în mod clar scopul unei întâlniri sau prin care pot alege un anumit public către care se dorește a participa. Un exemplu concret ar putea fi organizarea unei întâlniri pentru posesorii de autovehicule ale unei mărci auto (Mazda, Honda, BMW, Audi). Există foarte mulți pasionați auto care dețin un model foarte rar de autovehicul și care nu au posibilitatea planificării unei întâlniri pentru se cunoaște și pentru a împărtășii diverse impresii corelate cu pasiunea lor. Auto Club le satisfac această nevoie și le asigură astfel posibilitatea cunoașterii și a ajutorului reciproc prin schimbul de informații cu caracter vital în supraviețuirea acestor comunități restrânse de pasionați, care duc lipsă de mecanici, piese și indicații de întreținere de care vehiculele lor au nevoie. În continuare se va folosi termenul de “meet” atunci cand se va face referire la această funcționalitate.

- **Garage:**

Aici utilizatorii își pot verifica și modifica propriul profil, în aşa fel încât să reflecte cât mai bine prezența lor în comunitatea auto. Auto Club pune accent pe prezentarea profilului unui utilizator oferind posibilitatea adăugării mai multor mașini la profil, fiecare mașină având o secțiune dedicată în care proprietarul adaugă specificațiile de bază ale respectivei mașini (model, marcă, număr de înmatriculare), plus o descriere personalizată unde se pot trece toate modificările sau detaliile cu caracter definitiv în descrierea vehiculului. Un detaliu menționat destul de des poate fi puterea dezvoltată de vehicul, care de cele mai multe ori în lumea pasionaților auto, este influențată de modificările software sau tehnice realizate ulterior fabricației. Pe lângă parcul auto, profilul conține și detalii personale ale utilizatorului, precum

numele, profilele altor rețele de socializare populare, numărul de urmăritori și numărul de postări.

2.3 Ghid de utilizare

În acest subcapitol se vor prezenta toate componentele aplicației, alături de explicațiile necesare pentru înțelegerea logicii de funcționare. Prin termenul de componentă se înțelege orice funcționalitate care este livrată utilizatorului sub forma unui aspect grafic. Complexitatea unei componente poate varia de la un simplu text, până la un întreg ansamblu de alte componente, care împreună formează componenta respectivă. Un exemplu de componentă complexă este aşa-zisul “ecran”, fiind astfel denumit datorită cantității mari de informație, expusă pe aproape întreg ecranul dispozitivului, cu un rol bine definit și specific. Mai este folosit și termenul de “fereastră” cu același sens. Acești doi termeni vor fi folosiți în continuare pentru a evidenția complexitatea componentei respective și a dimensiunii relativ mari a acesteia, raportată la ecranului fizic al dispozitivului de pe care rulează aplicația.

2.3.1 Autentificare

Prima interacțiune a utilizatorului cu aplicația este marcată de ecranul de autentificare. Procesul de autentificare constă în folosirea unei metode de autentificare:

- **Autentificarea cu contul de Google**

Prin apăsarea butonului de autentificare cu google se va declanșa o nouă activitate în care utilizatorul este rugat să-și introducă datele de conectare aferente contului Google. Acest proces este destul de comun pentru majoritatea utilizatorilor sistemului de operare Android, deoarece este folosit și în alte părți ale ecosistemului Android, cum ar fi în managerul de aplicații Google Play, sau în momentul primei inițializări a dispozitivului, unde este necesar un cont de utilizator Google. Faptul că aspectul și funcționalitatea acestui ecran de conectare nu diferă de cel standard, oferă utilizatorilor încrederea necesară folosirii acestei metode de autentificare, care se dovedește a fi foarte practică și rapidă, având în vedere faptul că majoritatea utilizatorilor au deja configurat un astfel de cont pe dispozitivele lor. Cu atât mai mult, dacă autentificarea a mai fost deja

realizată pe dispozitivul curent, ecranul de introducere a datelor va fi omis, procesul de conectare fiind aproape instant.

- **Autentificarea cu un cont local**

Utilizatorul are și o opțiunea de a se autentifica cu un cont local, introducând email-ul și parola, urmate de apăsarea butonului de logare. Dacă această combinație este greșită sau dacă email-ul introdus nu are asociat un cont, o eroare va apărea pe ecran pentru a informa utilizatorul despre acest lucru. Există și un buton de înregistrare care va duce utilizatorul către ecranul prezentat în secțiunea 2.3.2.

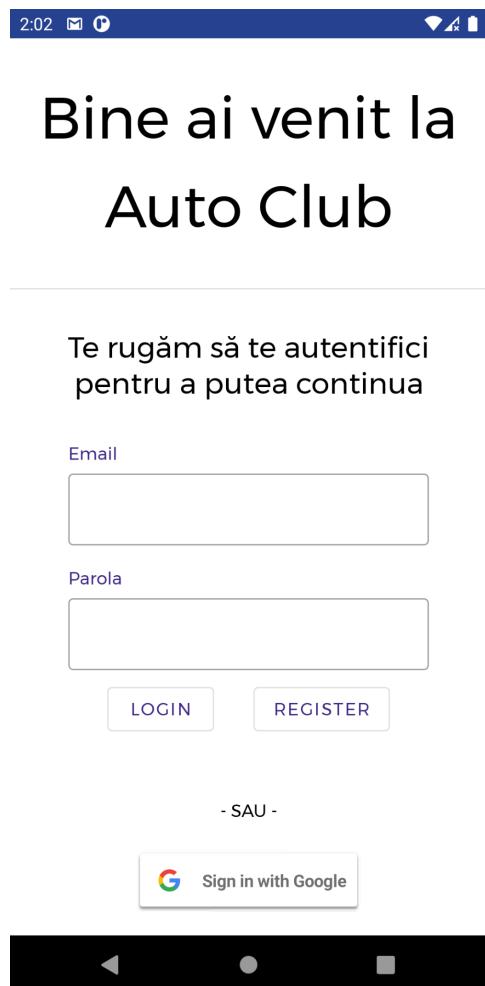


Figura 2.1. Ecranul de autentificare

2.3.2 Înregistrare

Procesul de înregistrare presupune crearea unui cont de utilizator care va fi folosit pe parcursul aplicației pentru accesarea tuturor funcționalităților. Pașii de înregistrare ai unui cont sunt intuitivi, utilizatorul fiind rugat pentru început să-și introducă email-ul personal și să-și aleagă o parolă, această combinație fiind necesară în momentul autentificării de la **2.3.1**.

După introducerea combinației, în cel de-al doilea pas utilizatorul trebuie să-și aleagă un nume prin care va fi identificat de către ceilalți utilizatori ai aplicației. Pasul final constă într-un mic rezumat unde se pot verifica datele introduse anterior. După terminarea acestui proces de înregistrare, utilizatorul va fi redirecționat către aplicația propriu-zisă.

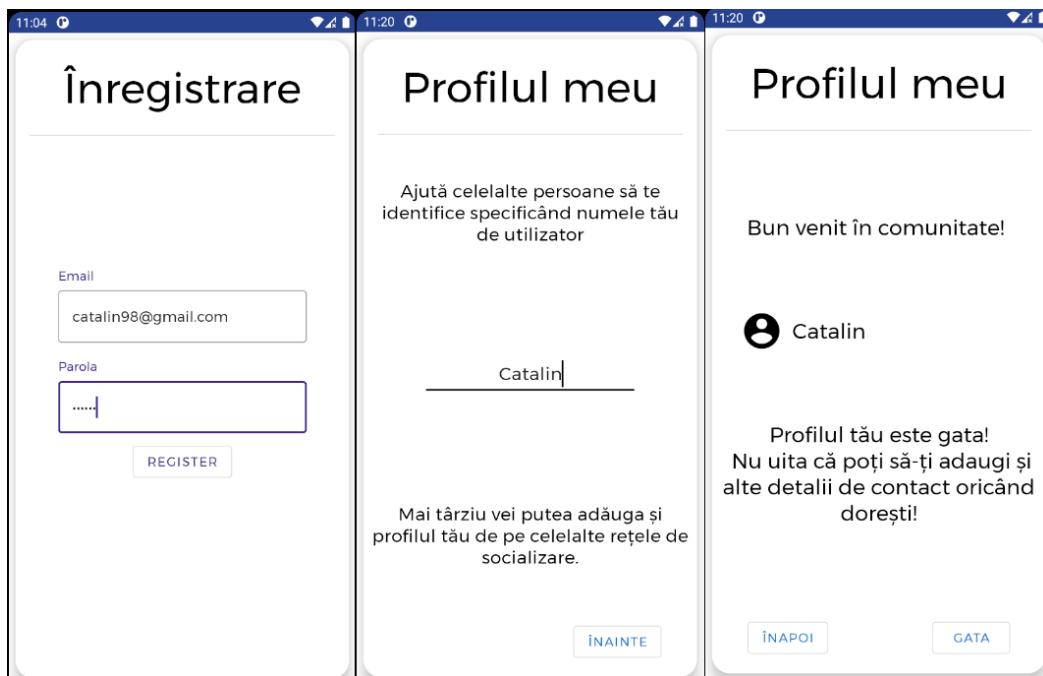


Figura 2.2. Ecranul de înregistrare

2.3.3 Meniul principal

Logica de utilizare a aplicației a fost gândită în jurul unui meniu principal din care se poate ajunge la funcționalitățile de bază ale aplicației, care totodată vor fi și cele mai folosite de-a lungul acestui flux.

Pentru o interacțiune ușoară și intuitivă, meniu de navigare este poziționat în partea de jos a ecranului, fiind permanent vizibil pe parcursul utilizării, mai puțin în momentele în care pe ecran are loc o activitate importantă, de obicei de mărimea întregului ecran, la care utilizatorul este obligat să realizeze o anumită acțiune. Această abordare este foarte folosită în ecosistemul Android, utilizatorul fiind cel mai probabil obișnuit cu acest tip de meniu.



Figura 2.3. Meniul principal

Meniul este compus din mai multe secțiuni, denumite și “tab-uri” în terminologia din limba engleză, fiind asociate unor ferestre care corespund ecranelor principale ale aplicației, descrise în secțiunile **2.3.4**, **2.3.6** și **2.3.8**. La apăsarea unui astfel de “tab”, utilizatorul va fi direcționat către fereastra aferentă fără ca meniul să dispară de pe ecranul dispozitivului.

Navigarea de la o fereastră la alta face ca fereastra inițială (cea din care s-a plecat) să dispară definitiv, orice progres efectuat în această fereastră fiind pierdut definitiv. Astfel, dacă se apasă din nou un “tab” care este deja selectat, fereastra asociată va fi reîncărcată, chiar dacă teoretic va fi înlocuită cu aceeași fereastră. Comportamentul prezentat este impus de acest tip de meniu, unde se presupune că o nouă selecție a ferestrei curente este perfect validă și va declanșa o reîmprospătare a informației.

2.3.4 Vizualizare meet-uri

Această fereastră oferă utilizatorului o listă cu evenimentele create de către toți utilizatorii aplicației Auto Club. Datorită potențialului atingerii unui număr mare de astfel de evenimente, utilizatorul are opțiunea sortării acestora după criteriile: cele mai recente și cele mai apropiate. În funcție de opțiunea de sortare aleasă, lista evenimentelor se va ajusta pentru a satisface criteriul ales.

Fiecare meet este reprezentat de o componentă individuală care afișează cele mai importante caracteristici: numele evenimentului, data, ora, adresa, descrierea și o poză

reprezentativă cu locul ales. Meet-ul poate fi ulterior extins într-o fereastră separată, prin apăsarea acestuia, unde vor fi apărea mai multe detalii, într-o manieră mai clară și “aerisită”.

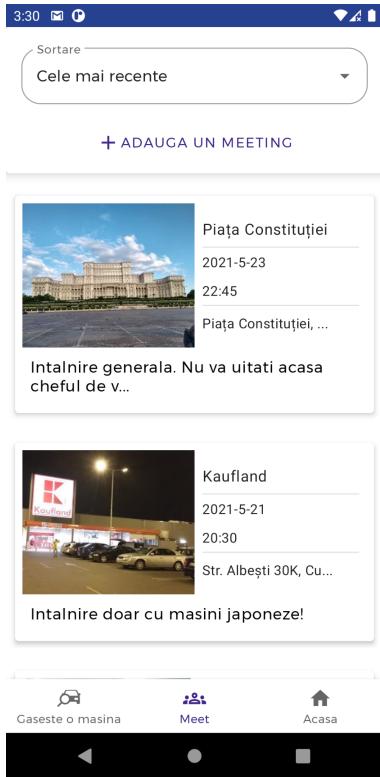


Figura 2.4. Ecranul cu toate meet-urile

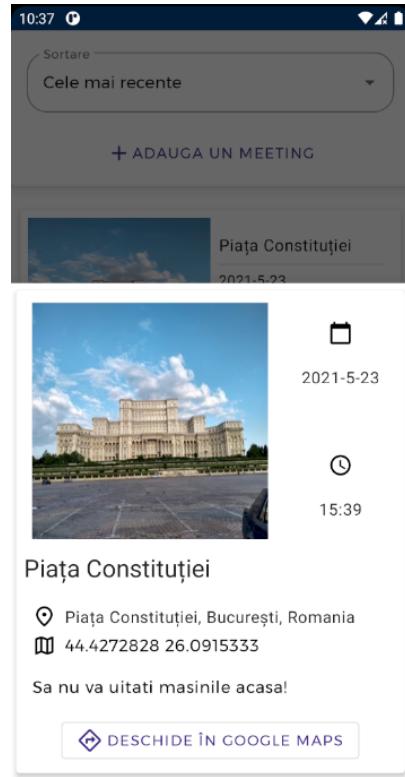


Figura 2.5. Ecranul unui meet individual

Tot în acest ecran se află și opțiunea de adăugare a unui meet nou, prin apăsarea unui buton din partea superioară a ecranului. Procesul de adaugare a unui meet este unul complex și este descris în propriul subcapitol.

2.3.5 Adăugare meet

Procesul de adăugare al unui meet este asemenea unui configurator, în sensul că utilizatorul este rugat să introducă anumite date pe parcursul mai multor pași, fiecare pas fiind o fereastră separată în care sunt cerute anumite informații. La intenția trecerii de la un pas la altul, informația introdusă la pasul curent trece printr-un proces de validare, utilizatorul fiind oprit din a înainta la pasul următor până ce nu rezolvă erorile de validare. Există și posibilitatea revenirii la un pas anterior în orice moment al procesului, precum și anularea acestuia prin apăsarea butonului fizic de mers înapoi al dispozitivului. Concret, adăugarea unui meet este formată din 4 pași:

1. Alegerea locului. Se va deschide o fereastră în care utilizatorul va introduce numele locației. Căutarea va porni automat în momentul introducerii de caractere. Pentru fiecare modificare a caracterelor introduse, serviciul de localizare Google va sugera cele mai bune potriviri, urmând ca utilizatorul să selecteze locul pe care îl dorește. Acest pas este unul obligatoriu.

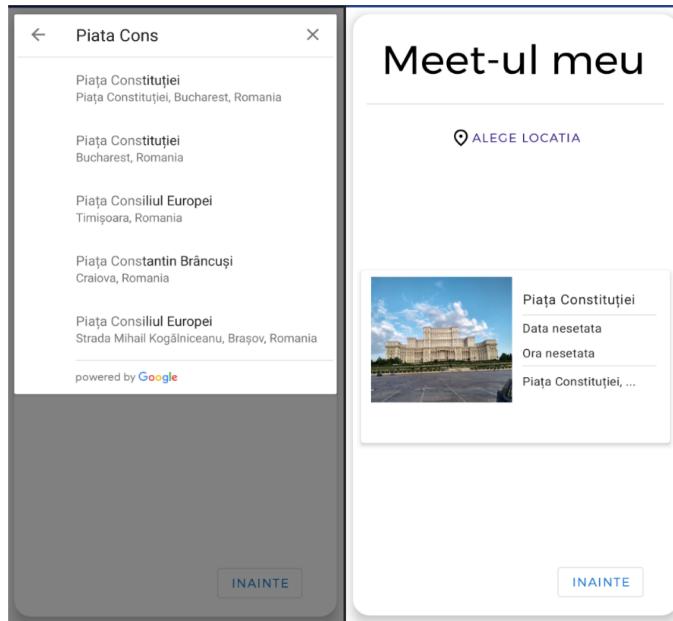


Figura 2.6. Alegerea locului

2. Alegerea descrierii. Introducerea unei descrieri a meet-ului nu este un pas obligatoriu, însă dacă se dorește completarea câmpului, utilizatorul trebuie să se încadreze în maximum 100 de caractere, altfel o eroare va fi afișată și trebuie rezolvată pentru trecerea la pasul următor

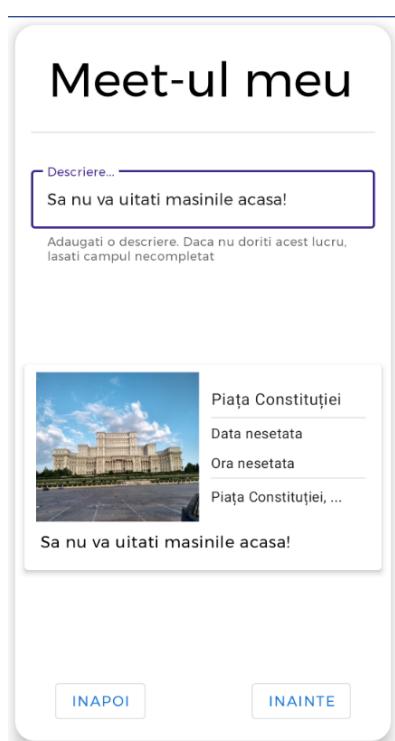


Figura 2.7. Adăugarea unei descriieri

3. Alegerea datei și a orei. Data și ora sunt câmpuri obligatorii, utilizatorul neavând opțiunea de a omite completarea acestora, chiar și parțial (un singur câmp). Se vor alege o dată și o oră prin apăsarea butoanelor asociate celor două câmpuri. După apăsare, o fereastră de selecție modernă și intuitivă se va suprapune peste ecranul curent și va fi închisă doar la confirmarea sau anularea selecției de către utilizator.

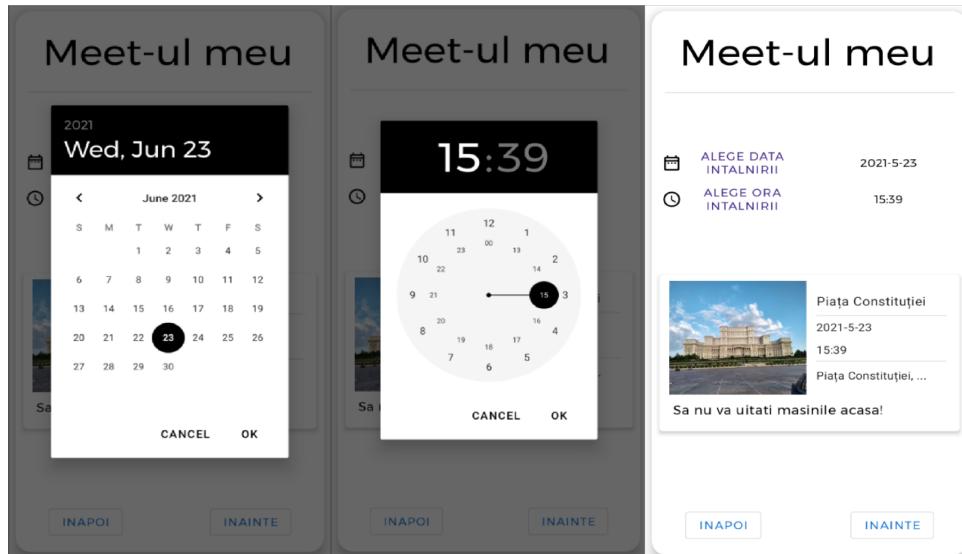


Figura 2.8. Alegerea datei și a orei

4. Finalizare și confirmare. Ultimul pas va reprezenta varianta finală a meet-ului care se dorește a fi creat. Dacă totul este pe placul utilizatorului, acesta poate finaliza întreg procesul prin confirmarea ferestrei curente. Dacă totuși ceva nu este în ordine, se poate reveni oricând la ferestrele anterioare, unde se pot realiza modificările dorite.



Figura 2.9. Rezumatul evenimentului

2.3.6 Profilul meu

Acest ecran este responsabil cu afişarea profilului utilizatorului. Profilul unui utilizator este împărțit în două categorii de informaţii: Date personale şi autovehicule deținute. Accesarea acestor categorii de informaţii se face printr-un meniu de selecţie aflat în partea superioară a ecranului. Componenta asociată acestui meniu de selecţie permite şi trecerea dintr-o categorie la alta prin glisarea orizontală cu degetul în sensul dorit de către utilizator. Cele două categorii de informaţii sunt ilustrate vizual cu ajutorul a două ferestre independente, descrise după cum urmează:

- Fereastra datelor personale:** Numele utilizatorului, data de înscriere în aplicaţie, precum şi alte informaţii personale sunt listate alături de două contoare, care ţin evidenţa numărului de meet-uri organizate şi a numărului de maşini deținute. În partea de jos a ferestrei există un buton de modificare a acestor informaţii, apăsarea acestuia ducând la un nou ecran responsabil de această funcţionalitate.

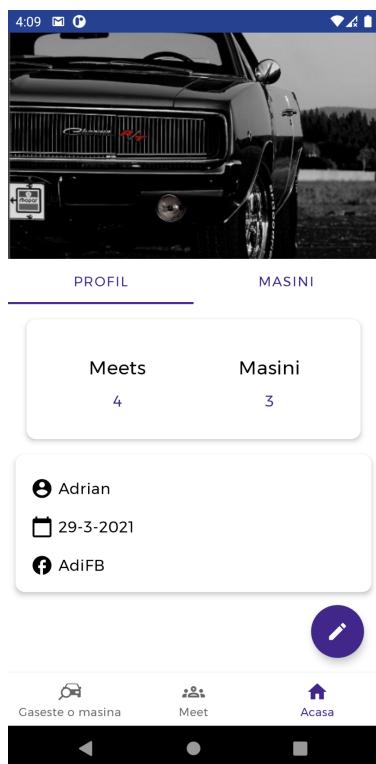


Figura 2.10 Profilul utilizatorului

- Fereastra mașinilor deținute:** Aici utilizatorul își poate vedea mașinile adăugate de el în aplicație, fiind în primă fază afișată o listă cu toate aceste autovehicule. Asemenea ferestrei datelor personale, există un buton în partea de jos a ecranului, doar că de această dată rolul lui este de a adăuga un nou autovehicul. Apăsarea lui duce la pornirea procesului de creație, care este descris în subcapitolul dedicat.

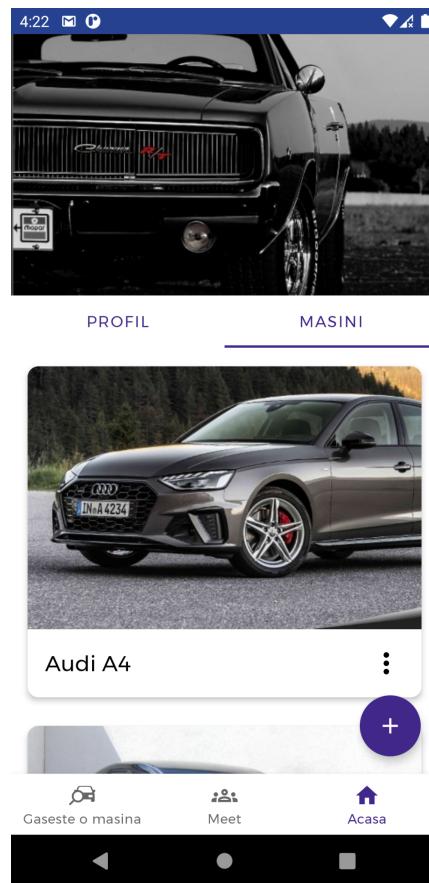


Figura 2.11 Autovehiculele utilizatorului

2.3.7 Adăugare autovehicul

Procesul de adăugare al unui autovehicul este foarte asemănător ca logică cu procesul de adăugare al unui meet, fiind bazat pe aceleași acțiuni, reguli și principii. Diferența este dată în mod evident de specificul acestui proces, pașii de configurare fiind următorii:

1. Introducerea mărcii și a modelului. În câmpurile aferente celor trei informații cerute, utilizatorul este rugat să completeze marca, modelul, respectiv anul fabricației autovehiculului pe care dorește să-l adauge. Introducerea unor informații greșite duce la afișarea unor mesaje de eroare, ca în figura de mai jos.

The figure consists of three side-by-side screenshots of a mobile application interface for entering car details. Each screen has a header 'Mașina mea' at the top.

- Screenshot 1:** Shows three input fields: 'Marca' (Audi), 'Modelul' (A4), and 'Anul de fabricație' (2015). Below the fields is the text 'Completați câmpurile de mai jos'. At the bottom is a blue 'ÎNAINTE' button.
- Screenshot 2:** Shows the same three input fields. The 'Marca' field is highlighted in red with an exclamation mark icon and the error message 'Campul este gol'. The other fields are green with checkmarks. Below the fields is the text 'Completați câmpurile de mai jos'. At the bottom is a blue 'ÎNAINTE' button.
- Screenshot 3:** Shows the same three input fields. The 'Anul de fabricație' field is highlighted in red with an exclamation mark icon and the error message 'Anul trebuie să fie între 1900 2021'. The other fields are green with checkmarks. Below the fields is the text 'Completați câmpurile de mai jos'. At the bottom is a blue 'ÎNAINTE' button.

Figura 2.12. Introducerea informațiilor despre autovehicul

2. Încărcarea unei fotografii. În acest pas utilizatorul este rugat să incare o fotografie cu autovehiculul. Acesta dispune de două moduri de încărcare: folosind camera foto sau folosind aplicația de galerie foto a telefonului. Indiferent de opțiunea folosită, utilizatorul este rugat să ajusteze dimensiunea fotografiei realizate/alese astfel încât să se încadreze în formatul cerut.

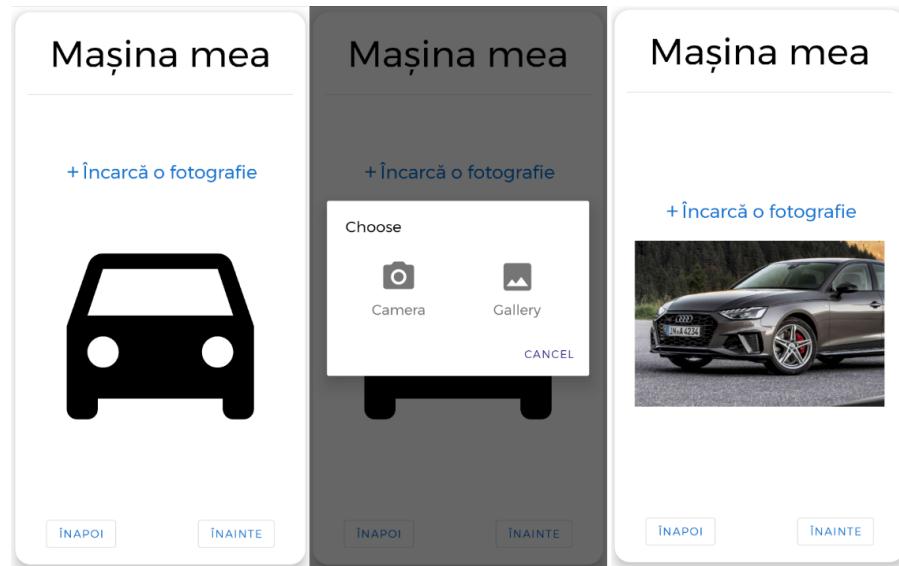


Figura 2.13. Încărcarea pozei de profil a autovehiculului

3. Introducerea numărului de înmatriculare. Această fereastră are un singur câmp și anume numărul de înmatriculare al autovehiculului. Față de celelalte câmpuri cu care utilizatorul este obișnuit, în momentul tastării de caractere câmpul va porni o validare automată a datelor introduse. În cazul unor informații eronate sau indisponibilității numărului de înmatriculare, o eroare va fi afișată pe ecran, fiind obligatoriu de rezolvat pentru trecerea la pasul următor.

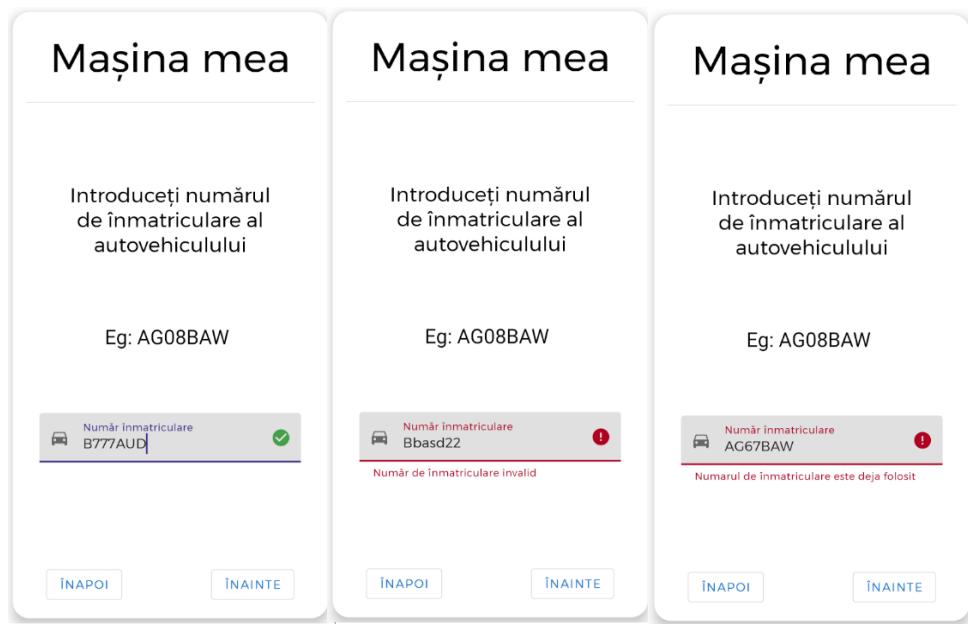


Figura 2.14. Tastarea numărului de înmatriculare

4. Finalizare și confirmare. Ultimul pas reprezintă varianta finală a autovehiculului creat. Dacă ceva nu este în regulă, utilizatorul poate reveni la pașii anteriori pentru a face modificările dorite. Altfel, poate finaliza acest proces, urmând ca autovehicul nou adăugat să fie afișat în lista sa de mașini deținute.

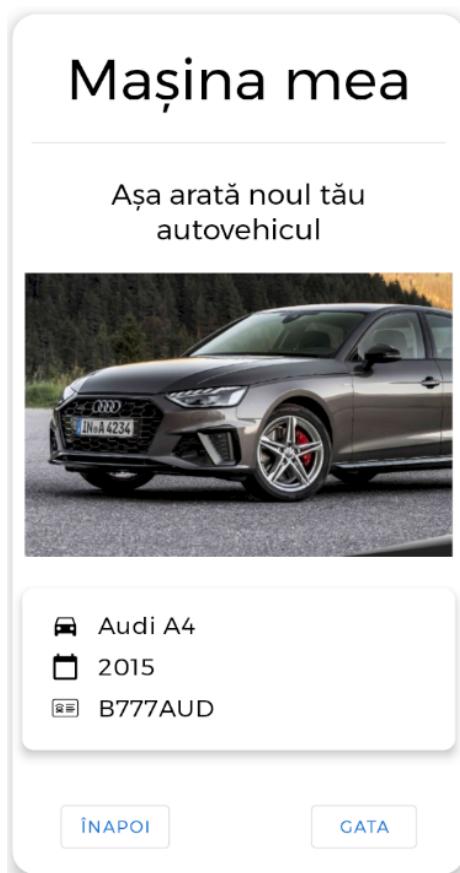


Figura 2.15. Tastarea numărului de înmatriculare

2.3.8 Căutare

Acest ecran este corespunzător funcției de căutare al unui vehicul sau al unui utilizator, aşa cum a fost descrisă în secțiunea 2.2. La prima accesare a ecranului utilizatorul va vedea pe ecran un câmp completabil și două opțiuni de selecție aferente celor două tipuri de căutare. În mod implicit căutarea va fi efectuată pentru găsirea de utilizatori, fiind ușor sesizabil culoarea mai închisă a opțiunii de selecție “Nume utilizator”, precum și prin cerința câmpului de selecție cu același text (figura 2.15).

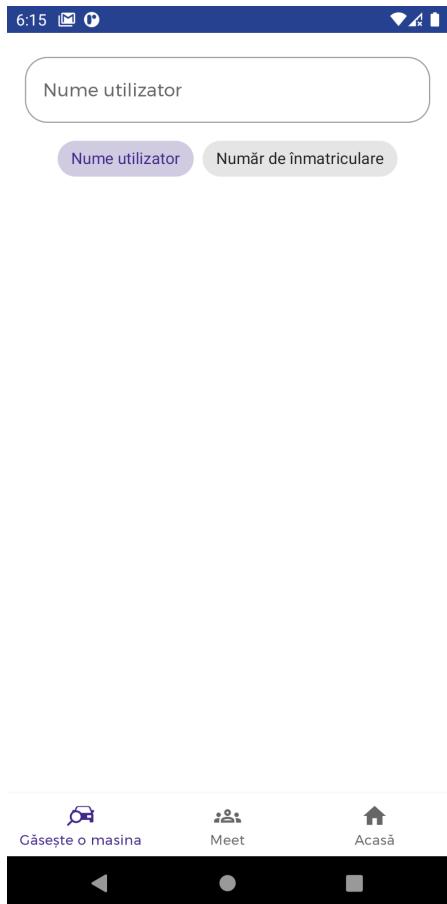


Figura 2.16. Ecranul de căutare

În funcție de opțiunea selectată (nume utilizator sau număr de înmatriculare) și caracterele introduse în câmpul de text, o listă cu rezultate va apărea dacă există autovehicule sau persoane care respectă criteriile alese (figurile 2.17 și 2.18). După afișarea eventualelor rezultate, utilizatorul poate selecta o altă opțiune de căutare, inclusiv cea deja selectată. Realegerea opțiunii curente va face ca acesta să fie deselectată și va ascunde toate informațiile afișate, blocând câmpul de text până ce o opțiune va fi din nou aleasă (figura 2.16). O nouă selecție va însemna de fiecare dată o ștergere a rezultatelor precum și o ștergere a textului introdus în câmpul aferent căutării.



Figura 2.17. Reselectarea opțiunii curente



Figura 2.18. Căutarea unui utilizator



Figura 2.19. Căutarea unui autovehicul

Din meniul de rezultate se poate vizualiza mai departe profilul utilizatorului care prezintă interes, prin apăsarea componentei vizuale a acestuia. În urma acestei acțiuni se va deschide un nou ecran, identic cu cel din secțiunea 2.3.6 (ecranul de home), numai că de această dată informațiile sunt strict corespunzătoare utilizatorului căutat.

Pentru a reveni la lista de rezultate inițială se pot folosi atât butonul fizic de mers înapoi al dispozitivului, cât și săgețica din partea superioară a ecranului.

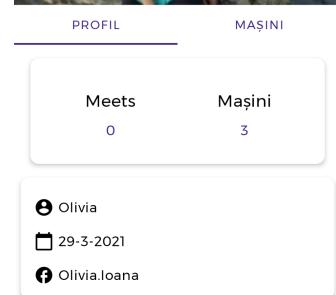
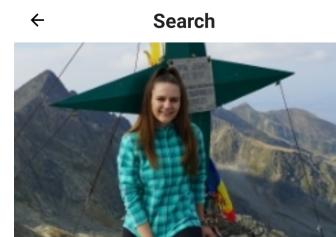


Figura 2.20. Profilul utilizatorului căutat

3. Dezvoltarea aplicației

Acet capitol va aborda toată partea tehnică a acestei lucrări de licență. Ordinea subcapitolelor va urmări etapele dezvoltării, de la alegerea și înțelegerea tehnologiilor până la implementarea diverselor funcționalități și rezolvarea anumitor probleme întâlnite pe parcurs.

3.1 Platforma de dezvoltare

Aplicația Auto Club este construită pentru a fi rulată pe dispozitivele cu sistem de operare Android. Această decizie este fundamentată de numărul foarte mare de utilizatori ai acestei platforme, peste 70% din totalul de dispozitive mobile fiind cu sistem de operare Android [3], dar și de documentația oficială foarte bine structurată, precum și numărul foarte mare de resurse online puse la dispoziția dezvoltatorilor [5].

Arhitectura platformei este bazată pe 5 nivele de abstractizare, fiind construite prin suprapunere astfel încât fiecare nivel să aibă nevoie de cel de “dedesubt”, dar să nu depindă de cel de “deasupra”. Prin urmare cele 5 nivele în ordinea gradului de abstractizare sunt [4]:

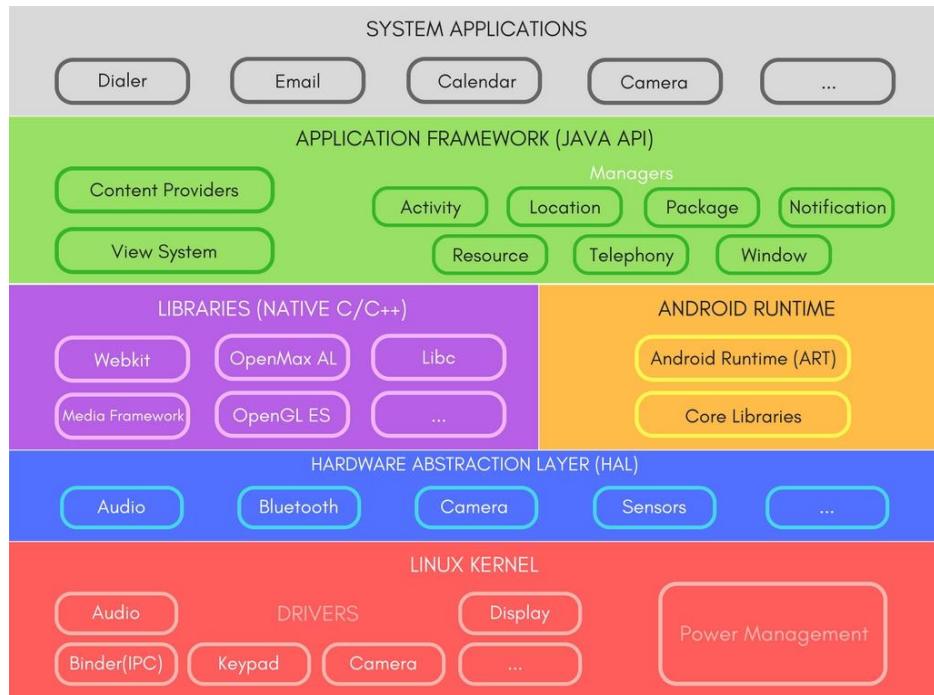


Figura 3.1. Arhitectura sistemului de operare Android [5]

- **Linux Kernel:** Punctul de plecare al Android este bazat pe kernelul Linux, avantajele acestei decizii fiind performanța, siguranța și stabilitatea pe care o Linux le oferă. Mai mult decât atât, fiind un kernel destul de răspândit, producătorii de dispozitive au o viață mai ușoară în dezvoltarea de drivere hardware.
- **Hardware Abstraction Layer (HAL):** Acest nivel de abstractizare oferă o interfață de interacțiune cu componentele hardware ale dispozitivului. Este alcătuit dintr-o colecție de librării de nivel “mic”, fiecare librărie fiind specializată în utilizarea unei componente hardware, cum ar fi camera foto, modulul de conectivitate bluetooth sau difuzorul audio.
- **Android Runtime și librăriile native C/C++:**
 - Începând cu versiunea 5.0 aplicațiile Android sunt proiectate să ruleze în procese separate și în mașini virtuale proprii, prin rularea unor fișiere executabile denumite DEX (Dalvik Executable format), optimizate special pentru folosirea a cât mai puțină memorie posibilă. Pe lângă această eficientizare, ART oferă îmbunătățiri ale performanței prin compilarea AOT și folosirea unui nou garbage collection (GC), precum și optimizarea procesului de dezvoltare și testare a aplicațiilor [6].
 - Sistemele și serviciile Android, inclusiv ART și HAL, sunt implementate din cod nativ, care necesită librării native scrise în C și C++. Pe baza acestora s-au construit și celealte librării de nivel “înalt” ușor de folosit, descrise la nivelul următor, însă dacă se dorește accesarea unor funcționalități mai exclusiviste și mai apropiate de hardware se pot folosi direct aceste librării.
- **Java API Framework:** Întregul set de funcționalități ale sistemului de operare este transpus printr-o metodă foarte convenabilă dezvoltatorului de aplicații și anume printr-un API [7] scris în limbajul de programare Java [8].

Accesul la componentele și sistemele sistemului de operare este realizat prin implementările oferite de către acest API: managerul de resurse, managerul de notificări, managerul de activități, sistemul de componente vizuale (pentru construirea interfeței grafice) etc. Cu alte cuvinte, fără acest nivel de abstractizare nu ar fi posibilă comunicarea cu sistemul de operare într-un mod ușor, sigur și practic pentru dezvoltatorii de aplicații.

- **Aplicațiile de sistem:** Sistemul de operare Android oferă în mod implicit aplicații de bază cum ar fi: camera foto, mesagerie SMS, browser de internet, contacte etc. Aceste aplicații nu au un caracter permanent, putând fi înlocuite cu alte aplicații care să ofere aceleași funcționalități. Din punct de vedere al unui dezvoltator de aplicații, acest nivel din ecosistemul Android este foarte util când vine vorba de nevoia folosirii lor pentru dezvoltarea unei funcționalități, precum cea descrisă în subcapitolul **2.3.5** sau **2.3.7**, unde utilizatorul este rugat să încarce o fotografie, fie din galeria de fotografii, fie instantaneu cu ajutorul camerei foto. Astfel utilizatorul este scutit de implementarea acestor acțiuni, fiind de ajuns doar transmiterea unei astfel de intenții pe care sistemul de operare va știi să o interpreteze și să o servească corespunzător [9].

3.2 Mediul de dezvoltare

Mediul de dezvoltare (IDE [13]) recomandat de către Google pentru dezvoltarea aplicațiilor Android este Android Studio. Fiind bazat pe IntelliJ IDEA, oferă un editor de text excelent, cu funcții avansate de editare și structurare a linilor de cod, însă nu este limitat doar la aceste funcționalități. Android Studio oferă multe unele utile de dezvoltare, în special gândite pentru ecosistemul Android, cum ar fi: sistem de compilare a proiectului bazat pe Gradle, emulator al dispozitivelor Android, şabloane de cod, unelte de testare, analizatoare de performanță și nu în ultimul rând un editor excelent pentru componentele vizuale ale aplicației (figura 2).

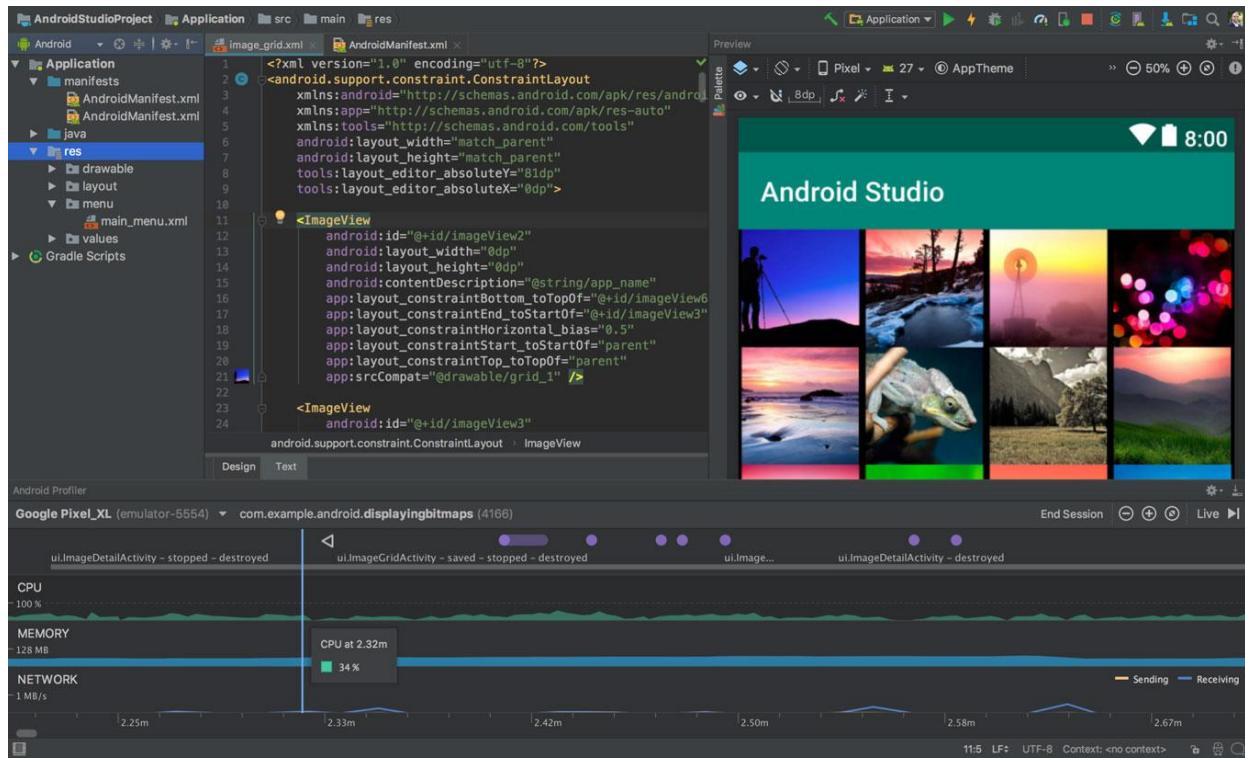


Figura 3.2. Android Studio - Editor vizual + analizator performanță [14]

3.3 Analiza procesului de dezvoltare

Un pas foarte important în proiectarea unei aplicații este analizarea problemei și stabilirea scopului aplicației, pentru a putea planifica încă de la început funcționalitățile de bază. Acest pas este unul necesar, deoarece arhitectura aplicației trebuie gândită astfel încât să se muleze pe cerințele proiectului. Tratarea superficială a acestui pas rezultă în timp și resurse prost investite sau chiar risipite.

Înainte de a se stabili aceste funcționalități ale aplicației, este util să se cunoască puterea tehnologiilor disponibile în momentul proiectării, astfel încât în timpul dezvoltării să nu se ajungă în puncte moarte din cauza unor limitări sau din lipsa “uneltelor” corespunzătoare.

3.3.1 Alegerea versiunii minime de Android

Sistemul de operare Android este într-o continuă schimbare încă de la lansare. De-a lungul existenței, a suferit 30 de nivele de actualizări [10], dintre care 11 sunt versiuni majore în care schimbările au fost notabile. Din prisma dezvoltării, aceste schimbări înseamnă îmbunătățiri ale API-ului, dar și o problemă în plus când vine vorba de compatibilitatea aplicației dezvoltate cu dispozitivele vechi care rulează pe o versiune mai veche a sistemului de operare și care nu au acces la noile facilități aduse de noile versiuni. Ideal ar fi ca o aplicație să fie accesibilă pentru toate dispozitivele aflate în uz, însă din motivele prezentate este imposibil acest lucru dacă se dorește dezvoltarea unei aplicații moderne, atât ca funcționalitate, cât și ca aspect vizual.

Un compromis bun între folosirea unui nivel de API recent și compatibilitatea cu dispozitivele aflate în uz este alegerea unei versiuni minime a sistemului de operare care să afecteze cât mai puțini utilizatori. Ținând cont că un utilizator își schimbă telefonul în medie o dată la doi ani [11], alegerea unei versiuni vechi de acum 4-5 ani ar trebui să fie suficientă, deși calcularea vechimii este decisă și de alți factori precum publicul țintă sau importanța accesibilității.

În momentul creării unui proiect nou în Android Studio există opțiunea alegerei versiunii minime a SDK-ului, alături de o statistică care estimează numărul de dispozitive care sunt compatibile cu versiunea selectată. În cazul aplicației Auto Club, versiunea minimă selectată este API 21: Android 5.0 (Lollipop), aşa cum se poate observa în figura 3, alături de informația că aplicația va rula pe aproximativ 94.1% din dispozitive.

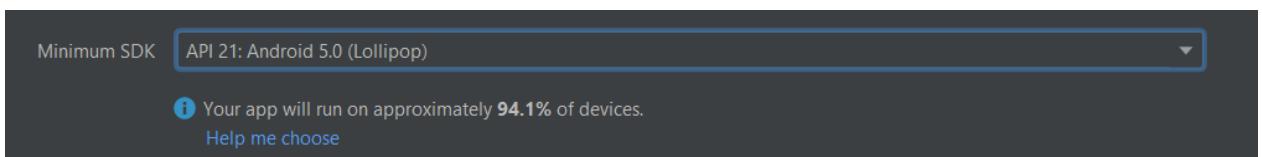


Figura 3.3. Alegerea versiunii minime a SDK-ului.

3.3.2 Alegerea arhitecturii

Când vine vorba despre arhitectura unei aplicații, modelul arhitectural MVC este cel mai răspândit, fiind de altfel și primul şablon gândit să standardizeze dezvoltarea aplicațiilor cu interfață grafică [16]. MVC împarte aplicația în 3 mari componente: Model, View și Controller (de aici și abrevierea).

- **Model.** Responsabilitatea acestei componente este de a stoca și a accesa datele, fiind independentă (complet decuplată) de partea grafică a aplicației. Este identificată ca o componentă de bază deoarece se ocupă și de logica de funcționare a aplicației.
- **View.** Este componenta care se ocupă cu desenarea elementelor grafice pe ecranul dispozitivului, monitorizând în același timp și interacțiunea utilizatorului cu acestea. Poate fi văzută ca o reprezentare a modelului în formă vizuală. În general View-ul nu ar trebui să știe nimic despre logica aplicației, scopul său fiind simplu și bine definit.
- **Controller.** Rolul său este de a lega cele două componente, adică logica aplicației de aspectul acesteia, acționând asemenea unui intermediar. [17][18]

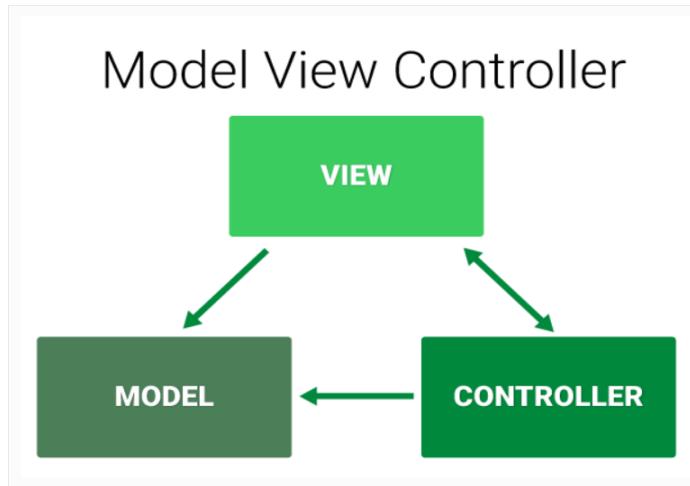


Figura 3.4. Arhitectura MVC [18]

În ecosistemul Android, MVC este regăsit astfel: **modelele** reprezintă clasele de date (locale sau de pe rețea); **view-urile** sunt fișierele de layout, iar **controller-ele** sunt de cele mai multe ori reprezentate de activități și fragmente. Deoarece sistemul Android este unul bazat pe

foarte multă interacțiune din partea utilizatorului, Controllerul devine mai mult o extensie a componentei view. Fiecare componentă vizuală are de cele mai multe ori o parte de logică în Controller, rezultând o legătură foarte strânsă între aceste două componente. O legătură strânsă în contextul arhitecturilor se traduce prin lipsa posibilității de testare a fiecărei componente în parte, dar și prin îngreunarea menținerii unui cod curat și scalabil.

Ca un răspuns la problemele din arhitectura MVC, s-a propus folosirea unei alte arhitecturi și anume MVP. Această nouă abordare propune înlocuirea Controller-ului cu o nouă componentă (Presenter) care să se comporte aproximativ la fel, dar care să nu permită accesul direct la View, de unde va rezulta și decuplarea celor două. Acest lucru este posibil prin considerarea fragmentelor și a activităților ca fiind parte din View, cu condiția existenței unei căi de comunicare, obținută prin crearea unor contract între cele două părți. La nivel de implementare View-ul se angajează să ofere o interfață care să expună metode utile pentru logica internă a Presenter-ului, respectiv Presenterul oferă la rândul lui o interfață prin care să-i comunice View-ului noile date pe care trebuie să le reprezinte vizual.

Model View Presenter

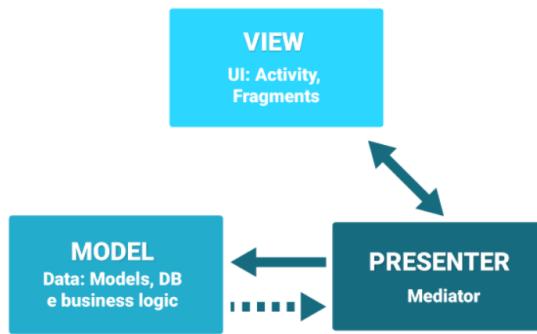


Figura 3.5. Arhitectura MVP [18]

Așa cum este ilustrat și în figura 3.5, în arhitectura MVP Presenter-ul nu îi va da indicații View-ului cum să afișeze informația, ci doar îi va spune ce să afișeze. Această delimitare a responsabilităților oferă modalități de testare independentă a componentelor și de menținere a codului sursă mult mai bune față de arhitectura MVC clasică. Spre exemplu, pentru a testa un View dacă se desenează corect pe ecranele mici atunci cand avem foarte multă informație de afișat, este de ajuns simularea unui Presenter care oferă doar niște cantități mari de informații, View-ul știind de unul singur în ce mod să le așeze pe ecran. Pe cealaltă parte, în arhitectura

MVC ar fi trebuit simulat un Controller căruia pe lângă setul de date de mărimi mari oferit, ar avea nevoie și de o logică a afișării, fiind nevoie astfel de referințe către componentele View-ului. De asemenea, un Presenter oferit de către MVP nu este neapărat asociat unui View anume, putând fi refolosit dacă logica sa este compatibilă și cu alte View-uri, precum și testat independent [19].

Și totuși, arhitectura oficială a aplicațiilor Android este MVVM (Model-View-ViewModel), fiind singura arhitectură care este documentată și implementată nativ în SDK-ul Android [20]. Deși arhitectura MVP este în continuare o arhitectură utilă, MVVM-ul vine ca o derivare a sa, prin reinterpretarea comunicării dintre componenta de date și cea de afișare. Aceste schimbări vin în contextul în care Android a introdus conceptul de “data binding”, fiind o modalitate de a menține componenta View permanent actualizată cu schimbările componentei de date prin observarea schimbărilor unor câmpuri “observabile” [21].

View-ul în arhitectura MVVM rămâne în continuare asemănător ca și concept cu cel din arhitectura MVP, doar că sursa de informație care până acum a fost Presenter-ul este înlocuită cu ViewModel-ul. Această nouă componentă este responsabilă cu pregătirea și prelucrarea datelor “observabile”, dar și cu oferirea de metode publice pe care View-ul le poate apela atunci când se întâmplă un eveniment, cum ar fi apăsarea unui buton. Alte puncte cheie ale ViewModel-ului sunt:

- **Nu are referință către View.** Datorită acestui lucru, ViewModel-ul rezistă mai mult timp în memorie și nu este distrus în momentul schimbărilor din ciclul de viață al componentei vizuale (figura 5). Spre exemplu, la rotirea ecranului deși activitatea va fi recreată, ViewModel-ul va continua să existe, datele furnizate fiind deja pregătite [20].
- **Poate fi împărțit de către mai multe View-uri.** Acest lucru permite utilizarea unui ViewModel în scenariile unde datele care urmează să fie reprezentate sunt aceleași, dar aspectul grafic este diferit. Poate fi folosit și pentru momentele când se dorește salvarea unui progres al utilizatorului care se întinde pe mai multe ferestre, așa cum se întâmplă la **2.3.5** [22].
- **Ușor de menținut și de testat.** Asemenea arhitecturii MVP, MVVM este ușor de testat și de menținut, decuplarea dintre ViewModel și View fiind chiar mai

pronunțată în cazul MVVM deoarece nici măcar nu este nevoie să simulam un View pentru testarea ViewModel-ului.

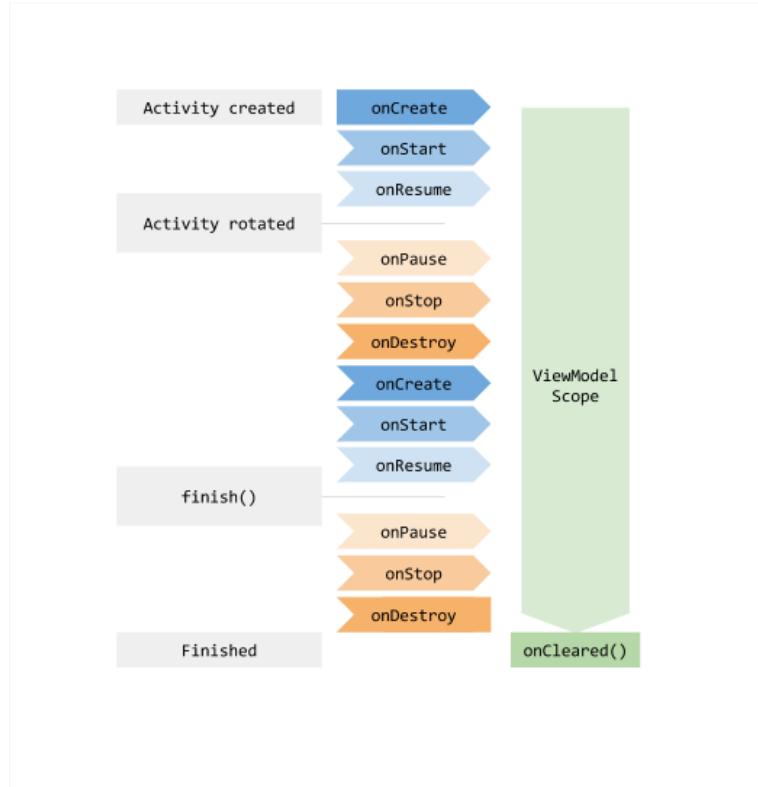


Figura 3.5. Ciclul de viață al ViewModel-ului [22]

În urma trecerii în revistă a tuturor arhitecturilor implementabile în ecosistemul Android, se poate lua decizia folosirii MVVM-ului pentru arhitectura proiectului tratat în această lucrare, deoarece este alegerea cea mai bună în acest moment datorită faptului că este ușor de folosit, ușor de testat și beneficiază de suport oficial din partea ecosistemului Android.

Tot în discuția despre arhitectura proiectului se încadrează și folosirea unui “design pattern” destul de comun în proiectarea componentei de date a unei aplicații și anume “Repository pattern” [23]. Datorită nevoii aplicațiilor de a comunica cu bazele de date pentru stocarea și primirea datelor, nu este o practică bună permiterea ViewModel-ului să obțină singur datele pe care dorește să le prelucreze. Pe lângă faptul că se încalcă principiul responsabilității [24], printre argumentele care stau la baza acestei decizii se enumeră: posibilitatea schimbării mediului de stocare, posibilitatea folosirii unui mediu de stocare în funcție de diversi parametrii (memorie cache), duplicarea codului în mai multe locuri. Așa cum este specificat și în

documentația Android de la [20], componentele arhitecturale care au nevoie de date (ViewModel) nu trebuie să acceseze direct componenta de date (Model), ci trebuie să ceară acest lucru printr-un intermediar (*repository*) care va oferi datele potrivite în funcție de diversi factori. Un exemplu în care acest *repository* este foarte eficient este la încărcarea unei pagini care cuprinde și o parte de informații. La încărcarea paginii, repository-ul poate furniza niște date deja obținute cu o altă ocazie în trecut (din memoria rapidă cache), pentru a nu se oferi senzația că aplicația s-a blocat, urmând ca în fundal să se efectueze o cerere către baza de date aflată la distanță. În momentul în care apelul din fundal s-a încheiat și noile date au ajuns, *repository-ul* va furniza noile date către ViewModel, care mai departe va actualiza View-ul, iar pentru a avea în vedere posibilitatea revenirii utilizatorului pe această pagină în viitor, *repository-ul* poate alege să salveze sau să actualizeze aceste date în memoria rapidă cache.

3.3.3 Aplicarea arhitecturii în proiect

Arhitectura MVVM a dictat structura fișierelor sursă încă de la începutul dezvoltării, componentele fiind regăsite sub următoarea ierarhie:

- **Model.** Directoarele *model* și *repository* (figura 6) formează împreună această componentă. Directorul model conține toate “modelele” aplicației, adică toate entitățile necesare manipulării diferitelor date/obiecte stocate în baza de date. În funcție de specificul manipulării de date a unei funcționalități, un obiect din baza de date poate apărea sub mai multe forme. De exemplu, în momentul listării autovehiculelor unui utilizator, este nevoie decât de informații de bază precum: id-ul, marca, modelul și avatarul (poza de profil) a autovehiculului, nefiind nevoie de descriere, anul fabricației sau colecția de fotografii asociată (figura 7). Această alegere de folosire a mai multor entități pentru reprezentarea diferită a aceluiași obiect este motivată de reducerea memoriei necesare stocării acestor informații, prin eliminarea informațiilor inutile, dar mai ales și din motive de reducere a datelor mobile necesare “descărcării” informațiilor din baza de date de la distanță, lucru care se traduce și printr-o încărcare mai rapidă a ecranelor aplicației. În imaginea de mai jos (figura 6) se pot observa toate formele prin

care entitatea *Car* poate fi reprezentată: *Car* (entitatea de bază aşa cum este regăsită în baza de date), *CarCreationModel* (folosită în procesul de creare a unui autovehicul), *CarDetailsModel* (foarte asemănătoare cu entitatea de bază, doar că unele câmpuri sunt prelucrate pentru un acces mai facil), *CarPhotoModel* (folosită în afişarea colecției de fotografii), *CarProfileModel* (folosită în scenariul din exemplul anterior).

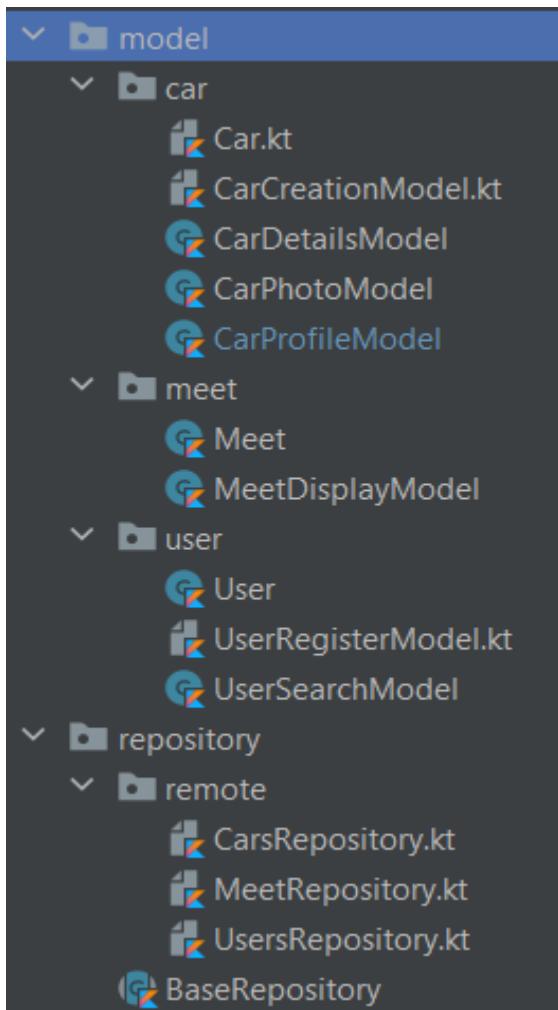


Figura 3.6. Structura fișierelor Model

The screenshot shows two code editors side-by-side:

- Car.kt** (top editor):


```
12
13     data class Car(
14         var id: String? = null,
15         var make: String? = null,
16         var model: String? = null,
17         var year: Int? = null,
18         var numberPlate: String? = null,
19         var ownerUid: String? = null,
20         var avatarUri: String? = null,
21         var photosUri: List<String>? = null,
22         var description: String? = null
23     )
```
- CarProfileModel.kt** (bottom editor):


```
6
7     data class CarProfileModel(
8         var id: String?,
9         var make: String?,
10        var model: String?,
11        var avatarDownloadLink: Uri?
12    )
```

Figura 3.7. Comparație între două modele

Pentru interacțiunea cu mediul de stocare a datelor necesare aplicației, s-a folosit “*Repository pattern*”, aşa cum a fost explicitat la secțiunea 3.3.2. În funcție de specificul interacțiunii, s-au creat mai multe repository-uri care să trateze problemele legate de

obținerea, modificarea, salvarea și ștergearea fiecărei entități în parte. Astfel, repository-urile obținute sunt: *CarsRepository* (manipulează entitatea *Car*), *MeetsRepository* (manipulează entitatea *Meet*), *UsersRepository* (manipulează entitatea *User*). Metodele de acces implementate de către aceste clase sunt de asemenea specificate în interfețele corespunzătoare fiecărui repository, pentru a putea abstractiza modul în care implementarea a fost făcută atunci cand avem nevoie să testăm sau să “injectăm” o instanță concretă. Un exemplu de interfață este cea a repository-ului de autovehicule.

```
interface ICarsRepository {
    suspend fun addCar(car: Car): Car
    suspend fun setAvatar(id: String, photo: Bitmap)
    suspend fun getCarsByUserId(uid: String): List<Car>
    suspend fun getCarsByUserIdAsFlow(uid: String):
        Flow<State<List<Car>>>
    suspend fun getCarById(id: String): Car?
    suspend fun addPhoto(carId: String, bitmap: Bitmap)
    suspend fun addDescription(carId: String, description:
        String?)
    suspend fun deletePhoto(carId: String, photoUri: String)
    suspend fun deleteCar(carId: String)
}
```

- **View.** În arhitectura MVVM, activitățile și fragmentele, alături de fișierele de layout, sunt considerate parte din componenta vizuală. Prin natura ecosistemului Android, bazat în special pe interacțiunea utilizatorului cu componentele vizuale, cea mai mare parte din codul sursă este destinată acestei componente MVVM. Datorită acestui fapt, structura logică a aplicației este organizată pe funcționalități (figura 8). Mai exact fiecare funcționalitate are propriul director de fișiere, fiind foarte ușor și intuitiv de găsit fișierele sursă atunci cand se dorește modificarea unor anumite lucruri ce țin de o anumită funcționalitate (figura 9). Toate aceste directoare sunt conținute de un director părinte numit “*UI*” (de la englezescul “User Interface”), care mai conține și alte componente utile ce nu pot fi încadrate într-o funcționalitate anume, dar care aparțin tot de partea grafică, cum ar fi componentele din directorul “*custom*”.

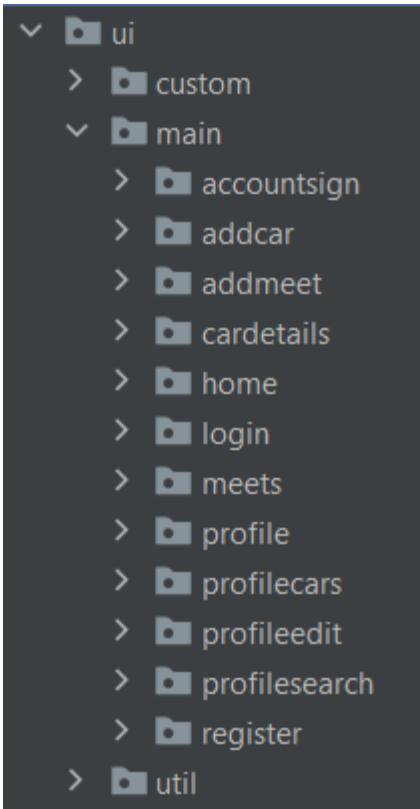


Figura 3.8. Gruparea funcționalităților

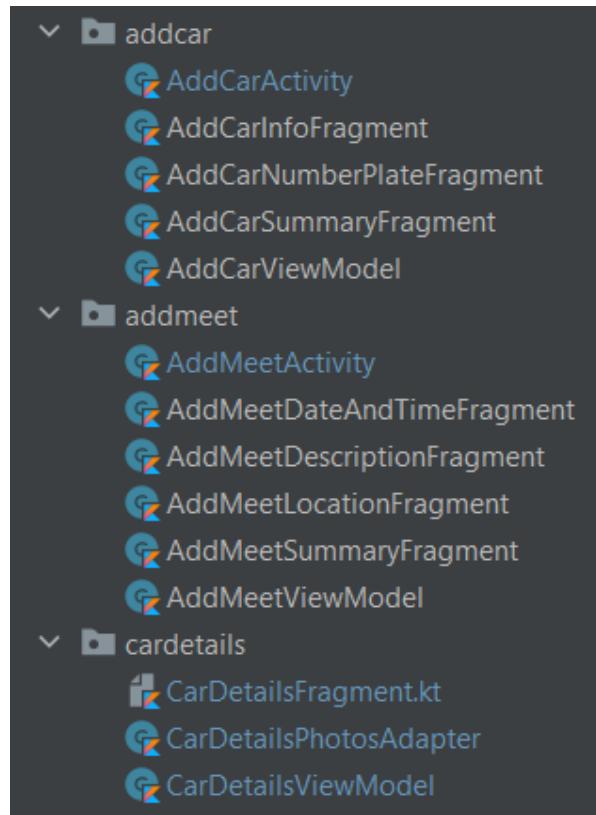


Figura 3.9. Fișierele unei funcționalități

- ViewModel.** În directorul de fișiere ale unei funcționalități (figura 9) se pot observa și apariția unor fișiere cu sufixul “*ViewModel*”. Aceste fișiere sunt practic clasele *ViewModel* oferite de API-ul Android și extinse pentru a servi informațiile necesare componentelor vizuale. Cum aceste componente vizuale sunt strâns legate de funcționalitățile aplicației, în consecință și un *ViewModel* va face același lucru. Astfel, decizia de a grupa *ViewModel*-urile în directoarele funcționalităților este una destul de naturală. În general un *ViewModel* este asociat unui singur *View*, aşa cum se întâmplă în aproape toate cazurile în care există doar un *View* pentru o funcționalitate (cum ar fi ecranele de la secțiunile 2.3.4, 2.3.6 și 2.3.8), însă există și scenarii unde acesta poate fi partajat de mai multe *View*-uri.

3.3.4 Limbajul de programare folosit

În proiectarea oricărui produs software este nevoie de găsirea unui limbaj de programare care să satisfacă cerințele de proiectare cât mai bine. De cele mai multe ori criteriile alegerii limbajului de programare sunt restrâns de platforma pentru care se dezvoltă acel produs software. Acest lucru este valabil și pentru aplicațiile Android, unde până nu de multă vreme singurul limbaj de programare ce se putea folosi în dezvoltare era **Java**. Acest alegere a fost făcută de către Google din mai multe motive:

- Este în top 5 cele mai folosite limbi de programare din 2020. [30]
- Poate rula pe orice dispozitiv.
- Android Runtime este bazat pe mașina virtuală Java (JVM).
- Este orientat pe obiecte.
- Dispune de foarte multe librării pe lângă cele din ecosistemul Android. [29]

În secțiunea 3.1 se menționează faptul că ecosistemul Android oferă dezvoltatorilor un SDK scris în Java pentru accesarea tuturor funcționalităților sistemului de operare, dar și librării scrise în C/C++ care sunt mai mult specializate pe interacțiunea de nivel “mic” cu sistemul de operare. Așadar, singura opțiune viabilă pentru dezvoltarea aplicațiilor a fost multă vreme limbajul de programare Java.

În februarie 2016, o nouă întorsătură avea să ia loc printre membrii comunității de dezvoltatori ai Android. Se lansează prima versiune a limbajului de programare **Kotlin** [31]. Creat cu scopul de a oferi o alternativă modernă pentru clasicul limbaj de programare Java, câștigă foarte multă simpatie din partea dezvoltatorilor pe care reușește să îi cucerească cu noile sale funcționalități:

- ➔ Combină paradigma limbajului orientat pe obiecte cu paradigma funcțională.
- ➔ Este 100% compatibil cu cod Java. Orice cod Java poate fi folosit de către un cod scris în Kotlin și reciproc.
- ➔ Liniile de cod scrise în Kotlin sunt cu aproximativ 40% mai puține decât echivalentul din Java.

- Oferă tipuri de date “*non-null*able” pentru a trata direct din faza de compilare posibilele întreruperi de sistem care apar din neînțelegerea corectă a tipului de date *null* (regăsit și în Java).
- Se pot folosi funcții de nivel înalt care nu necesită apelarea lor prin intermediul unui obiect, față de Java unde funcțiile reprezintă doar metodele unui clase.
- Clasele deja implementate pot fi “extinse” prin implementarea așa-numitelor “extension functions” [31]
- Etc.

Datorită compatibilității perfecte cu codul scris în limbajul de programare Java, limbajul Kotlin a fost foarte ușor adoptat de către dezvoltatori, fiind încurajați și de decizia Google din anul 2019 de a declara ecosistemul Android “Kotlin-first” (limbajul de programare principal în ecosistemul Android este Kotlin) [32]. Această decizie s-a tradus prin folosirea Kotlin în scrierea documentațiilor oficiale și în dezvoltarea noilor librării și tehnologii.

	Java language	Kotlin
Platform SDK support	Yes	Yes
Android Studio support	Yes	Yes
Lint	Yes	Yes
Guided docs support	Yes	Yes
API docs support	Yes	Yes
AndroidX support	Yes	Yes
AndroidX Kotlin-specific APIs (KTX, coroutines, and so on)	N/A	Yes
Online training	Best effort	Yes
Samples	Best effort	Yes
Multi-platform projects	No	Yes
Jetpack Compose	No	Yes

Figura 3.10. Kotlin în ecosistemul Android

O funcționalitate importantă a limbajul Kotlin care nu a fost menționată în lista de mai sus este posibilitatea folosirii de *coroutine* pentru programarea asincronă. Este un concept foarte asemănător cu funcționalitățile de “*await*” și “*async*” ale multor limbi de programare moderne. Oferă o formă de execuție de cod suspendabilă, fiind echivalentul unui *thread* din sistemele de operare, dar doar ca și concept de funcționare. Codul executat de o corutină nu este neapărat executat pe un *thread* separat față de *thread*-ul din care corutina a fost lansată, fiind ca implementare mult mai eficient și ușor de creat față de un *thread* real al sistemului de operare. Este posibilă lansarea a 100.000 de corutine fără ca sistemul să primească vreo eroare fatală [33].

Pentru a crea o corutină este nevoie de crearea unui context în care corutina să poată rula, fiind numit și sferă de acțiune, sau “*scope*” în terminologia din limba engleză. Acest context are rolul de a oferi corutinei informații despre mediul de rulare și ciclul de viață al acesteia. Aceste informații sunt utile pentru a oprirea apelul asincron atunci când acțiunea inițială devine irelevantă datorită distrugerii contextului. Un exemplu concret de context este *ViewModelScope*, oferit implicit de clasa *ViewModel* din SDK-ul Android. Acest context este limitat la durata de viață a *ViewModel*-ului, orice apel asincron care este lansat și nu-și termină execuția fiind automat abandonat și distrus. Acest context este cel mai des folosit context în dezvoltarea aplicației Auto Club, deoarece toate apelurile asincrone, în general către baza de date de la distanță, sunt realizate în componenta *ViewModel*. Spre exemplu, pentru a primi datele unui utilizator al aplicației în funcție de identificatorul acestuia, se folosește următorul cod:

```
@Inject -> (1)
constructor(
    private val usersRepository: IUsersRepository
) : ViewModel() {

    val userLiveData:
        MutableLiveData<User> = MutableLiveData(); -> (2)

    fun getUserByUid(uid: String) {
        viewModelScope.launch { -> (3)
            val user = usersRepository.getUserByUid(uid)
            if(user != null)
                userLiveData.postValue(user)
        }
    }
}
```

La linia de cod (3) se lansează corutina care conține liniile de cod din interiorul acoladelor, execuția fiind executată asincron și este dependentă de contextul ViewModel-ului din care corutina a fost lansată. Dacă presupunem că utilizatorul dorește închiderea ferestrei de informații despre utilizatori, ViewModel-ul va fi distrus, declanșând la rândul lui închiderea corutinei proaspăt lansate. Liniile de cod (1) și (2) sunt specifice altor tehnologii explicate în secțiunea următoare, care sunt folosite foarte des în combinație cu aceste corutine.

3.3.5 Librăriile Android Jetpack

S-a menționat de mai multe ori pe parcursul acestei lucrări despre uneltele oferite dezvoltatorilor în procesul de dezvoltare a aplicațiilor Android. O componentă importantă din acest SDK este reprezentată de existența unei colecții de librării regăsite sub numele de “*Android Jetpack*”. Această colecție de librării a fost proiectată să ofere ajutor pentru cele mai întâlnite probleme ce intervin pe parcursul procesului de dezvoltare a aplicațiilor, oferind în același timp garanția funcționării corecte și consistente indiferent de versiunea sistemului de operare și a tipului de dispozitiv folosit. În procesul dezvoltării aplicației Auto Club s-au folosit multe componente din această colecție, urmând să se prezinte doar cele mai importante dintre acestea.

❖ LiveData

LiveData este un container de date ce implementează pattern-ul Observer, funcționând asemenea unei surse de date care notifică toți ascultătorii despre eventualele modificări ale valorilor stocate în container [36]. Diferența între LiveData și un o clasă obișnuită ce implementează pattern-ul Observer este data de proprietatea de a emite notificări doar atunci când ascultătorii sunt într-un ciclu de viață favorabil (în stările *Started* sau *Resumed* specifice Android). Când un ascultător nu mai este disponibil (este în starea *Destroyed*), acesta va fi eliminat în mod automat din lista ascultătorilor [27]. Avantajele folosirii acestei librării sunt:

- ➔ Asigură o împrospătare automată a interfeței grafice atunci când se produce o modificare.
Nu mai este nevoie de cod specific pentru actualizarea elementelor grafice.
- ➔ Nu se produc scurgeri de memorie datorită eliminării automate a ascultătorilor inactivi.
- ➔ Nu există riscul ca aplicația să se întrerupă în timpul transmisiunii de date.
- ➔ Scapă dezvoltatorul de grija schimbărilor din ciclul de viață al fragmentelor/activităților.

- După revenirea un ascultător din starea de pauză, acesta va primi în mod automat ultimele actualizări disponibile.

❖ Data Binding

Această librărie permite “legarea” componentelor grafice la o sursă de informație, direct din fișierul de layout (locul unde se declară elementele grafice ale unui fragment sau ale unei activități). Cel mai folosit scenariu unde această librărie se arată a fi cu adevărat utilă, este legarea componentelor grafice, care au și o afișare de tip casetă de text, la o sursă de informație provenită dintr-un ViewModel. Pentru a putea oferi acest comportament, librăria generează automat, în timpul compilării, câte o clasă specifică fiecărui fișier de layout, toate elemente grafice fiind astfel transpusă în cod Kotlin. [37]

Pentru a marca un fișier de layout ca fiind capabil de “data binding”, se cuprind toate liniile fișierului între tag-urile “`<layout>`” și “`</layout>`”. Sursa de informație este specificată tot în cadrul acestui fișier, prin tag-urile “`<variable>`” și “`</variable>`”, alături de câmpurile care se doresc să fie accesate, specificate la rândul lor în dreptul locului unde se doresc să fie “legate”. Un exemplu minimal al acestei librării este următorul:

```
<layout...>
    <data>
        <variable
            name="myViewModel"
            type="com.myApplication.UserViewModel" />
    </data>
    <TextView ...
        android:text="@{myViewModel.userFirstName}" />
    <TextView ...
        android:text="@{myViewModel.userLastName}" />
</layout>
```

În acest exemplu sursa de informație este UserViewModel, reprezentat prin numele de variabilă “myViewModel”. Presupunem că în cadrul acestui ViewModel am avea două proprietăți publice “`userFirstName`” și “`userLastName`” pe care dorim să le afișăm în două componente vizuale de tip `TextView`. Pentru a realiza acest lucru se trec cele două variabile în dreptul proprietăților de text ale componentelor vizuale, urmând ca librăria să genereze tot codul necesar acestei acțiuni, interacțiunea utilizatorului fiind astfel minimală. Pentru a diferenția

numele sursei de informație de un text normal, este obligatorie folosirea caracterelor speciale “{@{ }”, după care se poate scrie între cele două accolade numele variabilei care se vrea să fie folosită. Se pot folosi de asemenea și expresii sau funcții în interiorul celor două accolade, libraria putând să folosească chiar și pentru “legarea” inversă, atunci când se specifică o funcție în dreptul câmpurilor de eveniment ale elementelor grafice. Un exemplu ar fi notificarea ViewModel-ului de bifarea de către utilizator a unei căsuțe, prin apelarea unei metode fictive “onBoxChecked”.

```
<CheckBox
    ...
    android:onCheckedChanged="@{myViewModel.onBoxChecked()}"
/>
```

❖ Navigation

Este o librărie creată cu scopul de a rezolva problema navigării utilizatorului prin ecranele aplicației. Oferă un mod organizat de a specifica toate destinațiile posibile, precum și legăturile dintre acestea. Asemenea librăriei *Data Binding*, pentru a ușura munca dezvoltatorilor se va ocupa de toată generarea de cod Kotlin, punând la dispoziție niște acțiuni pe care utilizatorul le va putea folosi atunci când dorește navigarea către o anumită destinație. Aceste acțiuni sunt chiar legăturile dintre fragmente specificate de către utilizator cu ajutorul unui fișier XML [2].

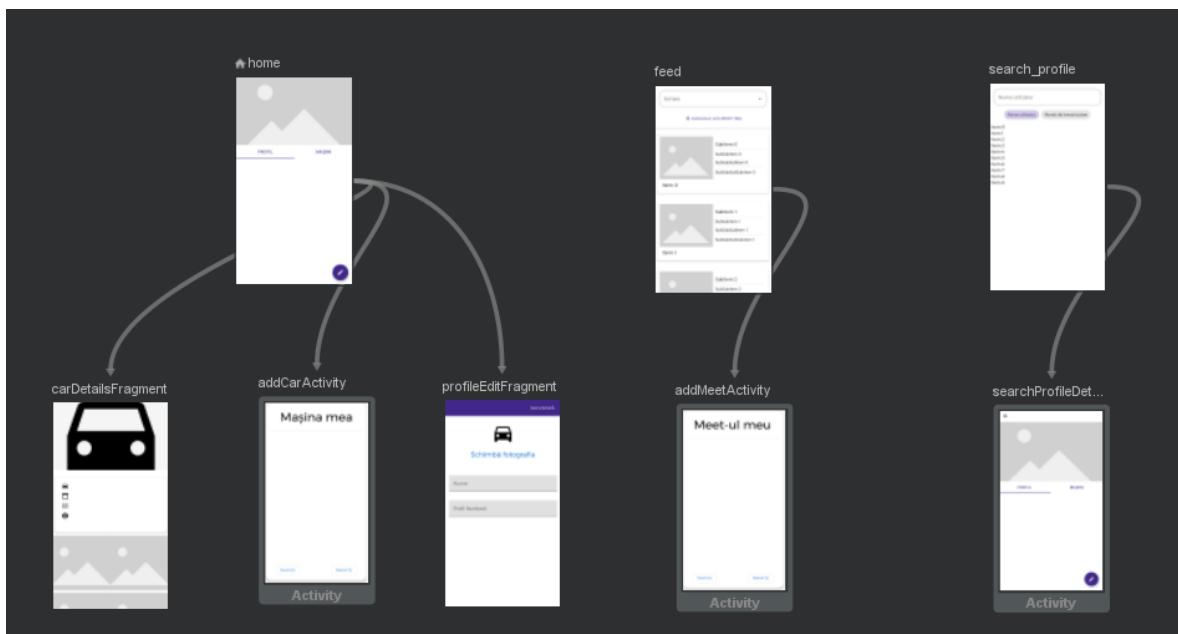


Figura 3.11. Graful de navigare al aplicației

În figura 3.11 se pot observa toate destinațiile și direcțiile asociate meniului principal. Cele 3 ferestre “*home*”, “*feed*” și “*search_profile*” nu au nevoie de un părinte comun deoarece librăria oferă posibilitatea sincronizării meniului principal cu graful de navigare. Gesturile utilizatorului de a reveni la fereastra anterioară sunt de asemenea sincronizate cu funcționalitatea acestei librării, fiind foarte ușor de gestionat aceste schimbări de ecrane de către dezvoltator, față de metoda clasică a tranzacțiilor cu fragmente. Librăria oferă și alte funcționalități foarte utile cum ar fi posibilitatea transmiterii de argumente către destinații prin folosirea extensiei “Safe Args”, care generează cod Kotlin special pentru lucrul cu argumente. Este o funcționalitate foarte importantă deoarece se elimină pașii variantei clasice în care era nevoie de o atribuire a unor identificatori unici care să localizeze argumentele trimise și primite. Vechea metodă reprezintă un risc deoarece din neatenția dezvoltatorului se pot folosi identificatori diferiți în momentele primirii și a trimiterii de argumente, aplicația urmând să se închidă neașteptat din cauza accesării unui element nonexistent. Un exemplu de folosire a acestei funcționalități poate fi văzută în secțiunea 3.3.6, unde se pasează un argument între două destinații.

3.3.6 Probleme și rezolvări

În această secțiune vor fi menționate diverse probleme întâmpinate în timpul dezvoltării aplicației, alături de soluțiile propuse și eventualele implementări.

Prima problemă cu care m-am confruntat a fost oferirea unor informații permanent actualizate pentru afișarea lor pe ecranul utilizatorului. Această problemă a apărut în momentul implementării funcționalității de editare a profilului, unde după terminarea modificărilor și salvarea acestora, informațiile despre utilizator rămâneau neactualizate în fereastra de detaliu a ecranului de “Home”. În urma documentării am aflat faptul că librăria Data Binding se poate folosi în combinație cu librăria LiveData (secțiunea 3.3.5), combinând ușurința legării unei componente vizuale de o sursă de informație cu proprietatea unei informații de a fi actualizată permanent. Pașii de implementare au fost următorii:

1. Folosirea LiveData în ViewModel pentru stocarea informației

```
val userLiveData: MutableLiveData<User> = MutableLiveData();
```

2. Specificarea sursei de informație în câmpurile din layout, alături de folosirea unor funcții din clasa *String* pentru convertirea numerelor întregi la un tip de date *String*.

```
<layout ...>
    <data>
        <variable
            name="viewModel"
            type=
"com.catasoft.autoclub.ui.main.profile.ProfileViewModel" />
    </data>

    ...
    <TextView
        ...
        android:id="@+id/tvDisplayName"
        android:text="@{viewModel.userLiveData.name}" />

    <TextView
        ...
        android:id="@+id/tvJoinDate"
        android:text="@{viewModel.userLiveData.joinDate}"
        android:textAppearance="?attr/textAppearanceBody1"/>

    <TextView
        ...
        android:id="@+id/tvFacebookProfile"
        android:text="@{viewModel.userLiveData.carsCount}"/>

    <TextView
        ...
        android:id="@+id/tvCarsCount"
        android:text=
"@{String.valueOf(viewModel.userLiveData.carsCount)}"/>
```

```
<TextView  
    ...  
    android:id="@+id/tvPostsCount"  
    android:text  
    ="@{String.valueOf(viewModel.userLiveData.postsCount)}" />  
  
...  
</layout>
```

3. Oferirea variabilei *viewModel* fișierului de layout, atunci când se desenează pentru prima dată interfața fragmentului.

```
//clasa generată în timpul compilării  
private lateinit var binding: FragmentProfileBinding  
//ViewModel-ul corespunzător fragmentului  
private val viewModel: ProfileViewModel by viewModels()  
  
override fun onCreateView(...): View {  
    binding = FragmentProfileBinding.inflate(inflater)  
    binding.viewModel = viewModel  
    ...  
    return binding.root  
}
```

4. Actualizarea informației din LiveData atunci când apelul la baza de date s-a terminat.

```
fun getUserByUid(uid: String) {  
    viewModelScope.launch {  
        val user = usersRepository.getUserByUid(uid)  
        if(user != null)  
            userLiveData.postValue(user)  
    }  
}
```

După acești pași, orice actualizare a informației din LiveData va declanșa împrospătarea informației afișată pe ecranul dispozitivului. Această metodă a fost mai apoi folosită în majoritatea ecranelor aplicației, deoarece s-a dovedit a fi foarte practică și se integrează mult mai bine cu arhitectura MVVM față de varianta clasică.

A doua problemă ține de capitolul arhitecturii aplicației (secțiunea 3.3.3), și anume partajarea informațiilor între mai multe *View*-uri. Motivația acestei probleme a fost nevoia de a reține informațiile introduse în mai multe ferestre succesive pentru scenarii precum funcționalitățile de adăugare a evenimentelor și a autovehiculelor (secțiunile 2.3.5 și 2.3.7), unde utilizatorul este rugat să introducă anumite informații de-a lungul unui proces, pentru nu există o singură fereastră plină de câmpuri de completare.

Prima soluție posibilă este transferarea informațiilor între ferestre prin transmiterea acestora ca și argument. Pentru ușurința explicațiilor, vom considera că există doar două ferestre, prima fereastră din care se dorește avansarea (notată cu “A”), iar a doua fereastră în care se dorește să se ajungă (notată cu “B”). Transmiterea informației de la A la B se poate realiza prin folosirea funcționalității de pasare a argumentelor din “Navigation Component” (din secțiunea 3.3.5). În scop didactic, o implementare posibilă este următoarea:

1. Se crează un graf de navigare între cele două ferestre.

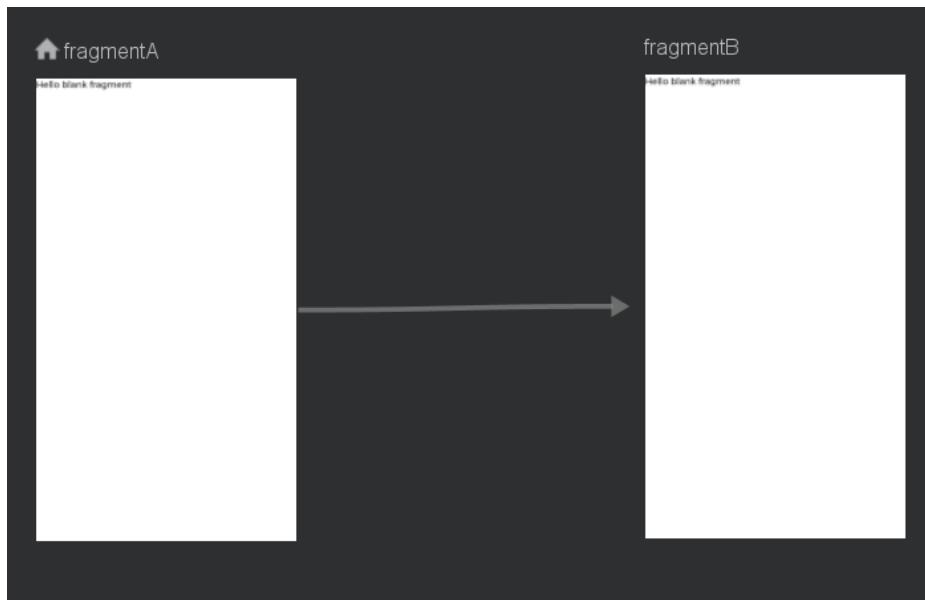


Figura 3.12. Graful de navigare între destinații

Se adaugă fragmentului A acțiunea de avansare către fragmentul B, iar fragmentului B îi este asociat un argument de tip String denumit “*myArgument*”. După procesul de compilare vor fi generate clasele specializate “*FragmentBArgs*” și “*FragmentADirections*” de care ne vom folosi la pașii următori pentru accesarea acțiunii și a argumentului.

```
<navigation ...>

<fragment
    android:id="@+id/fragmentA"
    android:name="com.catasoft.autoclub.FragmentA"
    android:label="fragment_a"
    tools:layout="@layout/fragment_a" >

    <action
        android:id="@+id/action_fragmentA_to_fragmentB"
        app:destination="@+id/fragmentB" />

</fragment>

<fragment
    android:id="@+id/fragmentB"
    android:name="com.catasoft.autoclub.FragmentB"
    android:label="fragment_b"
    tools:layout="@layout/fragment_b" >

    <argument
        android:name="myArgument"
        app:argType="string"
        app:nullable="true" />

</fragment>

</navigation>
```

2. Se apeleză din fragmentul A acțiunea de avansare către fragmentul B, declarată la punctul anterior, căreia îi este atașat și argumentul care se dorește a fi trimis.

```
class FragmentA : Fragment() {

    override fun onCreate(savedInstanceState: Bundle?) {
```

```

        super.onCreate(savedInstanceState)

        val argumentToPass = "Hello! I am the argument!"
        val action =
FragmentADirections.actionFragmentAToFragmentB(argumentToPass)
        findNavController().navigate(action)
    }
}

```

3. Se primește argumentul în fragmentul B printr-o proprietate delegată [25].

```

class FragmentB : Fragment() {

    private val args: FragmentBArgs by navArgs()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d("Args test", "Am primit argumentul: ${args.myArgument}")
    }
}

```

Avantajele acestei soluții sunt: Accesibilitatea independentă de arhitectura proiectului; Folosirea unor librării moderne și recomandate de către documentația Android, care asigură generarea automată a tuturor claselor și a câmpurilor necesare; Ușor de urmărit și de înțeles procesul de transmitere doar prin vizualizarea grafică a grafului de navigație.

Dezavantajele sunt însă cele care cântăresc destul de greu din perspectiva unui implementări ușor de întreținut, și anume: În cazul mai multor fragmente procesul de mai sus trebuie repetat pentru fiecare pasă de argument în parte (i.e. pentru N fragmente sunt $N-1$ pase); Nu se pot transmite informații de dimensiuni mari, cum ar fi imaginile, deoarece există o limită maximă de 1 MB de date [26]; Dacă se dorește revenirea la o fereastră anterioară, informațiile trebuie reținute și pasate în sens invers față de transmiterea inițială; Graful de navigație poate deveni destul de încărcat și greu de urmărit.

A doua soluție posibilă și aleasă în implementarea finală este folosirea unui ViewModel partajat pentru toate fragmentele din proces, prin intermediul căreia să se ofere un loc pentru

reținerea și accesarea tuturor informațiilor. Implementarea acestei soluții este facilă și intuitivă dacă toate fragmentele aparțin de aceeași activitate, lucru care se respectă în cazul problemei de la care s-a plecat. Ideea de rezolvare este folosirea unui ViewModel asociat activității de bază care va fi primit de fiecare fragment în parte. Un exemplu concret este implementarea funcționalității de adăugare a unui autovehicul (secțiunea 2.3.7) care este formată din 4 fragmente: introducerea mărcii și a modelului, introducerea fotografiei, introducerea numărului de înmatriculare, confirmarea autovehiculului creat. Pașii implementării sunt următorii (liniile de cod au fost modificate pentru a se rezuma doar la transmiterea de informații, nu și la întreaga funcționalitate aşa cum se regăsește în sursa originală):

1. Se crează *ViewModel*-ul care va fi partajat. Va conține câte o variabilă de tipul LiveData [27] pentru fiecare informație în parte.

```
class AddCarViewModel: ViewModel() {  
    val carModelLiveData: LiveData<String> = MutableLiveData()  
    val carMakeLiveData: LiveData<String> = MutableLiveData()  
    val carYearLiveData: LiveData<Int> = MutableLiveData()  
    val avatarLiveData: LiveData<Bitmap> = MutableLiveData()  
    val numberPlateLiveData: LiveData<String> = MutableLiveData()  
}
```

Se folosește LiveData pentru a comunica schimbarea acestor valori în timp “real”, fără a fi nevoie de rescrierea codului de actualizare a datelor la fiecare schimbare a valorilor.

2. Se obține *ViewModel*-ul din pasul anterior în fiecare fragment.

```
class AddCarMakeAndModelFragment : Fragment() {  
    private val viewModel: AddCarViewModel by activityViewModels()  
}  
  
class AddCarNumberPlateFragment : Fragment() {  
    private val viewModel: AddCarViewModel by activityViewModels()  
}
```

```

class AddCarAvatarFragment : Fragment() {
    private val viewModel: AddCarViewModel by activityViewModels()
}

class AddCarSummaryFragment : Fragment() {
    private val viewModel: AddCarViewModel by activityViewModels()
}

```

3. Se adaugă în ViewModel metodele necesare manipulării variabilelor.

O funcție de acest tip poate fi metoda folosită pentru memorat marca și modelul mașinii introduse de către utilizator:

```

fun setMakeAndModel(make: String, model: String) {
    carMakeLiveData.value = make
    carModelLiveData.value = model
}

```

Notă: Nu este nevoie de implementarea unor funcții de acces a acestor valori atunci când se folosește limbajul de programare Kotlin.

Avantajele acestei soluții sunt: Simplitatea implementării prin scrierea unui singur ViewModel pentru toate fragmentele; Stocarea și accesul informațiilor sunt ușor de folosit și de implementat; Posibilitatea întoarcerii la un ecran anterior este implicit funcțională deoarece informația nu aparține de fragmente; Posibilitatea folosirii tipului de date LiveData pentru actualizarea în timp real a interfeței; Metodă recomandată și documentată de către Google [28]. Dezavantajele notabile ale acestei soluții sunt obligativitatea folosirii arhitecturii MVVM și amestecarea în ViewModel a metodelor specifice fiecărui fragment, rezultând într-un fișier de dimensiuni mari și greu de urmărit.

În urma acestei comparații de soluții, alegerea devine destul de evidentă din mai multe motive. Deoarece se dorește transmiterea unei fotografii, atât pentru crearea unui autovehicul, cât și pentru crearea unui eveniment, limita de 1MB pentru datele transmise prin prima metodă reprezintă o problemă majoră. Se adaugă de asemenea și faptul că revenirea la un ecran anterior este greu de obținut datorită numărului de pași suplimentari pentru actualizarea *View*-ului și pentru retransmiterea datelor. Astfel, cea mai bună soluție este cea de-a doua deoarece servește mult mai bine nevoile funcționalității dorite, în contextul în care aplicația este deja bazată pe arhitectura MVVM, dezavantajele fiind mai ușor de suportat.

4. Concluzii

Această lucrare și-a propus rezolvarea unei probleme semnalată în cadrul unui public restrâns de utilizatori ai dispozitivelor mobile, respectiv persoanelor pasionate de automobile. Deși rezultatul este cu scop demonstrativ, aplicația se află într-un stadiu funcțional și destul de avansat, oferind suficientă utilitate pentru a putea fi lansată și oferită utilizatorilor sistemului de operare Android.

Abordarea lucrării a fost orientată pe partea de înțelegere și implementare a conceptelor și a practicilor moderne de dezvoltare a aplicațiilor mobile, fiind tratate superficial părțile de stilizare a componentelor vizuale sau părțile de utilizare a librăriilor. Alegerea este motivată de dorința explorării modalităților prin care componente tehnice mai vechi, care deși și-au dovedit utilitatea și fiabilitatea, au fost regândite și transformate în tehnologii moderne, cu scopul de a oferi o experiență cât mai plăcută atât dezvoltatorului cât și utilizatorului final.

Deși aplicația abordată este stabilă și utilizabilă, există unele funcționalități care nu au făcut subiectul acestei lucrări și care sunt în curs sau urmează a fi dezvoltate:

- Dezvoltarea unui sistem de prietenie între membrii aplicației, bazat pe oferirea și acceptarea unor cereri de prietenie
- Oferirea unei modalități de conversare pentru utilizatorii aplicației
- Posibilitatea creării unor grupuri de interese comune, unde utilizatorii se pot alătura și discuta despre subiecte specifice acestor grupuri. Această funcționalitate ar încuraja membrii aplicației să ceară și să primească sfaturi în materie de întreținere și modificare a autovehiculelor, acest lucru fiind posibil momentan doar prin participarea la evenimentele organizate în cadrul aplicației.
- Adăugarea unui ecran de noutăți pentru postarea de fotografii sau de știri importante din lumea automobilistică. Această funcționalitate presupune și implementarea unor moduri de vizualizare a acestor postări de către ceilalți utilizatori, cum ar fi: toate postările, cele mai recente, cele mai apreciate, postările prietenilor.

Pe lângă adăugarea de noi funcționalități, s-au luat în calcul și niște îmbunătățiri substanțiale care necesită a fi implementate:

- Proiectarea unui sistem de caching performat pentru oferirea unei experiențe de utilizare mai plăcută în timpul încărcării datelor.
- Desenarea unei interfețe care să se muleze corect pentru modul de utilizare landscape al dispozitivului.
- Reținerea autentificării utilizatorului după terminarea folosirii aplicației.
- Realizarea unui studiu privind fezabilitatea utilizării unei baze de date relațională.

Consider că prezenta lucrare este un punct de plecare solid pentru continuarea dezvoltării conceptului abordat, având o arhitectură și o bază solidă care permit realizarea funcționalităților și a îmbunătățirilor menționate mai sus, cu scopul atingerii unui stadiu de produs final, complet utilizabil și comercializabil.

Bibliografie

- [1] DataReportal (2021). DIGITAL 2021: GLOBAL OVERVIEW REPORT. Disponibil la: <https://datareportal.com/reports/digital-2021-global-overview-report>. Accesat: 26 Mai 2021.
- [2] Google. Navigation. Disponibil la: <https://developer.android.com/guide/navigation>. Accesat: 6 Iunie 2021.
- [3] Statcounter (2021). Mobile Operating System Market Share Worldwide. Disponibil la: <https://gs.statcounter.com/os-market-share/mobile/worldwide>. Accesat: 26 Mai 2021.
- [4] Google (2021). Platform Architecture. Disponibil la: <https://developer.android.com/guide/platform>. Accesat: 26 Mai 2021.
- [5] Google (2021). Android for developers. Disponibil la: <https://developer.android.com/>. Accesat: 26 Mai 2021.
- [6] Google (2021). Android Runtime (ART) and Dalvik. Disponibil la: <https://source.android.com/devices/tech/dalvik>. Accesat: 27 Mai 2021.
- [7] HubSpire (2021). Application Programming Interface. Disponibil la: <https://www.hubspire.com/resources/general/application-programming-interface/>. Accesat: 27 Mai 2021.
- [8] Oracle (2021). What is Java technology and why do I need it?. Disponibil la: https://java.com/en/download/help/whatis_java.html. Accesat: 27 Mai 2021.
- [9] Google (2021). Take photos. Disponibil la: <https://developer.android.com/training/camera/photobasics>. Accesat: 27 Mai 2021.
- [10] Google (2021). SDK Platform release notes. Disponibil la: <https://developer.android.com/studio/releases/platforms>. Accesat: 27 Mai 2021.

[11] Statista (2017). Smartphone Life Cycles Are Changing. Disponibil la: <https://www.statista.com/chart/8348/smartphone-life-cycles-are-changing/>. Accesat: 27 Mai 2021.

[12] Google (2021). Meet Android Studio. Disponibil la: <https://developer.android.com/studio/intro>. Accesat: 27 Mai 2021.

[13] Steven J Zeil (2017). Integrated Development Environments . Disponibil la: <https://www.cs.odu.edu/~zeil/cs350/f17/Public/IDEs/index.html>. Accesat: 27 Mai 2021.

[14] Google (2021). Android Studio. Disponibil la: <https://developer.android.com/studio>. Accesat: 27 Mai 2021.

[15] Trygve Reenskaug (1979). MODELS - VIEWS - CONTROLLER. Disponibil la: <https://folk.universitetetioslo.no/trygver/1979/mvc-2/1979-12-MVC.pdf>. Accesat: 30 Mai 2021.

[16] G. E. Krasner, S. T. Pope (1982), “A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System”, Journal of object oriented programming, Vol. 1, nr. 3, p. 26 - 49.

[17] Lou T (2018), “A comparison of Android Native App Architecture, MVC, MVP and MVVM”, Eindhoven University of Technology.

[18] Tin Megali. An Introduction to Model View Presenter on Android. Disponibil la: <https://code.tutsplus.com/tutorials/an-introduction-to-model-view-presenter-on-android--cms-26162>. Accesat: 30 Mai 2021.

[19] Eric Maxwell. MVC vs. MVP vs. MVVM on Android. Disponibil la: <https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>. Accesat: 30 Mai 2021.

[20] Google (2021). Guide to app architecture. Disponibil la: <https://developer.android.com/jetpack/guide>. Accesat: 27 Mai 2021.

[21] Google (2021). Data Binding Library. Disponibil la: <https://developer.android.com/topic/libraries/data-binding>. Accesat: 27 Mai 2021.

[22] Google (2021). ViewModel Overview . Disponibil la:
<https://developer.android.com/topic/libraries/architecture/viewmodel>. Accesat: 27 Mai 2021.

[23] DevIQ. Repository Pattern. Disponibil la:
<https://deviq.com/design-patterns/repository-pattern>. Accesat: 30 Mai 2021.

[24] Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95. ISBN 978-0135974445.

[25] Kotlin Foundation. Delegated properties. Disponibil la:
<https://kotlinlang.org/docs/delegated-properties.html>. Accesat: 3 Iunie 2021.

[26] Google (2021). TransactionTooLargeException. Disponibil la:
<https://developer.android.com/reference/android/os/TransactionTooLargeException>. Accesat: 3 Iunie 2021.

[27] Google (2021). LiveData Overview. Disponibil la:
<https://developer.android.com/topic/libraries/architecture/livedata>. Accesat: 3 Iunie 2021.

[28] Google (2021). Share data between fragments. Disponibil la:
<https://developer.android.com/topic/libraries/architecture/viewmodel#sharing>. Accesat: 3 Iunie 2021.

[29] Shibaji Debnath. Why java for android development?. Disponibil la:
<https://www.shibajidebnath.com/java-android-development/>. Accesat: 4 Iunie 2021.

[30] Stack Overflow. 2020 Developer Survey. Disponibil la:
<https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents>. Accesat: 4 Iunie 2021.

[31] Kotlin Foundation. FAQ. Disponibil la: <https://kotlinlang.org/docs/faq.html>. Accesat: 4 Iunie 2021.

[32] Google (2021). Android's Kotlin-first approach. Disponibil la:
<https://developer.android.com/kotlin/first>. Accesat: 4 Iunie 2021.

[33] Kotlin Foundation. FAQ. Disponibil la: <https://kotlinlang.org/docs/coroutines-basics.html>. Accesat: 4 Iunie 2021.

[34] Kotlin Foundation. CoroutineScope. Disponibil la:
<https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html>. Accesat: 5 Iunie 2021.

[35] Google. Android Jetpack. Disponibil la: <https://developer.android.com/jetpack>. Accesat: 5 Iunie 2021.

[36] Richard Carr. Observer Design Pattern. Disponibil la:
<http://www.blackwasp.co.uk/Observer.aspx>. Accesat: 6 Iunie 2021

[37] Google. Data Binding Library. Disponibil la:
<https://developer.android.com/topic/libraries/data-binding>. Accesat: 6 Iunie 2021.