Tehnologije i arhitektura razvoja mobilne aplikacije za android operativni sustav na primjeru aplikacije za pružanje usluga selidbe i transporta

Kovačević, Karlo

Sveučilište u Zadru

Odjel za informacijske znanosti

Sveučilišni prijediplomski studij

Informacijske tehnologije

Karlo Kovačević

TEHNOLOGIJE I ARHITEKTURA RAZVOJA MOBILNE APLIKACIJE ZA ANDROID OPERATIVNI SUSTAV NA PRIMJERU APLIKACIJE ZA PRUŽANJE USLUGA SELIDBE I TRANSPORTA

Završni rad

Zadar, 2024.

Sveučilište u Zadru
Odjel za informacijske znanosti
Stručni prijediplomski studij
Informacijske tehnologije

TEHNOLOGIJE I ARHITEKTURA RAZVOJA MOBILNE APLIKACIJE ZA ANDROID OPERATIVNI SUSTAV NA PRIMJERU APLIKACIJE ZA PRUŽANJE USLUGA SELIDBE I TRANSPORTA

Završni rad

Student/ica: Karlo Kovačević

Mentor/ica: Doc. dr sc. Tomislav Jakopec

Zadar, 2024.

Izjava o akademskoj čestitosti

Ja, **Karlo Kovačević,** ovime izjavljujem da je moj **završni** rad pod naslovom **Tehnologije i arhitektura razvoja mobilne aplikacije za Android operativni sustav na primjeru aplikacije za pružanje usluga selidbe i transporta** rezultat mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na izvore i radove navedene u bilješkama i popisu literature. Ni jedan dio mojega rada nije napisan na nedopušten način, odnosno nije prepisan iz necitiranih radova i ne krši bilo čija autorska prava.

Izjavljujem da ni jedan dio ovoga rada nije iskorišten u kojem drugom radu pri bilo kojoj drugoj visokoškolskoj, znanstvenoj, obrazovnoj ili inoj ustanovi.

Sadržaj mojega rada u potpunosti odgovara sadržaju obranjenoga i nakon obrane uređenoga rada.

Zadar, 15. rujna 2024.

SAŽETAK

Ovaj rad ima za zadatak istražiti i opisati proces razvoja, prvo kroz teorijski dio, a onda i kroz samu izradu Android aplikacije koja će omogućiti korisnicima povezivanje i jednostavnu komunikaciju sa pružateljima usluga selidbi i prijevoza namještaja. Tehnologije koje će biti obrađene i korištene u praksi su: programski jezik Kotlin, razvojno okruženje Android Studio, alat za dizajn sučelja Figma te sveobuhvatna platforma za razvoj aplikacija Firebase. Korištene vanjske biblioteke koje čine dio praktičnog rada su Jetpack Compose, Dagger Hilt, Glide, Gson i Retrofit. Navedene biblioteke su kao prvo omogućile drugačiji pristup izrade sučelja bez XML datoteka do jednostavne primjene *singletona* za sve komponente te jednostavnu manipulaciju podataka i slika u komunikaciji s bazom podataka. U prvom dijelu opisuje se sama svrha i cilj rada, nakon toga ide teorijski dio s pregledom pristupa razvoju android aplikacija kroz povijest te na taj način opisati teorijski dio pojedinih tehnologija i pristupa poput SOLID i *clean* pristupa. Drugi dio diskutira korištene tehnologije, njihove prednosti i nedostatke, ali i razlog njihovog odabira u okviru ovoga rada. Zadnji dio se tiče konkretne implementacije kroz android aplikaciju gdje opisujemo razvojni proces od samog dizajna do funkcionalne aplikacije. Kompletan kod aplikacije dostupan je s javnog repozitorija na GitHubu i nalazi se na poveznici: https://github.com/XSSmachine/CityGo/tree/master

Ključne riječi: Android aplikacija, Kotlin, Android Studio, Figma, Firebase, Clean arhitektura

POPIS KORIŠTENIH KRATICA

HTTP Hypertext Transfer Protocol = protokol za prijenos hiperteksta

JSON JavaScript Object Notation = format za razmjenu podataka baziran na JavaScript sintaksi

XML Extensible Markup Language = proširivi označni jezik za strukturiranje podataka
POJO Plain Old Java Object = jednostavni Java objekt bez posebnih ograničenja

UI User Interface = korisničko sučelje

UX User Experience = korisničko iskustvo

CRUD Create, Read, Update, Delete = osnovne operacije za rad s podacima: stvaranje, čitanje, ažuriranje, brisanje

MVVM Model-View-ViewModel = arhitekturni obrazac za razvoj korisničkih sučelja

MVP Model-View-Presenter = arhitekturni obrazac za razvoj korisničkih sučelja

API Application Programming Interface = sučelje za programiranje aplikacija

SQL Structured Query Language = strukturirani upitni jezik za rad s relacijskim bazama podataka

SADRŽAJ

1. UVOD ... 1

1.1. Ideja, namjena, tehnologije i alati ... 1

1.2. Izvori podataka i metode prikupljanja ... 1

1.3. Sadržaj i struktura rada ... 2

2. TEHNOLOGIJE I ARHITEKTURA RAZVOJA MOBILNE APLIKACIJE ZA ANDROID OPERATIVNI SUSTAV ... 3

2.1. Povijesni pregled razvoja aplikacija za Android operativni sustav ... 3

2.2. Primjer legacy Android aplikacije ... 3

2.3. Definiranje granica između ovisnosti klasa ... 5

2.3.1. SOLID principi ... 6

2.3.2. Načela kohezije komponenti ... 7

2.3.3. Principi spajanja komponenti ... 8

2.4. Evolucija Android sustava ... 9

2.4.1. Fragmenti ... 9

2.4.2. Gradle sustav za upravljanje ovisnostima ... 10

2.4.3. Umrežavanje ... 11

2.4.4. *Humble objects* ... 11

2.4.5. *Functional paradigms* ... 11

2.4.6. Kotlin ... 12

2.4.7. *Dependency injection* ... 12

2.4.8. Android arhitektonske komponente ... 12

2.4.9. Korutine ... 13

2.4.10. Jetpack Compose ... 13

4. Korištene Tehnologije ... 18

4.1. Figma ... 18

4.2. Android Studio ... 19

4.3. Kotlin – programski jezik ... 20

4.4. JetPack Biblioteka ... 20

4.5. Firebase Servisi u Oblaku ... 20

5. PRIMJER APLIKACIJE ZA PRUŽANJE USLUGA SELIDBE I TRANSPORTA ... 22

5.1. Minimalno vitalni proizvod ... 22

5.2. Dizajn korisničkog sučelja ... 23

5.2.1. Sign up/Sign in screen ... 23

5.2.2. Dashboard ... 25

5.2.3. Zasloni za kreiranje oglasa 26
5.2.4. Lista vlastitih oglasa 29
5.2.5. Detalji oglasa 30
5.2.6. Ponude prijevoznika 31
5.2.7. Dogovor za posao 32
5.3. Baza podataka 33
5.4. Arhitektura Android aplikacije 36
5.5. Implementacija ključnih funkcionalnosti 41
6. Rasprava 42
7. Zaključak 44
Summary 46
Popis literature 47

1. UVOD

1.1. Ideja, namjena, tehnologije i alati

U ovom radu biti će fokus na razvoju programskog rješenja u Android operativnom sustavu kroz realan primjer aplikacije koja bi olakšala organizaciju prijevoza s obje strane tržišta. Za prikaz skupa tehnologija kroz izradu mobilne aplikacije s pohranom podataka u oblaku izrađena je mobilna aplikacija City Go. Pepoznata je potreba za digitalnom platformom koja bi olakšala ovaj proces te pomogla u rješavanju problema s kojima se korisnici svakodnevno suočavaju. No pored osnovne ideje, cilj implementacije ovoga rada je da se samo programsko rješenje može proširiti u ostale branše koje također zahtijevaju bolje organizacijsko rješenje kada je u pitanju prijevoz. Aplikacija se povezuje na Firebase servis i koristi *Realtime Database* za pohranu podataka u oblaku, dok se na lokalnoj razini koristi *Room Database* kako bi se omogućila izvanmrežnu funkcionalnost. Aplikacija je namijenjena svima koji su u potrazi za uslugama selidbe i transporta namještaja, kao i pružateljima istih usluga koji se žele lakše i efikasnije povezati s potražnjom. Za razvoj odabran je Android operacijski sustav zbog široke rasprostranjenosti te praktičnosti prijelaza u *multiplatform* razvoj. Aplikacija je izrađena u programskom jeziku Kotlin i bazirana je na MVVM arhitekturi i *clean* pristupu. Za razvoj prototipa i dizajna mobilne aplikacije korišten je alat Figma, a za razvoj programskog dijela Android Studio.

1.2. Izvori podataka i metode prikupljanja

Za pisanje teorijskog dijela rada korištena je službena dokumentacija navedenih tehnologija i alata kao i trenutno aktualnih knjiga koje opisuju i ulaze u zadanu problematiku. Dio materijala je preuzet od stručnih i znanstvenih publikacija [1]od provjerenih stručnjaka kao i informacije stečene razgovorom i komunikacijom s mentorom i profesorom koji ima preko 15 godina iskustva rada s Android tehnologijama.[2]

[1] Nayab Akhtar i Sana Ghafoor, Analysis of Architectural Patterns for Android Development, 2021.
[2] Google Android developer mrežna stranica, https://developer.android.com/guide [pristupljeno 26. lipnja 2024.]

1.3. Sadržaj i struktura rada

Rad je strukturiran u pet glavnih cjelina. U prvoj, uvodnoj cjelini su navedeni glavni ciljevi i svrha ovog rada kao i opis odabrane problematike.

U drugoj cjelini proći će se kroz povijesni razvoj Androida iz perspektive razvojnog programera. U tom poglavlju se utvrđuju pojedini problemi i kako se vremenom s njima suočava i napreduje sustav ali uz to obrađujemo promjene u načinu razvijanja programskih rješenja. Na kraju poglavlja predstavit će se pristup razvoju koji ide korak dalje nakon svih navedenih iskoraka i koji je korišten u sklopu izrade praktične aplikacije za ovaj rad.

U trećem poglavlju fokus prelazi na razvojni pristup *Clean* arhitekture koji predstavlja trenutni industrijski standard. No također biti će riječ i o ostalim dobrim praksama kod pisanja koda, kao što su SOLID, različiti strukturalni pristupi i putokazi koji predstavljaju širu sliku kvalitetnog razvoja aplikacijskih rješenja.

U četvrtom poglavlju opisuju se glavne tehnologije korištene u razvoju Android aplikacije. Opisom pojedinih tehnologija omogućen je kronološki uvod u praktični dio razvoja koji predstavlja pozitivne i negativne strane svake od njih, ali na kraju stvara veću sliku mentalnog sklopa potrebnog za stvaranje programskog rješenja koje nije samo funkcionalno već i skalabilno.

U petom dijelu opisuje se razvojni proces i implementacija City Go aplikacije s naglaskom na prednosti odabranih tehnologija, arhitekture i pristupa. Prvo se demonstrira početna faza koja sadrži dizajn sučelja i konkretiziranje same ideje kroz različite perspektive koje nisu samo ograničene na programiranje. Nadalje opisuju se modeli za spremanje u baze podataka te važnost Firebase servisa za samu implementaciju. Postavljanje projekta kroz odabrano razvojno okruženje predstavlja kompletan proces implementacije korištenjem programskog jezika Kotlin. Cilj rada je ponuditi kvalitetan uvod i putokaz kroz pregled svih tehnologija i metodologije potrebne za razvoj robusne i skalabilne Android aplikacije, potkrijepljen konkretnim primjerom implementacije kroz Android operativni sustav.

2. TEHNOLOGIJE I ARHITEKTURA RAZVOJA MOBILNE APLIKACIJE ZA ANDROID OPERATIVNI SUSTAV

2.1. Povijesni pregled razvoja aplikacija za Android operativni sustav

Povijesno gledano, zastarjeli razvoj Android aplikacija nosi sa sobom cijeli niz problema s kojima su se programeri susretali.

Prva stvar koju treba raščistiti je razlika između arhitekture i dizajna aplikacije. To se može jednostavnim riječima predstaviti kroz primjer iz građevinske industrije - arhitektura je plan strukture same građevine, dok dizajn podrazumijeva plan za izradu svakog dijela tog objekta. Prijevod te metodologije u našem slučaju je da arhitektura aplikacije predstavlja plan za provedbu poslovnih i tehničkih zahtjeva, dok se dizajn brine za integraciju svih dijelova, modula i sustava u samu arhitekturu. Idealno bi bilo da se za svaku aplikaciju prepozna koju arhitekturu zahtijeva te kako ukomponirati sve komponente kako bi se pojedini pristupi maksimalno isplatili.[3]

Baza svake Android aplikacije počiva na 4 glavne komponente: [4]

- Aktivnosti – predstavljaju ulaznu točku interakcije s korisnikom
- Servis – predstavlja ulaznu točku rada u pozadini
- *Broadcast receiver* – dopušta sistemsku interakciju s aplikacijom
- *Content provider* – omogućava upravljanje podacima

2.2. Primjer legacy Android aplikacije

Opisan je legacy Android aplikacije kako bismo dobili bolju sliku o daljnjem razvoju i napretku samog razvoja.

Zadatak aplikacije je dohvatiti podatke s *backend* servisa u JSON obliku kroz klasičnu HTTPS vezu. Prvi izazov na ovom zadatku postoji kod dohvaćanja podataka u JSON formatu, što zahtijeva neki mapper kako bi se HTTPS response pretvorio u POJO. Taj mapper bi

[3] Lucidchart mrežna stranica, https://www.lucidchart.com/blog/software-architecture-vs-design, [pristupljeno 24.travnja 2024.]
[4] Android developers mrežna stranica, https://developer.android.com/guide/components/fundamentals, [pristupljeno 24.travnja 2024.]

implementirao sučelje koje predstavlja apstrakciju konverzije JSON objekta. To omogućava konkretiziranje konverzije na više načina.[5]

S druge strane imamo Aktivnost koja pokreće zadani request na server i ažurira UI s rezultatom. No imamo problem - ne možemo izvršavati dugotrajne zadatke na glavnoj niti, te uz to ne možemo osvježiti UI s niti jedne druge niti. U ovoj situaciji trebamo iskoristiti 'AsyncTask' klasu koja nudi metode za odrađivanje posla na posebnoj niti, dok na glavnoj prikazujemo rezultate. 'AsyncTask' klasa nam nudi niz metoda za nesmetano odrađivanje rada na odvojenim nitima kako bi naša aplikacija radila bez zastoja na glavnoj niti. Sada aplikacija funkcionira prema zadanim specifikacijama i odrađuje ono što se traži.

No pogledajmo malo dublje - sama struktura i međusobna ovisnost između klasa nam otkriva moguće probleme u budućnosti ako se budu trebale uvoditi nove promjene, što je vrlo realna situacija za koju se treba pripremiti.

Spomenuti su i unit testovi jer oni predstavljaju smjernice za dobru praksu kod planiranja vlastite arhitekture, ali dalje od toga se neće ići, jer je fokus ovoga rada stvaranje nekog početnog proizvoda u obliku aplikacije koju tek čeka daljnji razvoj i testiranja kako bi išla u produkciju. Ovaj rad ima za cilj omogućiti paletu informacija i uputa kako krenuti u konkretnu izgradnju proizvoda na Android platformi.

Sada treba postaviti nekoliko pitanja. Prvo - „Što od ovoga možemo testirati?“. Odgovor na ovo nije jednostavan jer postoje ograničenja na tehničkoj razini, budući da se kod izvršava na mobilnom uređaju ili emulatoru, a nama treba da izvršimo testove na uređaju koji se koristi za sam razvoj aplikacije. No, sada je tu 'Firebase Test Lab' koji omogućuje testiranje kroz oblak, ali ni to nije idealno jer zahtijeva ulaganje novca. S time ostaje samo jedna opcija, a to je napisati lokalne unit testove umjesto instrumentalnih.

Drugo pitanje - „Kako izvesti dohvat podataka s drugog HTTP endpointa pored već postojećeg?“. Odgovor je jasan - treba kreirati novu implementaciju za nove podatke koji će biti dodani. To se može realizirati tako da se napravi posebna klasu koja nadograđuje 'AsyncTask' ili izvršiti oba requesta u postojećoj implementaciji i iz nje mijenjati UI. U prvom slučaju situacija je nezgodna jer sada postoje dva 'AsyncTaska' koje treba održavati ud dodatni

[5] Alexandru Dumbravan, Clean Android Architecture, 2022., str. 5-11.

problem kod testiranja. U slučaju dohvata iz iste klase, tada se odgovornost same klase povećava, što zapravo treba izbjeći.

Treće pitanje – „Kako riješiti pred memoriranje podataka kroz 'SQLite'?“. Treba dodati novu database klasu koja sadrži CRUD metode i odlučiti koja od klasa će imati ovisnost s njom. Ili povećati odgovornost klase za dohvat podataka ili 'AsyncTask' klase, što opet nosi svoju cijenu.

Finalno pitanje – „Koliko posla je potrebno za izvoz trenutne biznis logike u drugi projekt?“. Ovo pitanje je bitno jer pokazuje kako treba razmišljati i planirati strukturu koda, jer inače, kao i u ovoj situaciji, dolazi do problema zbog bliske ovisnosti na 'MainActivity'.

Kod definiranja arhitekture, važno je podijeliti komponente u slučaju višekomponentnog projekta i važno je razmisliti da svaki modul predstavlja najmanju cjelinu koda koji djeluje kao nezavisna cjelina. U ovom slučaju jedan modul ima par postojećih ovisnosti koje su zavisne o JSON formatu i pitanje je hoće li ta struktura biti u sklopu drugog projekta. Tu treba povući granice između ovisnosti klasa tako da se jednostavno mogu prilagoditi bilo kojoj strukturi podataka.6

Važno je postavljati ovakva pitanja zato što su to sve situacije koje su se već dogodile programerima u praksi, odnosno svi ti izazovi su dio životnog ciklusa aplikacije. Sada se ima bolji uvid kako je to izgledalo prije, problemi koji vode između dvije vatre, pristupi koji kompliciraju situaciju i još mnoge druge stvari, većinom prouzrokovane međuovisnostima u Android sustavu.

2.3. Definiranje granica između ovisnosti klasa

U ovoj sekciji se prolazi kroz neke principe dizajna koji se smatraju važni, kako bolje složiti samu arhitekturu aplikacije. Dosad je jasno da kod mora biti održiv, jasan i fleksibilan. To nije lak zadatak, ali tu u pomoć dolaze ovi principi kod izrade klasa ili komponenti koje predstavljaju minimalnu količinu koda koja predstavlja zaseban dio sustava.

[6] Alexandru Dumbravan, Clean Android Architecture, 2022., str. 11-13.

2.3.1. SOLID principi

Ovo su jedni od poznatijih principa dizajna koje je prvi opisao Robert C. Martin, a sam ovaj naziv predstavlja akronim za 5 principa koje ćemo ukratko objasniti:

Single Responsibility Principle (SRP) - Znači da klasa, odnosno modul treba imati samo jednu odgovornost odnosno zadatak, tako da se bilo kakve promjene koje se tiču izvršavanja te zadaće odnose samo na ovu klasu.

Open-closed Principle (OCP) - Ovo dosta ljudi smatra najbitnijim pravilom u ovom akronimu, a tiče se ideje da kod treba dizajnirati tako da se nove funkcionalnosti mogu dodavati bez potrebe za mijenjanjem postojećeg koda. Neki će odmah pomisliti na nasljeđivanje, no to je zapravo velika greška jer dovodi do uskog povezivanja kroz ovisnost podklase o implementacijskim detaljima nadklase. Bolje rješenje je predložio sam Robert C. Martin - da kroz polimorfistički mehanizam primjenom sučelja, koji su sami po sebi zatvoreni za modifikaciju, omogući jednostavno proširenje implementacijom metoda. Tu još treba napomenuti strategy pattern koji direktno zagovara ovu metodologiju.[7]

Liskov Substitution Principle (LSP) - Ovaj princip je pomalo zbunjujuć i dosta nezgodan za shvatiti te se često preskače u implementacij upravo iz tog razloga. Zapravo, sve što on zagovara je da sve izvedene klase trebaju biti uporabljive kroz osnovni interfejs bez potrebe da klijent zna razliku. Na primjer, ako koristimo klasu 'Ptica' koja ima metode za mogućnost letenja, hranjenja i glasanja, klasa 'Patke' koja nasljeđuje klasu 'Ptice' će se u biti isto ponašati kao obična ptica, tako da nije bitno koristi li se parent ili child klasa jer obje pružaju iste funkcionalnosti.

Interface Segregation Principle (ISP) - Ovaj princip je vrlo jednostavan za razumjeti i koristiti, a govori da klasa ne bi trebala biti prisiljena implementirati sučelje koje sadrži metode irelevantne za tu klasu. Taj problem treba riješiti na način da se razbiju veća sučelja na više manjih primjenom SRP principa.

Dependency Inversion Principle (DIP) - Skreće pozornost na važnost ovisnosti o apstraktnim klasama i sučeljima te da zapravo treba izbjegavati ovisnosti o konkretnim klasama

[7] Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017., str. 57-62.

jer onda postoji problem kod bilo kakvih promjena, zahtijeva promjene od najnižeg do najvišeg sloja.

SOLID principi se koriste u objektno orijentiranom programiranju s ciljem stvaranja fleksibilnih aplikacija koje imaju u vidu proširivanje s novim funkcionalnostima. U nastavku slijedi nadogradnja na opisanu metodologiju.

2.3.2. Načela kohezije komponenti

Referiraju se na ideju da klase odnosno moduli trebaju imati jasno određene odgovornosti i ciljeve te u širem smislu postavljaju pitanje koliko dobro sve klase u određenoj komponenti pristaju zajedno, to jest koje klase bi trebale biti zajedno u toj komponenti. Postoji nekoliko principa koji se obično koriste da se ostvari prava kohezija, tri najvažnija su:

Reuse/Release Equivalence Principle (REP) – Upućuje da svaka grupa klasa treba biti zasebna cjelina odnosno modul koji se može eksportirati i podijeliti s drugim programerima.

Common Closure Principle (CCP) - Predlaže da komponente trebaju imati samo jedan razlog za izmjenu, a to je ništa više nego implementacija pravila o samo jednoj odgovornosti komponente.

Common Reuse Principle (CRP) – Početna ideja je da komponenta treba sadržavati samo klase koje trebaju biti zajedno, jer korisnici komponenti ovise o svim klasama koje se u njoj nalaze, a ne samo o nekima.

Kada se stave svi ovi principi u praksu, dolazi do novih konflikata i problema. Na primjer, CRP i CCP idu prema većim i kompleksnijim komponentama, dok CRP zagovara manje komponente. No zato su to samo principi, a ne pravila. Uvijek treba naći balans između njih jer za svaki projekt koji se radi postoje određeni zahtjevi i prema tome nema jedinstvenog rješenja odnosno recepta za definiranje savršene arhitekture. No sa sigurnošću se može reći kako je primjena ovih principa *'trial and error'* put koji podrazumijeva konstantne promjene i prilagodbe.[8]

[8] Huawei developers članak s Medium mrežne stanice, https://medium.com/huawei-developers/kotlin-solid-principles-tutorial-examples-192bf8c049dd, [pristupljeno 28.travnja 2024.]

Sada kada se prošlo kroz SOLID i osnovne principe izrade pojedinih komponenti, treba dalje vidjeti kako upravljati cijelim setom komponenti.

2.3.3. Principi spajanja komponenti

Ovi principi usmjeravaju kako pravilno upravljati odnosima između komponenti u Android aplikaciji, odnosno kako pravilno manipulirati Gradle ovisnostima među različitim modulima.

Acyclic Dependencies Principle

Ukazuje da treba izbjegavati kružne ovisnosti gdje imamo tri modula koji međusobno zatvaraju krug ovisnosti. Zapravo, u praksi je nemoguće napraviti takvu ovisnost jer Gradle odmah izbacuje grešku i ne da pokrenuti aplikaciju. Jednostavno rješenje je 'Dependency Inversion Principle' koji predlaže da se napravi novi modul koji će ovisiti o apstrakciji i na taj način kreirati implementaciju drugog modula. No ako to nije moguće izvesti, moguće je samo napraviti novi modul koji će imati ovisnost na druga dva modula zajedno.

Stable Dependencies Principle

Upućuje da komponente koje su vjerojatnije sklone promjenama trebaju biti konkretne, dok one stabilne trebaju težiti apstraktnosti. Treba postići da manje stabilne komponente budu zavisne o više stabilnim komponentama. To znači da se stabilnost komponenti mjeri omjerom izlaznih zavisnosti prema ukupnom broju zavisnosti, što je ovaj broj bliže nuli, znači da je komponenta stabilnija. Jedno od rješenja je da se koriste apstraktne komponente i komponente s manje zavisnosti koje imaju jasno određene odgovornosti, što znači manja vjerojatnost promjena.[9]

Stable Abstractions Principle

Princip stabilnih apstrakcija predstavlja važan koncept u softverskoj arhitekturi koji se nadovezuje na princip otvorenosti-zatvorenosti. Ovaj princip nalaže da komponente podložne promjenama trebaju biti konkretnije, dok stabilnije komponente trebaju biti apstraktnije. Cilj je postići fleksibilnost arhitekture koja omogućuje promjene bez potrebe za modificiranjem postojećeg izvornog koda, što se često postiže korištenjem apstraktnih klasa. Apstraktnost

[9] Martin, Robert C., Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017., str. 111-114.

komponente definira se omjerom broja apstraktnih klasa i sučelja prema ukupnom broju klasa u komponenti. Idealno, komponente bi trebale težiti ekstremima - ili visokoj stabilnosti s niskom apstraktnošću, ili niskoj stabilnosti s visokom apstraktnošću, izbjegavajući tzv. "zonu boli" (niska stabilnost i niska apstraktnost) i "zonu beskorisnosti" (visoka stabilnost i visoka apstraktnost). Ovaj pristup omogućuje razvoj robusnih i prilagodljivih softverskih sustava.[10]

S tim zaokružujemo ovaj dio gdje se prošlo kroz neke glavne principe dizajna koje pomažu kod rješavanja problema kada se bude razvijala aplikacija.

2.4. Evolucija Android sustava

U ovom dijelu su opisani svi iskoraci i ključne komponente u Android ekosustavu koje su definirale sam razvoj aplikacija, koji se dogodio rješavanjem pojedinih problema koji su iskakali na tom putu.

Dosad je ustanovljeno kako je to izgledalo prije, no u vrijeme pisanja rada relevantne tehnologije i alati su opisani u nastavku, kako su prethodni problemi svladani uvođenjem novijih tehnologija.

2.4.1. Fragmenti

Prije uvođenja fragmenata, Android aplikacije su se oslanjale isključivo na aktivnosti za upravljanje korisničkim sučeljem, što je često rezultiralo krutim dizajnom, posebno na većim zaslonima. No, fragmenti su riješili ovaj problem omogućujući developerima da kreiraju ponovno iskoristive komponente korisničkog sučelja koje se mogu kombinirati i raspoređivati ovisno o veličini zaslona i orijentaciji uređaja.[11]

Fragmenti su značajno utjecali na dizajn Android aplikacija, potičući razvoj prilagodljivih sučelja i omogućavajući bolje korisničko iskustvo na različitim uređajima. Njihova uloga u podržavanju različitih veličina zaslona bila je ključna za uspjeh Android tableta te većih uređaja. Međutim, fragmenti imaju 'poboljšani' životni ciklus koji je donio novu dimenziju kompleksnosti kroz zaseban životni ciklus te međusobnu komunikaciju između

[10] ProAndroidDev mrežna stranica - https://proandroiddev.com/the-stable-abstractions-principle-in-android-architecture-de2a4c33dddd, [pristupljeno 5.svibnja 2024.]
[11] Yener, Murat i Onur Dundar, Expert Android Studio, 2016., str. 349-352.

pojedinih fragmenata, što je dosta zakompliciralo razvoj, posebno u dijelu upravljanja View-om i prikazom podataka.

2.4.2. Gradle sustav za upravljanje ovisnostima

Razvoj Android aplikacija doživio je značajnu transformaciju s uvođenjem Gradle sustava izgradnje.

U ranim fazama Android razvoja, Eclipse IDE u kombinaciji s Ant sustavom izgradnje bio je primarno razvojno okruženje. Ovaj pristup, iako funkcionalan, imao je određena ograničenja, posebice u pogledu fleksibilnosti konfiguracije i skalabilnosti projekata. Lansiranje Android Studio-a 2013. godine označilo je prekretnicu u Android razvoju, ponajviše zbog integracije Gradle sustava izgradnje.[12]

Gradle je donio niz prednosti u Android razvoj. Prvenstveno, omogućio je znatno veću fleksibilnost konfiguracije kroz sofisticirane skripte pisane u Groovy ili Kotlin DSL. Ova fleksibilnost se očituje u mogućnosti detaljnog prilagođavanja procesa izgradnje specifičnim potrebama projekta. Nadalje, Gradle je značajno olakšao razvoj i održavanje modularnih aplikacija, što je postalo sve važnije s rastućom kompleksnošću Android projekata.

Jedna od ključnih prednosti Gradle-a je pojednostavljeno upravljanje ovisnostima. Ovaj aspekt je od posebne važnosti u modernom razvoju gdje aplikacije često ovise o brojnim vanjskim bibliotekama. Gradle omogućuje developerima jednostavno dodavanje, ažuriranje i upravljanje svim ovisnostima, što značajno ubrzava proces razvoja i smanjuje mogućnost grešaka.

Podrška za različite varijante to jest takozvani 'okusi' aplikacije je još jedna značajka koju je Gradle uveo. Ovo omogućuje kreiranje različitih verzija aplikacije iz istog izvornog koda, što je posebno korisno za razvoj aplikacija s različitim funkcionalnostima za različite korisnike ili tržišta.

Utjecaj Gradle-a na razvoj Android aplikacija je višestruk. Značajno je unaprijedio automatizaciju procesa, pojednostavljujući kompleksne zadatke izgradnje i testiranja. Prilagodljivost koju Gradle pruža omogućila je programerima da prilagode proces izgradnje specifičnim potrebama svojih projekata. Također, olakšao je integraciju raznih alata za praćenje

[12] Annuzzi Jr, Joseph, Lauren Darcey, and Shane Conder, Introduction to Android Application Development: Android Essentials, 2014., str. 55-58.

performansi, analitiku i izvještavanje o greškama, što je postalo ključno u razvoju prvoklasnih aplikacija. Sustav se kontinuirano razvija, s novijim verzijama koje podržavaju Kotlin DSL, korištenjem .kts ekstenzije.[13] Ovo dodatno poboljšava čitljivost i održavanje skripti izgradnje, posebno za timove koji već koriste Kotlin u svojim projektima.

2.4.3. Umrežavanje

Evolucija mrežne komunikacije u Android razvoju donijela je značajne promjene u načinu na koji aplikacije komuniciraju s pozadinskim sustavima. S pojavom popularnih biblioteka otvorenog koda, poput Volley i Retrofit za HTTPS komunikaciju, te Moshi i GSON za serijalizaciju/deserijalizaciju JSON podataka u POJO objekte, proces razmjene podataka između aplikacije i servera značajno je pojednostavljen. Ove biblioteke apstrahiraju kompleksnosti mrežne komunikacije, omogućujući razvojnim timovima da se fokusiraju na definiranje željenih podataka i modela, bez potrebe za detaljnim upravljanjem samim procesom komunikacije. Takav pristup ne samo da ubrzava razvoj, već i povećava pouzdanost i održivost koda, što je posebno važno s obzirom na to da velik broj aplikacija u Google Play trgovini ovisi o efikasnoj HTTPS komunikaciji i obradi JSON podataka. Ova transformacija predstavlja značajan korak naprijed u pojednostavljenju razvoja Android aplikacija, istovremeno podižući standard kvalitete i efikasnosti mrežne komunikacije u mobilnom ekosustavu.[14]

2.4.4. *Humble objects*

Koncept 'humble objects' proizašao je iz potrebe za testiranjem Android aktivnosti i fragmenata, koji su inherentno teški za jedinično testiranje. Ovaj pristup rezultirao je pojavom obrazaca MVP i MVVM. Suština ovih obrazaca leži u transformaciji aktivnosti i fragmenata u jednostavne objekte bez logike, koji zadržavaju reference na elemente korisničkog sučelja, dok se poslovna logika premješta u *presenter* ili *ViewModel*, omogućujući tako lakše jedinično testiranje ključnih komponenti aplikacije.

2.4.5. *Functional paradigms*

Usvajanje funkcionalnih paradigmi u Android razvoju manifestiralo se prvenstveno kroz biblioteku *RxJava*. Ova biblioteka omogućuje implementaciju aplikacija vođenih događajima, nudeći *observables* za emitiranje podataka i *subscribers* za pretplatu na te podatke.

[13] Gradle mrežna stranica - https://docs.gradle.org/current/userguide/kotlin_dsl.html, [pristupljeno 8.svibnja 2024.]
[14] Erik Hellman, Android Programming: Pushing the Limits, 2014., str. 158-161.

Glavna prednost *RxJave* leži u njenom elegantnom rješenju za upravljanje nitima, omogućujući developerima da jednostavno izvršavaju operacije na odvojenim nitima, transformiraju podatke i dobivaju rezultate bez potrebe za ručnim upravljanjem *AsyncTask*-ovima ili nitima. Ovaj pristup značajno pojednostavljuje asinkrono programiranje u Android okruženju.

2.4.6. Kotlin

Usvajanje Kotlina u Android razvoju donijelo je značajne promjene u paradigmi programiranja. Kotlin stavlja naglasak na nepromjenjivost varijabli, što poboljšava sigurnost niti u višenitnom okruženju. Uvođenje lambda izraza omogućilo je smanjenje količine *boilerplate* koda, posebno pri radu s povratnim pozivima. Dodatne prednosti uključuju data klase za jednostavnije predstavljanje POJO objekata i *sealed* klase za definiranje enum-sličnih struktura koje mogu nositi podatke. Ove značajke Kotlina doprinose čišćem, sigurnijem i ekspresivnijem kodu u Android aplikacijama.[15]

2.4.7. *Dependency injection*

Injekcija ovisnosti predstavlja ključan koncept u modernom Android razvoju, omogućujući razdvajanje stvaranja objekata od njihovog korištenja. Ovaj pristup značajno olakšava jedinično testiranje i omogućuje fleksibilniju implementaciju apstrakcija. Popularne biblioteke za injekciju ovisnosti u Android ekosustavu uključuju Dagger, Koin i Hilt. Svaka od ovih biblioteka nudi jedinstvene prednosti: Dagger pruža robusno rješenje primjenjivo i izvan Android okvira, Koin nudi jednostavniju implementaciju specifičnu za Android, dok Hilt, izgrađen na temelju Daggera, pojednostavljuje proces uklanjanjem *boilerplate* koda i pružajući podršku za Android-specifične ovisnosti.

2.4.8. Android arhitektonske komponente

Android arhitektonske komponente predstavljaju set biblioteka dizajniranih za poboljšanje skalabilnosti, testabilnosti i održivosti Android aplikacija. Ove komponente uvode koncepte poput vlasnika životnog ciklusa, Android ViewModela i LiveData, rješavajući probleme upravljanja stanjem pri uništavanju i ponovnom stvaranju komponenti. Room pojednostavljuje interakciju s SQLite bazama podataka, dok DataStore nudi moderno rješenje za pohranu ključ-vrijednost parova. Uvođenje ovih komponenti potaknulo je široku adopciju *Repository* obrasca, koji centralizira upravljanje različitim izvorima podataka u aplikaciji.

[15] Dmitry Jemerov i Svetlana Isakova. Kotlin in Action, 2017., str. 345-348.

Dodatno, *view binding* i *data binding* olakšavaju interakciju s elementima korisničkog sučelja, povećavajući sigurnost i efikasnost razvoja.

2.4.9. Korutine

Kotlin korutine predstavljaju revolucionarni pristup asinkronom programiranju u Android razvoju. One omogućuju pisanje asinkronog koda na sinkroni način, značajno pojednostavljujući rukovanje konkurentnošću. Korutine nisu vezane za specifičnu nit izvršavanja i mogu se suspendirati i nastaviti, što ih čini izuzetno fleksibilnima. Tokovi, kao nadogradnja korutinama, omogućuju rad s tokovima podataka na reaktivan način, pružajući funkcionalnosti slične RxJavi, ali s boljom integracijom s Kotlin jezikom i Android arhitekturnim komponentama.

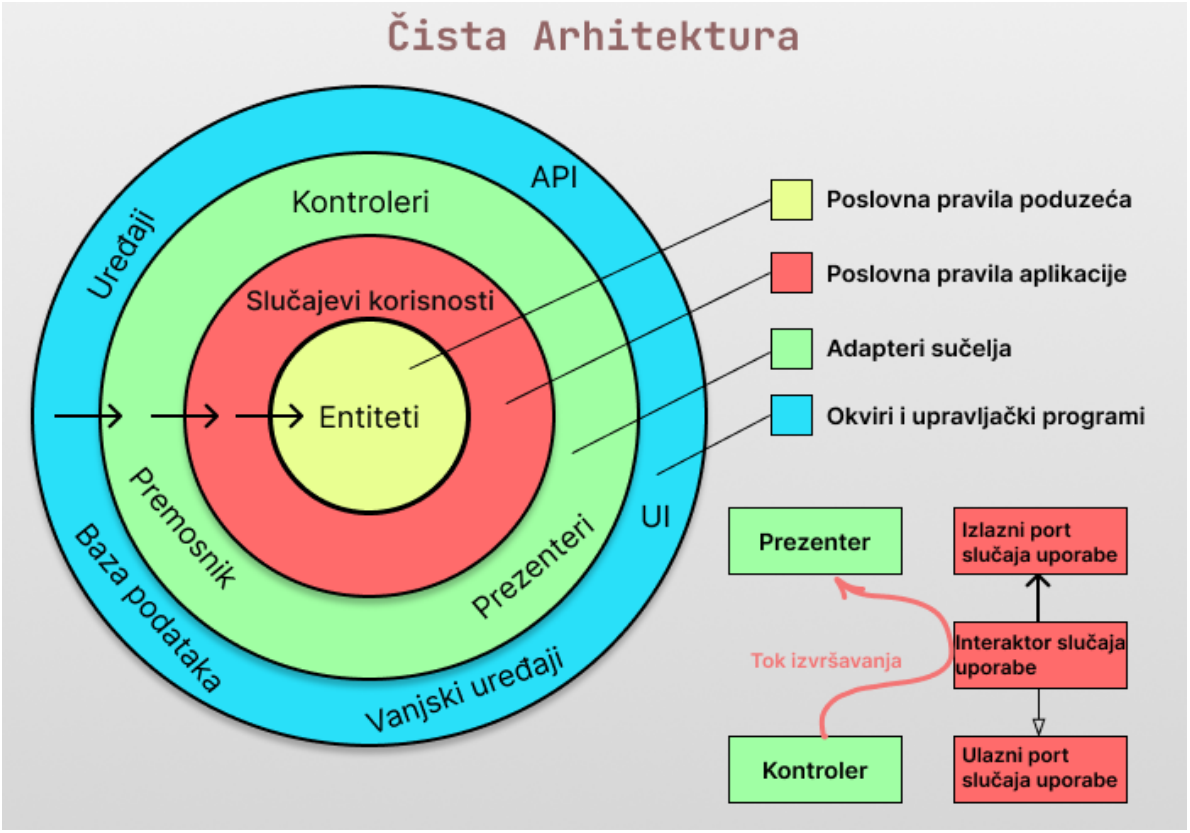
2.4.10. Jetpack Compose

Jetpack Compose predstavlja moderni toolkit za izgradnju nativnih Android korisničkih sučelja. Ovaj deklarativni framework omogućuje developerima da grade UI direktno u Kotlin kodu, eliminirajući potrebu za XML layout datotekama. Compose značajno pojednostavljuje proces kreiranja kompleksnih i dinamičnih korisničkih sučelja, poboljšava performanse renderiranja i olakšava održavanje koda. Njegova integracija s ostalim Android arhitekturnim komponentama omogućuje glatko uklapanje u postojeće aplikacije, pružajući put ka modernijem i efikasnijem razvoju Android aplikacija.

3. Clean arhitektura

Clean arhitektura se može reći da je filozofija koju je prvi definirao Robert C. Martin, koja govori i ističe važnost odvajanja odgovornosti u softverskom dizajnu te kreiranju modularnog projekta koji je jednostavan za testiranje i održavanje kroz duži period. Glavne stavke ovog pristupa su da omogućava dugotrajan razvoj kroz jasno određenu podjelu komponenti i odgovornosti među njima kako bi programeri imali kompaktniji i stabilniji razvojni proces bez puno koraka unatrag. Neke od ideja su posuđene od Hexagonalne arhitekture koja je poznata kao 'Ports and Adapters' koja zastupa pristup odvajanja business logike od vanjskih zavisnosti. Tako da obe arhitekture dijele mišljenje oko jednostavnog testiranja koje je rezultat odvajanja funkcionalnosti aplikacije od vanjskih knjižnica i servisa.

Ova arhitektura se jednostavno može vizualizirati kroz oblik kapule koja predstavlja više razina slojeva, od vanjske apstrakcije prema konkretnim slojevima kako idemo prema unutra.



Slika 1: Prikaz čiste arhitekturerhitekture[16]

Ti slojevi su :

Entiteti – predstavljaju fundamentalni građevni blok poslovne logike aplikacije, utjelovljujući ključne koncepte i pravila domene neovisno o specifičnostima implementacije. Ovi objekti enkapsuliraju najkritičnije poslovne podatke i operacije, održavajući integritet i konzistentnost poslovnih pravila kroz cijelu aplikaciju. Za razliku od tradicionalnih modela podataka, entiteti u Clean arhitekturi nisu puki spremnici informacija, već aktivni nositelji poslovne logike, sposobni za samostalno izvršavanje operacija i validaciju svog stanja.

Usecase - slučaj uporabe, predstavlja ključnu komponentu Clean arhitekture koja premošćuje jaz između apstraktnih poslovnih pravila i konkretnih zahtjeva aplikacije. Ovi elementi enkapsuliraju specifične poslovne procese ili operacije koje aplikacija treba izvršiti,

[16] Grafički prikaz „The Clean Architecture“, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, [pristupljeno 31.svibnja 2024.]

djelujući kao orkerstratori koji koordiniraju interakcije između entiteta i vanjskih slojeva aplikacije.

U kontekstu Android razvoja, *usecase* se često implementira kao samostalne Kotlin klasa, svaka fokusirana na jednu specifičnu funkcionalnost. Ova granularnost omogućuje visoku modularnost i fleksibilnost, olakšavajući održavanje i proširivanje funkcionalnosti aplikacije. *Usecase* tipično sadrži metode koje predstavljaju akcije koje korisnik može izvršiti, poput dohvati podatke korisnika ili izvršiti plaćanje.

Ključna prednost definiranja *usecase*-a je njegova sposobnost da apstrahira kompleksnost poslovne logike od prezentacijskog sloja. Ovo omogućuje da se ViewModel ili Presenter fokusiraju isključivo na upravljanje stanjem korisničkog sučelja, dok *usecase* preuzima odgovornost za izvršavanje stvarnih poslovnih operacija. Ovakva separacija odgovornosti značajno poboljšava mogućnost testiranja aplikacije, omogućujući jednostavno mockanje usecase-ova u testovima prezentacijskog sloja.

Usecase-ovi također služe kao prirodna granica za definiranje granica transakcija i upravljanje asinkronim operacijama, često koristeći Kotlin korutine za efikasno rukovanje dugotrajnim zadacima. Ova komponenta Clean arhitekture osigurava da poslovna logika ostane centralizirana i konzistentna kroz cijelu aplikaciju, istovremeno pružajući fleksibilnost za prilagodbu specifičnim zahtjevima implementacije.

Interfaces layer - poznat i kao adapters layer ili prevedeno kao sloj sučelja, predstavlja ključni dio Clean arhitekture koji djeluje kao posrednik između unutarnjih slojeva (entiteta i usecase-ova) i vanjskih komponenti aplikacije. Ovaj sloj ima kritičnu ulogu u održavanju principa inverzije ovisnosti, osiguravajući da unutarnji slojevi ostanu neovisni o detaljima implementacije vanjskih slojeva.

U kontekstu Android razvoja, sloj sučelja obično sadrži sučelja koja definiraju ugovore za interakciju s vanjskim resursima poput baza podataka, mrežnih servisa ili senzora. Ova sučelja su definirana prema potrebama unutarnjih slojeva, a njihove konkretne implementacije se nalaze u vanjskim slojevima.

Ključne komponente sloja sučelja uključuju:

1. **Repozitorije**: Sučelja koja definiraju metode za pristup i manipulaciju podacima, apstrahirajući izvore podataka od *usecase*-a.

2. **Presenter** ili **ViewModel**: Komponente koje prilagođavaju podatke iz *usecase*-a za prezentaciju u korisničkom sučelju.

3. **Data Transformatore**: Klase odgovorne za konverziju podataka između formata koje koriste unutarnji slojevi i onih koje koriste vanjski slojevi. Ovaj sloj omogućuje fleksibilnost u zamjeni vanjskih komponenti bez utjecaja na poslovnu logiku, olakšava testiranje kroz mogućnost mockanja vanjskih ovisnosti, te osigurava jasnu separaciju odgovornosti unutar arhitekture. U Android aplikacijama, interfaces layer igra ključnu ulogu u održavanju čistoće i modularnosti koda, omogućujući lakše održavanje i evoluciju aplikacije tijekom vremena.

Kontroleri - U kontekstu Android aplikacija, kontroleri u Clean arhitekturi često se manifestiraju kroz ViewModel klase ili ponekad kroz aktivnosti i fragmente. Njihova primarna uloga je koordinacija interakcije između korisničkog sučelja i poslovne logike aplikacije.

ViewModel, kao najčešća implementacija kontrolera u modernim Android aplikacijama, preuzima odgovornost za:

1. Obradu korisničkih akcija iz UI-a

2. Pozivanje odgovarajućih use case-ova

3. Transformaciju rezultata iz use case-ova u format pogodan za prikaz

4. Održavanje i ažuriranje stanja UI-a, često koristeći LiveData ili StateFlow

Ovi kontroleri djeluju kao most između prezentacijskog sloja i domenske logike, osiguravajući da UI komponente ostanu "glupe" i fokusirane samo na prikaz, dok se sva logika i obrada podataka odvija u kontroleru. Ovo omogućuje lakše testiranje, održavanje i skaliranje Android aplikacija u skladu s principima clean arhitekture.

Prezenteri - Prezenteri u čistoj arhitekturi, posebno u kontekstu Android razvoja, predstavljaju ključnu komponentu koja premošćuje jaz između poslovne logike i korisničkog

sučelja. Njihova primarna uloga je priprema podataka za prikaz i upravljanje interakcijama korisnika, istovremeno održavajući poslovnu logiku odvojenom od detalja prezentacije.

U Android aplikacijama, prezenteri tipično:

1. Komuniciraju s *usecase*-ovima za izvršavanje poslovnih operacija

2. Transformiraju rezultate iz domenske logike u format pogodan za prikaz

3. Ažuriraju stanje UI-a kroz definirano sučelje

4. Reagiraju na korisničke akcije proslijeđene iz View-a

Implementacija su prezentera odnosno sučelja često slijedi MVP obrazac, gdje prezenter djeluje kao posrednik između poslovne logike i UI komponenti. Ovo omogućuje strogu separaciju odgovornosti, olakšavajući testiranje i održavanje koda.

Sučelje u Android čistom pristupu doprinosi modularnosti aplikacije, omogućujući jednostavnu zamjenu UI komponenti bez utjecaja na poslovnu logiku. Također, oni apstrahiraju kompleksnost asinkronih operacija od View-a, često koristeći korutine ili RxJava za upravljanje asinkronim tokovima podataka.

Važno je napomenuti da u trenutno novijim Android arhitekturnim pristupima, uloga sučelja često biva zamijenjena ili kombinirana s ViewModel-ima, koji pružaju slične prednosti uz bolju integraciju s Android životnim ciklusom.

Infrastruktura - Infrastrukturni sloj u Clean arhitekturi, posebno u kontekstu Android razvoja, predstavlja vanjski sloj koji se bavi tehničkim detaljima i implementacijom interakcije s vanjskim resursima. Ovaj sloj je odgovoran za konkretnu implementaciju sučelja definiranih u infrastrukturni sloj.

U Android aplikacijama, infrastrukturni sloj tipično uključuje:

1. Implementacije repozitorija: Konkretne klase koje implementiraju sučelja repozitorija, upravljajući stvarnim pristupom podacima iz različitih izvora (baze podataka, mreža, lokalna pohrana).

2. Mrežne komponente: Implementacije API poziva, često koristeći biblioteke poput Retrofit za HTTPS komunikaciju.

3. Komponente za lokalnu pohranu: Implementacije pristupa lokalnim bazama podataka (npr. Room) ili drugim mehanizmima pohrane (SharedPreferences, DataStore).

4. *Maperi* podataka: Klase odgovorne za transformaciju podataka između formata korištenih u vanjskim izvorima i domenskih modela.

5. Implementacije platformskih servisa: Kod koji komunicira s Android-specifičnim API-ima i servisima.

Ovaj sloj izolira ostatak aplikacije od detalja implementacije vanjskih sustava, omogućujući fleksibilnost u promjeni tehnologija ili pristupa bez utjecaja na poslovnu logiku. Infrastrukturni sloj u Android aplikacijama igra ključnu ulogu u održavanju čistoće arhitekture, osiguravajući da unutarnji slojevi ostanu neovisni o specifičnostima Android platforme i vanjskih biblioteka.

4. Korištene Tehnologije

U ovom dijelu rada su navedene najvažnije tehnologije koje su korištene za izradu aplikacije 'City Go' te se odgovorilo na pitanja zašto su baš one odabrane te kako se uklapaju u trenutno moderan razvoj android aplikacija.

4.1. Figma

Figma je alat za izradu dizajna korisničkog sučelja. Besplatan je za osobno korištenje te pruža sve što treba da bi se izradila potpuno profesionalan dizajn spreman za implementaciju. Osim mogućnosti izrade dizajna samog sučelja koje je neophodno za samu izradu aplikacije, moguće je izraditi vrlo interaktivan prototip koji se može doslovno identično ponašati kao aplikacija u samoj produkciji, od animacija pa do navigacije odnosno prikaza *dummy* podataka. Ta funkcionalnost je velika prednost na samom početku razvojnog procesa jer na temelju dizajna može se vrlo lako prilagoditi i prepoznati mogući problemi kao i na *frontend*-u tako i u *backend*-u kada se uz pregled samog dizajna raspišu svi zahtjevi za manipulaciju podataka. No to nije sve, Figma je također vrlo korisna i za razvoj web aplikacija jer pored navedenih

mogućnosti direktno pruža CSS kod što zapravo omogućava direktno preslikavanje dizajna u samom kodu. Taj proces prijenosa dizajna u kod za mobilne aplikacije nije tako jednostavan, ali svakako izrada dizajna odnosno prototipa omogućava lakši razvojni proces. Osim toga, ovaj alat omogućuje timski rad i kolaboraciju u realnom vremenu na način da svi međusobni članovi koji rade na istom projektu međusobno vide kursore i prema tome se lakše organiziraju i podjele rad. Komuniciranje s klijentom jest još jedan izuzetno važan faktor koji treba voditi razvoju softvera. Naime današnje agilne metode razvoja softvera zahtijevaju brzi *feedback* od strane klijenta te upravo iz te problematike, Figma je ponudila rješenje koje omogućava jednostavno generiranje poveznice kako bi se dizajn mogao podijeliti s klijentima ili suradnicima. Ta funkcionalnost štedi vrijeme, poboljšava komunikaciju brzim pregledom najnovije verzije. Zanimljivo je koliko malo vremena je potrebno da se upozna s ovim alatom, no uz ovoliko korisnih funkcionalnosti ima se dosta za svladati kako bi se potencijal ovog alata iskoristio do kraja. No s druge strane za nekog tko nije imao doticaja s dizajnom i sličnim alatima, Figma nudi jako jednostavan ulaz u ovu domenu dizajniranja sučelja s intuitivnim funkcionalnostima koje omogućuju korisnicima da se prvenstveno usmjere na vlastitu kreativnost u tom procesu.

4.2. Android Studio

Android studio je službeni razvojni sustav od Google-a izrađen u koalicij s JetBrainsom. Ovo razvojno okruženje je izrađeno na JetBrainsovom IntelliJ IDEA softveru i posebno prilagođeno za Android razvoj. Kada krećete u sam razvoj Android aplikacija, imate nekoliko opcija, ali definitivno ako se traži posao i budućnost u Android razvoju, odabir je Android Studio. Ima daleko najviše mogućnosti, a zapravo postavlja visok standard svojoj konkurenciji. Pošto uživa direktnu podršku od Google-a, jednostavno integrira i uključuje pristup Google Play uslugama, Firebase platformi, ali i ostalim Google proizvodima i alatima koje olakšavaju razvojni proces. Pored standardnih značajki koje su glavne za pisanje i testiranje koda, ovaj razvojni sustav prati trendove s integracijom AI-a koji pomaže u kreiranju klasa, provjeri postojećeg koda, ali i s novom mogućnosti generiranja koda prema danom projektu, koristeći već napravljene komponente tako da na primjeru slike dizajna iz Figme može ponuditi kvalitetan kod koji predstavlja identičan dizajn. Programeri mogu koristiti emulator i brze preglede kako bi sve promjene u dizajnu mogli prilagoditi bez ponovnog pokretanja, a sam emulator služi za testiranje i provjeru same aplikacije. Još mnogo mogućnosti nosi ovaj alat, no to treba otkriti u procesu samoga razvoja.

4.3. Kotlin – programski jezik

Kotlin kao programski jezik je službeno objavljen u veljači 2016. godine i od tada je *open-source* te se razvija na Github-u od strane tima u JetBrainsu. U posljednjih 8 godina Kotlin je napravio velike pomake pogotovo za razvoj na Android operativnom sustavu gdje je od 2019. zamijenio Javu kao preferirani jezik za razvoj Android aplikacija od strane Googlea. Kotlin danas uživa ulogu moderne Jave koja je statički tipiziran, višeplatformski jezik visoke razine s zaključivanjem tipa. Potpuno je interoperabilan sa Java virtualnim strojem i pored mogućnosti u domeni web razvoja, dominira razvoj mobilnih aplikacija za Android jer se ističe pored Jave svojom jednostavnom sintaksom koja je vrlo ugodna za rad i razvoj. Naravno svaki programski jezik dolazi sa svojim prednostima i nedostatcima, no pored malo sporije kompilacije i ograničenim resursima za učenje, ti problemi se s vremenom smanjuju i vidljiv je trend prijelaza sa Jave na Kotlin, bar što se tiče razvoja Android aplikacija.

4.4. JetPack Biblioteka

Jetpack predstavlja kolekciju Android-ovih biblioteka koje na jedinstven način povezuju i ujedinjuju najbolje prakse u developmentu *native* android aplikacija. Također sve biblioteke su javno dostupne za korištenje na Google Maven repozitoriju i imaju kompatibilnost sa prošlim verzijama android aplikacija. U modernom razvoju Android aplikacija, Jetpack je postao neizostavan član svake aplikacije, ne samo poradi Jetpack Compose-a koji nas oslobađa od XML datoteka kada je u pitanju UI, no tu imamo i Room koji predstavlja način lokalnog spremanja podataka te WorkManager koji omogućuje da unaprijed zakažemo izvršavanje metode u pozadini. Uz to, za lakšu i robustniju navigaciju imamo Navigation *library*, a za korištenje kamere, tu je CameraX. Još mnogo korisnih biblioteka postoji u ovoj kolekciji koju svakako treba iskoristiti. Neke od biblioteka je teško koristiti na početku jer zahtijevaju dodatno razumijevanje sklapanja arhitekture Android aplikacije, a uz to na internetu fali dosta dokumentacije i sadržaja koji bi prikazao i poviše objasnio pojedine implementacije što bi uvelike poboljšalo iskustvo razvoja i korištenja dijelova ove sveobuhvatne biblioteke.

4.5. Firebase Servisi u Oblaku

Firebase je platforma za razvoj mobilnih i web aplikacija koju je razvio Google. Nudi niz alata i usluga koje developerima omogućuju brzu izgradnju visokokvalitetnih aplikacija,

rast korisničke baze i zaradu. Firebase je *Backend-as-a-Service* (BaaS) rješenje koje pruža *backend* infrastrukturu, API-je i alate potrebne za razvoj i pokretanje aplikacija.

Jedna od glavnih prednosti Firebasea je njegova sveobuhvatnost. Nudi širok spektar usluga, uključujući bazu podataka u realnom vremenu (Realtime Database), autentifikaciju korisnika, hosting, cloud funkcije, analitiku, testiranje i još mnogo toga. Ova integracija različitih usluga omogućuje developerima da se usredotoče na razvoj značajki aplikacije, umjesto da troše vrijeme na izgradnju i održavanje pozadinske infrastrukture. Firebase također nudi jednostavnu skalabilnost. Kako se korisnička baza aplikacije povećava, Firebase automatski skalira resurse kako bi zadovoljio potražnju. Ovo je posebno korisno za aplikacije koje doživljavaju nagli rast ili imaju nepredvidljive uzorke prometa.[17]

Realtime Database i Cloud Firestore, dva su glavna proizvoda baze podataka koje nudi Firebase. Oni omogućuju sinkronizaciju podataka u realnom vremenu između klijenata i pružaju offline podršku, što rezultira pouzdanim aplikacijama koje su ugodne za korištenje.

Međutim, Firebase ima i neke nedostatke. Iako je odličan za brzi razvoj i stvaranje prototipa, može postati dosta skup kako aplikacija raste. Firebase naplaćuje na temelju upotrebe, a troškovi mogu brzo porasti za aplikacije s velikim brojem korisnika ili intenzivnom upotrebom podataka.

Još jedno ograničenje Firebase-a je oslanjanje na specifične usluge Googlea. Iako to može biti prednost u smislu integracije i lakoće upotrebe, također znači da ste zaključani u Google-ov ekosustav. Migracija na drugi *backend* ili uslugu može biti izazovna.

Unatoč ovim ograničenjima, Firebase ostaje popularan izbor među programerima mobilnih i web aplikacija. Njegova sveobuhvatnost, jednostavnost upotrebe i brojne značajke čine ga snažnim alatom za brzi razvoj skalabilnih i funkcionalnih aplikacija.

[17] Firebase overview - https://firebase.google.com/docs/overview, [pristupljeno 21. kolovoza 2024.]

5. PRIMJER APLIKACIJE ZA PRUŽANJE USLUGA SELIDBE I TRANSPORTA

Nakon opisa pojedinih tehnologija, slijedi predstavljanje njihove primjene kroz pojedine korake razvoja aplikacije uz diskusiju o prednostima i nedostatcima pojedinih pristupa uzimajući u obzir različite parametre koji utječu na taj proces.

Prvi korak je uvijek ideja. Većini ljudi nije problem smisliti neku ideju što žele raditi, dok drugima to ide malo teže. Ali ideja nije ništa drugo nego ponuđeno rješenje na neki problem s kojim se susreće u svakodnevnom životu. Tih problema ima dosta, ali isto tako ima već dosta postojećih rješenja. Uvijek treba preispitati vlastite ideje, ali i postojeća rješenja iz druge perspektive kako biste dobili bolji uvid u stvarnu korisnost toga rješenja.

Ideja koja je odabrana za ovaj rad se tiče usluga prijevoza i transporta stvari, proizašla je iz iskustva rada u tom području gdje su uočeni brojni problemi, od velikih cijena koje pružaju malobrojni prijevoznici, kvalitete usluga, ali i teškoće s kojim se potražitelji susreću kako bi pronašli odgovarajućeg prijevoznika. To su samo neke od izazova s kojim se ovo područje djelovanja susreće. Važno je pogledati problem s obje strane, a to znači od strane kupca pa do ponuditelja usluge, jer na taj način se detaljno promotri cjelokupni ciklus kako bi jasno prepoznali nedostatke i probleme s kojim se treba suočiti. Poželjno je i proučiti trenutno tržište kako bi vidjeli postoje li već rješenja za vašu problematiku. Dostupne su brojne informacije koje treba koristiti kako bi se unaprijedila početna ideja, odnosno specificiralo rješenje za neki uži problem.

5.1. Minimalno vitalni proizvod

Nakon početne ideje potrebno ju je detaljno razraditi. Treba razmisliti o ključnim točkama koje su važne kako bi se definiralo ponuđeno rješenje. U smislu ove aplikacije je to MVP, to jest minimalno vitalni proizvod koji predstavlja implementaciju vašeg rješenja kako bi uz pomoć kritika od strane korisnika moglo lakše usmjeriti daljnji razvoj aplikacije. Treba suziti listu točaka, gdje svaka točka predstavlja jedan *usecase* koji zahtjeva rješenje kroz aplikaciju.
U smislu ove aplikacije to je:

1. Omogućiti kreiranje profila za korisnike i ponuditelje usluga transporta

2. Kreiranje oglasa, odnosno omogućiti prijevoznicima prijavu dostupnosti na pojedine oglase
3. Omogućiti korisnicima da biraju između više prijevoznika i po odabiru povezivanje korisnika s prijevoznikom kako bi izvršili dogovoreni posao.

Ovo su glavna tri zahtjeva koji će biti riješeni u sklopu ove Android aplikacije jer su oni sasvim dovoljni za implementaciju ove ideje u smislu operativne korisnosti. Naravno, uvijek se može proširiti s nekom idejom i postaviti dodatne funkcionalnosti, ali za početak se treba usmjeriti i ograničiti na glavne funkcije. Ovo su samo zasad operativni zahtjevi koje je potrebno riješiti, no njih je neophodno definirati, jer nam omogućuju daljnji razvoj u polju tehničke implementacije.

5.2. Dizajn korisničkog sučelja

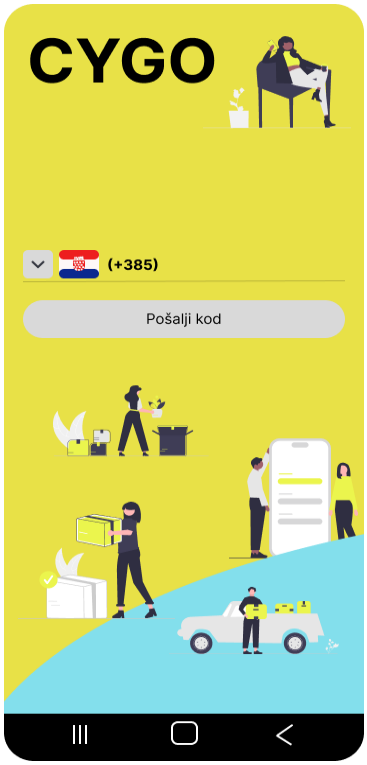
U prethodnom poglavlju je opisan program Figma koji nam omogućuje jednostavno i brzo kretanje u realizaciju ideje kroz sam dizajn koji nam omogućuje daljnje revizije i iteracije prije samog početka razvoja u kodu. Vrlo je bitno proučiti tržište u ovoj fazi kako bi dobili uvid u moguću konkurenciju i vidjeli kako to rade 'profesionalci' na globalnom tržištu te se na taj način inspirirati nekim od već postojećih dizajnova sa stranica kao što je Dribble, Figma ili Youtube videozapisi koji prikazuju neke postojeće realizacije. Potrebno je određenu ideju skicirati na papir te uz svaki screen dodati opis koji govori o čemu bi služio kojih se funkcionalnosti tiče. Nakon toga treba te skice prebaciti u stvarni dizajn pomoću Figme. Slijedi opis kroz dizajn 'City Go' aplikacije prateći realni korisnički *flow*.

5.2.1. Ulazni zaslon

Kao ulazni zaslon obično se koristi nekakav *splash* screen koji ima zadatak predočiti samu temu aplikacije, ali dati i neku ideju samom korisniku što ga to čeka. Taj početni zaslon je vrlo bitan jer je on prvi dojam koji će korisnik dobiti u kontaktu s aplikacijom, a svakako je važan taj prvi dodir s nečim novim, gdje već u prvih 5 sekundi mi ljudi stvaramo predodžbu i mišljenje koje je teško nakon toga promijeniti, zato je presudno uložiti još više vremena u osmišljavanje i dizajn korica aplikacije.

U ovom slučaju treba uzeti još jedan parametar u obzir, a to je jednostavnost korištenja. Naime sama prednost mobilnih aplikacija je u tome da su više responzivne, brže i jednostavnije za korištenje i stoga se može reći da drže brend visoke kvalitete i funkcionalnosti. Zato valja uvijek dobro razmisliti kako razviti aplikaciju u skladu sa današnjim standardom koji podrazumijeva što manje klikova da bi se provela određena akcija u samoj aplikaciji. Iz toga je

odlučeno spojiti taj standardni *splash* screen sa autentikacijom putem telefonskog broja. Prednost ovog pristupa je da se olakšava korisniku sam pristup aplikaciji na način da se skrati standardno maltretiranje kroz unos i verifikaciju emaila, ali i truda oko smišljanja i pamćenja lozinke što svakako nosi dodatne, a u ovom slučaju, nepotrebne korake koji definitivno odbijaju brojne korisnike od korištenja same aplikacije. Pogledom na sliku 2, treba istaknuti i prednosti samog UI-a, jer unos telefonskog broja zauzima jako mali dio ekrana što ostavlja mjesta za predstavljanje aplikacije putem različitih ilustracija. S bojama treba biti konzistentan i jednostavan te ovisno o ciljanoj skupini i problematici kojom se aplikacija bavi, prilagoditi paletu boja kako bi se dočarao pravi karakter aplikacije.



Slika 2: dizajn ulaznog zaslona s unosom telefonskog broja

5.2.2. Početni zaslon

Slika 3: dizajn početnog zaslona aplikacije

Slika 3 prikazuje prvi zaslon nakon uspješne autentikacije, donosi glavni *usecase*, što označava termin koji se često koristi u području informacijskih tehnologija te se koristi kako bi se opisao način na koji korisnik komunicira sa sustavom kako bi postigao određeni cilj. Taj *usecase* predstavlja mogućnost kreiranja oglasa za prijevoz ili selidbu materijalnih stvari. Na vrhu zaslona je istaknuta osnovna poruka korisnicima aplikacije koja stvara dodatno povjerenje. Ispod toga nalazi se informacija o trenutnoj lokaciji korisnika te glavnu karticu koja vodi dalje na proces kreiranja oglasa. Na kartici se nalaze ikone koje šalju jasnu poruku o funkcionalnosti. Također na zaslonu se može naći sažete informacije o kreaciji oglasa kako bi pomogli korisnicima koji prvi put koriste aplikaciju. Na dnu zaslon ističemo lokalne suradnje, koje su u ovom slučaju fiktivne, ali u slučaju aplikacije u produkciji to je dobra ideja jer na taj način usmjerava svjesnost ka čišćem okolišu i važnosti ekologije za poslovne procese koje podupire ova aplikacija.

5.2.3. Zasloni za kreiranje oglasa

Slika 4: dizajn zaslona za unos slike i opisa korisničkog oglasa

Gore na slici 4 prikazan je prvi zaslon nakon klika sa glavne kartice početnog zaslona koji korisnika vodi na prvi od tri koraka prema objavi vlastitog oglasa. U vrhu zaslona imamo grafički prikaz *steppera* koji jasno prikazuje gdje se trenutno nalazi. Grafika kamere predstavlja područje za unos slike gdje je ideja omogućiti korisniku da bira slike iz galerije ili doda novu sliku pomoću kamere kako bi dokumentirao stvari koje želi transportirati i na taj način omogućiti prijevoznicima bolji uvid u obim posla koji je tražen. Uz to korisnici imaju mogućnost unijeti dodatne informacije o poslu koji zahtijevaju.

Slika 5: dizajn zaslona za unos detalja o korisničkom oglasu

U drugom koraku, korisniku se predstavlja forma koju trebaju ispuniti, a koja sadrži podatke o početnoj i dostavnoj adresi. Za svaku adresu treba opisati podatke kao što su ime adrese, kat te ima li stambeni objekt lift ili samo stepenice. Pored toga mogućnost je da vlasnik nije u prilici biti na adresi u trenutku preuzimanja stvari i namještaja, stoga vlasnici imaju mogućnost unijeti šifru ulaznih vrata i kontakt broj u slučaju nekih poteškoća. Pored unosa

adresa, na slici 5 vidljiva je lista koja prikazuje zadana vremenska razdoblja. Važno je specificirati u kojem vremenskom periodu je prihvatljivo preuzimanje od strane prijevoznika, ali također pomoću zadanog okvira koji ne dopušta rezervaciju termina pored danas i sutra, potiče se korisnike i prijevoznike na brzu i hitnu reakciju što bi trebalo smanjiti ne odgovornost kod realizacije posla.

Slika 6: dizajn zaslona za unos veličine i cijene koju je korisnik spreman platiti za traženu uslugu

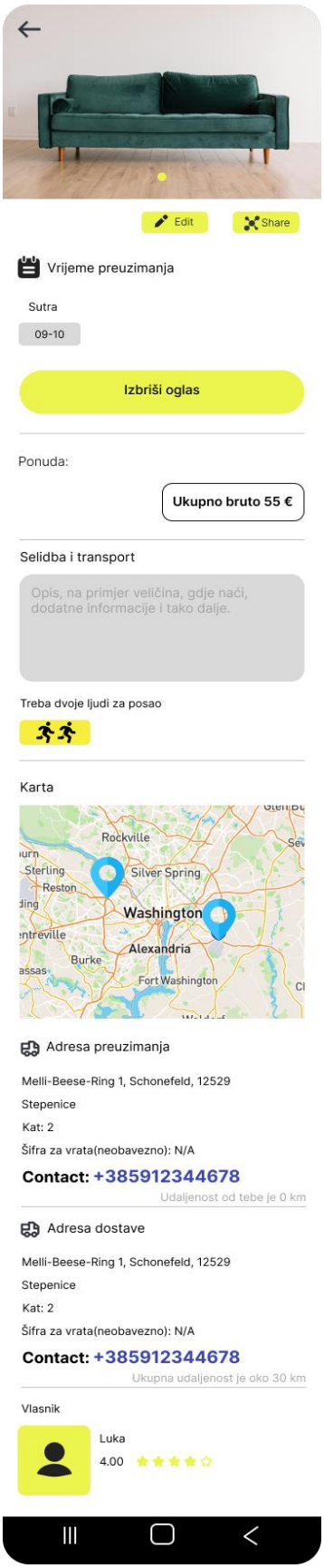
Treći i završni korak ovog procesa, prikazan na slici 6, služi za definiranje cijene, kao i procjenu veličine stvari za koje korisnik zahtijeva prijevoz jer na taj način sa strane dizajna omogućujemo prilagodbu samog algoritma za realni prijedlog cijene koja odgovara standardima tržišta. Uz to korisnik ima mogućnost podijeliti informaciju u slučaju da mu treba pomoć pri iznošenju stvari do vozila prijevoznika te to može učiniti na jednostavan način klikom na odabranu ikonu. Tu proces kreiranje oglasa završava klikom na gumb objavi.

5.2.4. Lista vlastitih oglasa

Slika 7: dizajn zaslona za pregled vlastitih oglasa

Nakon uspješno kreiranog oglasa korisnik može pregledati vlastite oglase te filtrirati između trenutno aktivnih i onih koji su već završeni. Slika 7 donosi zaslon koji prikazuje listu kartica koje sadrže detalje o svakom oglasu kao što su trenutni status, datum kreiranja, adresa i definirana cijena. Žuti gumb u dnu kartice daje mogućnost povećanja cijene ako se čini da nema posebnog interesovanja na predani oglas. U slučaju da postoji prijevoznik koji je spreman ponuditi svoje usluge kako bi realizirao zadani oglas, klikom na gumb moći će se pregledati detalji svih prijavljenih prijevoznika kao i njihove ponude. Cijela kartica omogućuje klik koji vodi na zaslon o detaljima oglasa koji predstavlja kompletan uvid u sve informacije te ga možete vidjeti na idućoj stranici.

5.2.5. Detalji oglasa



Slika 8: dizajn zaslona za pregled detalja o korisničkom oglasu

Prikazano na slici 8, nalazi se naslov koji sadrži sve informacije koje je korisnik dodao te tu ima mogućnost brisanja, promjene i dijeljenja što omogućava rješenje u slučaju pogreške ili promjene plana. Dijeljenjem oglasa može se brže doći do sklapanja dogovora i rješavanja zadanog posla.

5.2.6. Ponude prijevoznika

Slika 9: dizajn zaslona za pregled svih ponuda prijevoznika za odabrani oglas

Zaslon za prikaz ponuda na odabrani oglas daje uvid u popis prijevoznika koji su spremni odazvati se na korisnikov oglas te imaju mogućnost stvoriti protu ponudu na oglas sa cijenom i vremenom koji odgovaranju njihovim preferencijama. Ovakav poslovni model potiče

konkurenciju koja na kraju ide korisnicima u korist jer mogu birati i tražiti kvalitetu te bolju ponudu, dok se prijevoznici moraju maksimalno truditi omogućiti korisnicima kvalitetnu uslugu po prihvatljivoj cijeni. Dizajn prikazan na slici 9 to omogućuje, korisnik ima uvid u broj odrađenih poslova i ocjeni prijevoznika te na temelju tih mjerila kvalitete lakše odabrati kvalitetnu uslugu.

5.2.7. Dogovor za posao

Slika 10: dizajn zaslona za povezivanja korisnika s odabranim prijevoznikom

Na kraju uspješnog odabira prijevoznika dolazimo do kraja poslovnog procesa prikazom zaslona koji se vidi na slici 10, gdje je postignut cilj organiziranja prijevoza namještaja i materijalnih stvari koje vlasnik želi prebaciti. Ovaj zaslon omogućuje vlasniku da nazove i stupi

u kontakt sa odabranim prijevoznikom kako bi se dodatno dogovorili oko završetka posla. Tako mogu razmijeniti dodatne komunikacijske kanale te uvjerili u međusobnu vjerodostojnost.

Ovo predstavlja osnovni, ali i glavni tok aplikacije s korisničke strane koji treba ponuditi bolji pregled u poslovne procese ovog praktičnog rada, no treba napomenuti kako je ovo samo dio cjelokupne aplikacije, sve ovo je daljnje proširivo sa zaslonom za kvalitetu obavljene usluge, zaslonom za prikaz osobnog profila, zaslonom za registraciju prijevoznika te kompletan aplikacijski tok sa strane prijevoznika. Sve to se može pronaći na linku za Figma projekt: https://www.figma.com/design/NZHEVtw3iIhTbHVTeJrvHZ/CityGo?node-id=0-1&t=ktD1T7db9fephTGK-1

Na taj način je predstavljen kompletan dizajn proizašao iz početne ideje za boljom organizacijom prijevoza. Sam dizajn ide više prema MMP dok će se sam kod i razvoj android aplikacije zadržati na MVP-u. To se smatra boljim pristupom kod razvoja aplikacije, da se uvijek prvo napravi proizvod s osnovnim funkcionalnostima koje su istaknute u ovom dijelu, kako bi se što brže došlo do povratnih informacija od stvarnih korisnika kojima se na taj način omogućuje da dožive tu cjelovitost konkretne ideje.

5.3. Baza podataka

Nakon procesa dizajniranja i definiranja poslovnih zahtijeva aplikacije, treba prebaciti fokus na razvoj samog programskog rješenja. Pored cilja da se zadovolje svi poslovni zahtjevi, važno je razmisliti o samoj realizaciji jer svaka tehnologija ima svoja ograničenja i stoga treba pomno istražiti i utvrditi plan izvedbe odmah na početku kako bi se smanjila mogućnost za greške i velike gubitke vremena u idućim fazama.

Jedan od ključnih koraka je definiranje samog modela podataka, ali i vrste baze podataka koja čuva i omogućuje pristup svim podatcima na jednom mjestu. Postoji nekoliko mogućih rješenja kada je u pitanju odabir baze podataka. Najčešće se radi o relacijskoj bazi koja je vrlo robusno rješenje u većini slučajeva. U kontekstu ovoga praktičnog rada je odlučeno implementiranje spremanja podataka na način da korisnik ima pristup dijelu podataka čak i van mreže. To donosi novi izazov koji zahtijeva dvije baze podataka. U sklopu ovoga rada koristi se lokalno android integrirana baza s kojom se komunicira kroz MySQL *wrapper* poznat kao Room. S druge strane u Firebase-u imamo dva moguća rješenja, Firestore koji je preferiran od strane većine te nudi bolje rješenje kada su u pitanju *enterprise* rješenja gdje nudi spremanje

objekata u kolekcije koje se dalje mogu granati na sub-kolekcije koje na kraju omogućavaju moćno filtriranje i dohvat podataka, dok drugo rješenje predstavlja Realtime baza koja je jednostavnija u smislu da sprema jedno veliko JSON stablo gdje su kompleksnije strukture podataka teže za održavanje. Pored osnovnih karakteristika, u okviru ovoga rada želi se prikazati najbolje prakse kod razvoja aplikacijskih rješenja, stoga je odabran Realtime baza jer ona omogućava dohvat podataka kroz Retrofit library putem http metoda, dok Firestore jedino dopušta manipulaciju podataka kroz predefinirane metode što u potpunosti razbija načela *Clean* arhitekture. Poslovni zahtjevi zadane problematike tjeraju programsko rješenje na komunikaciju između više različitih korisnika, a tako i uređaja što znači da je potrebna baza koja je dignuta na server kako bi svi korisnici imali pristup dijeljenim podatcima. A s druge strane trebamo spremati podatke i lokalno u mobilne uređaje svakog korisnika kako bi pojedinci imali pristup podatcima čak i kada internet nije dostupan. Na prvu to se može činiti kao jednostavan zahvat, no svakako treba uzeti u obzir da kada postoje dvije baze podataka, one trebaju dijeliti podatke s korisnicima, ali i komunicirati međusobno kako bi se uspješno sinkroniziralo podatke između njih te korisnicima omogućilo uvid u ažurne podatke u oba slučaja.

Nadalje kod definiranja modela podataka važno je razmisliti o pojedinim entitetima. Najčešće se ovaj proces započinje izradom konceptualnog modela koji odražava strukturu informacija koja se pohranjuje u bazu. Taj model se sastoji entiteta i odnosa te tu olakšavaju posao alati za kreiranje UML dijagrama. Na taj način brzo i jednostavno dolazimo do modela, no u sklopu ovog praktičnog rada preskačemo ovaj korak jer se smatra kako ovo ne treba dodatno objašnjavati i prikazivati pored brojnog sadržaja na mrežnim stranicama. Tu opet treba spomenuti i istaknuti programski jezik Kotlin koji donosi niz integriranih alata koji čine kreiranje dobrog podatkovnog menadžmenta vrlo jednostavan zadatak.

Prije same konkretizacije entiteta, treba opisati strategiju sinkronizacije lokane i udaljene baze podataka. Korištenjem Retrofit HTTPS POST metode koja nazad vraća serverski ID kreiran od strane Firebase-a koji onda služi kako bi se povezao isti zapis u obe baze. Uz to u lokanoj bazi je uz SID potrebno dodatno polje nazvano 'sync' koje se ažurira pri svakoj promjeni podataka zapisa kako bi omogućilo usporedbu između zapisa u lokalnoj i udaljenoj bazi te na taj način ažuriralo zapise u obe baze sa najnovijim podatcima.

U okviru ovog praktičnog rada definirani su idući modeli podataka:

Korisnički oglas ('user_request') – sadržava podatke o poslu koji korisnik zahtijeva, kao što su podatci o početnoj i dostavnoj adresi, prihvatljivom datumu i vremenu obavljanja posla kao i cijeni koju je korisnik spreman platiti. Pored tih informacija, u smislu razvoja programskog rješenja bitno je imati identifikacijski ključ koji se dodjeljuje svakom zapisu u tablici te stoga ovaj entitet nije izuzetak te za lokalnu bazu podataka koristimo 'uuid', no uz to dodajemo ID jedinstveni ključ od regularnog korisnika koji je kreirao taj oglas kako bi na taj način mogli uspješno filtrirati podatke u tablici koja predstavlja dani entitet. No, tu treba spomenuti dodatna dva polja koja su nam važna za sinkronizaciju podataka u obje baze, a to su SID i 'sync' polje.

Realtime baza se razlikuje po dodatnom polju u koji spremamo sve ponude koje se tiču danog oglasa te na taj način vrlo jednostavno prikazujemo detalje svake ponude kroz aplikacijsko sučelje. Naime, da se radi o udaljenoj SQL bazi ne bi bilo potrebno dodavati ovo polje jer onda bi imali veću mogućnost dohvata i filtriranja podataka prema svakom atributu.

Korisnik ('user') – predstavlja svakog korisnika aplikacije koji verificira svoj broj te se u bazu zapisuju podatci o imenu i prezimenu, email, pripadajući broj telefona, ali i profilna slika te rejting. Identifikaciju pojedinog korisnika smo riješili kroz Firebase koji omogućuje kompletnu verifikaciju broja te pruža jedinstveni ključ za svaki autenticirani broj koji se koristi za spremanje ovog entiteta.

Na udaljenoj bazi imamo dodatno polje u koje spremamo listu sa ključevima svih oglasa koji su kreirani od strane istoga.

Prijevoznik ('service_provider') – ovaj entitet proizlazi iz korisnika jer svaki prijevoznik prvo mora kreirati obični korisnički profil kako bi imao mogućnost izvršiti prijavu za kreiranje vlastitog računa prijevoznika kako bi mogli odazvati se na poslove objavljene u aplikaciji. Ovaj entitet dijeli identifikacijski ključ sa običnim korisnikom no sadrži dodatna polja koji pohranjuju kompletan niz informacija kao što su datum rođenja, adresa stanovanja te slike osobne iskaznice i identiteta kako bi sigurnost i ponuda usluga kroz ovu aplikaciju išla glatko i bez većih problema. Uz to imamo podatak o trenutnom statusu koji označava može li prijevoznik krenuti s radom ili se još čeka potvrda podataka predanih u prijavi.

U Realtime bazi imamo dodatno polje koje označava odnos prijevoznika s kreiranim ponudama pa uz svakoga spremamo listu identifikacijskih ključeva svih ponuda koje se vežu uz njega kako bi omogućili jednostavniji upit na samu bazu.

Ponuda ('offer') – se stvaraju od strane prijevoznika koji se na taj način javljaju na oglas za posao. Entitet se veže uz identifikacijski ključ prijevoznika i ključ oglasa za koji je namijenjen. Ponuda je definirana cijenom i vremenskim okvirom kao i statusom.

Primjer entiteta za udaljenu bazu:
https://github.com/XSSmachine/CityGo/tree/master/CityGo/data/network/src/main/java/com/example/network/entities

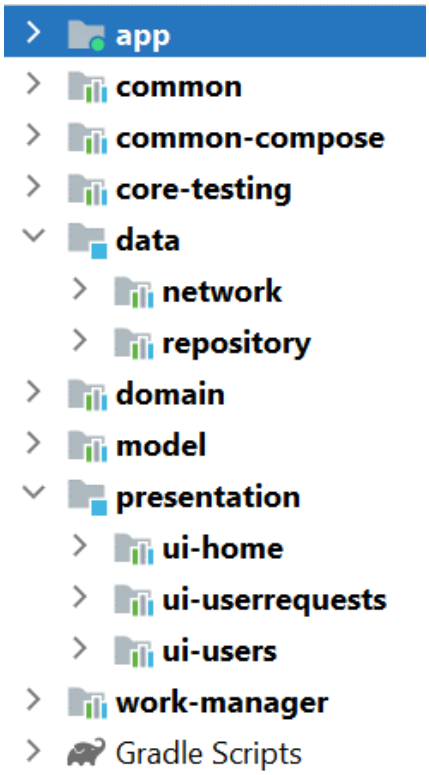
Primjer entiteta za lokalnu bazu:
https://github.com/XSSmachine/CityGo/tree/master/CityGo/data/repository/src/main/java/com/example/repository/datasources/room/entities

5.4. Arhitektura Android aplikacije

Arhitektura Android aplikacije 'City Go' u sklopu praktičnog rada temelji se na MVVM arhitekturi i *Clean* pristupu. Ova kombinacija predstavlja industrijski standard te se ističe brojnim prednostima kao što su modularnost, održivost i skalabilnost aplikacije. Glavna značajka MVVM arhitekture je odvajanje korisničkog sučelja odnosno UI komponenti od poslovne logike, dok je sama poslovna logika neovisna o operacijama nad bazom podataka, što rezultira jasnom podjelom odgovornosti i lakšim razumijevanjem koda. S druge strane, *Clean* pristup promovira neovisnost između modula i komponenti, što ide pod ruku sa MVVM arhitekturom što aplikaciju čini fleksibilnom i otpornom na promjene u tehnologijama u smislu lakše prilagodbe koda u slučaju promjene API-a.

U sklopu ovog praktičnog rada, odlučeno je podijeliti projekt u više modula. Takav pristup ima smisla kod većih aplikacija odnosno aplikacija koje se planiraju širiti u budućnosti. Više modularnost nosi svoje prednosti, jer omogućava da svaki modul bude zasebna cjelina koja se jednostavno može dodati u kontekst nekog drugog projekta te obavljati istu funkciju neovisno o okruženju u kojem se nalazi. Prema tome ovakav pristup je zahtjevan, ali se isplati u budućnosti kada nove projekte možete lako nadograditi sa već razvijenim modulima. Ovaj princip razvoja je direktan primjer dobre prakse u sklopu čistog pristupa odnosno SOLID načelima. Aplikacija se sastoji od tri glavna modula: 'data', 'domain' i 'presentation', no uz to definirali smo još dodatne module kao što su 'common' i 'common-compose' koji služe za definiranje UI komponenti, 'core-testing' koji bi u budućnosti omogućio testiranje na jednom mjestu te 'work_manager' koji predstavlja ideju za buduću implementaciju funkcionalnosti kroz

vlastiti modul. Također treba spomenuti 'app' modul koji je automatski dodan u svaki projekt koji je kreiran u Android Studio te u ovom slučaju koristi nam za određivanje navigacije te pokretanje aplikacije kroz glavni i jedini *activity*. Te model modul je dodan kako bi se izbjegla ciklička ovisnost među glavnim modulima, jer modeli koji se u njemu nalaze su potrebni u sva tri glavna modula. Na slici 11 su prikazani svi moduli koji daju uvid u kompletnu projektnu strukturu.



Slika 11: prikaz svih modula u sklopu praktičnog rada

Data modul ima zadatak pristupiti podatcima i mapirati ih u potrebni model. Ovaj modul sadrži implementacije repozitorija i omogućuje pristup podatcima na jednom mjestu. U sklopu ovog rada, data modul se sastoji od dva modula, 'repository' modul komunicira s lokalnom bazom podataka (Room) za offline funkcionalnost, dok network modul komunicira sa udaljenom bazom podataka (Firebase Realtime Database) za sinkronizaciju podataka u realnom vremenu. Svaki od navedenih modula sadržava entitete koji se pohranjuju u bazu ili mapiraju u modele podataka koji se koriste kroz aplikaciju. Treba spomenuti kako u ovom radu model u aplikaciji je podijeljen u dvije vrste, *response* model koji se koristi za prikaz podataka u aplikaciji i *request* model koji se koristi za spremanje podataka odnosno on se mapira na entitet od svake baze. Kod entiteta lokalne baze potrebno je još definirati ime tablice te primarne

ključeve, dok kod Realtime baze uz pomoć Gson biblioteke serijaliziramo entitet u odgovarajući JSON format. Uz to entitetu treba pridružiti metode za CRUD operacije te se zbog toga definira DAO sučelje za svaku bazu. Zadnja stvar koja je potrebna da bi ovaj sloj bio potpun je *datasource* objekt koji apstrahira i razdvaja metode od svakog entiteta te se time omogućava poziv funkcija iz drugih modula uz minimalnu međusobnu zavisnost. U implementaciji metoda definiranih u svakom *datasource* sučelju, važno je spomenuti rukovanje greškama gdje je korišten try-catch blok za siguran poziv DAO metoda te enkapsulacija rezultata poziva pomoću *state management* klase ili drugim riječima, to je klasa koja ima definirano više stanja kao što su uspjeh, učitavanje, greška te na taj način se omogućava konzistentno rukovanje operacijama. Također svaka metoda je definirana kao *suspend* što označava asinkrono izvođenje bez blokiranja glavne ili UI niti što sprječava zaleđivanje ekrana pri izvođenju dohvata podataka.

Domain modul nije neophodan za realizaciju projekta, no ipak kod kompleksnije poslovne logike je preporučljivo iskoristiti dodatan sloj koji enkapsulira logiku koju na taj način *presentation* sloj može iskoristiti više puta u obliku *singletona*. No to nije sve, *Clean* pristup zagovara jasno razdvajanje odgovornosti i to se definitivno tiče i ovoga sloja koji na našem primjeru služi za definiranje svih *usecase-ova* te time dijeli kod u puno manjih klasa koje su na taj način lakše za održavati i testirati. Uz ovaj modul, pridružen je dodatni modul koji sadržava sve modele za manipulaciju podataka kroz aplikaciju i to iz potrebe da se izbjegne kružna ovisnost među modulima (https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/model/src/main/java/com/hfad/model/UserRequest.kt).

Prvi korak kod realizacije *domain* sloja je bio definiranje sučelja za sve metode koje planiramo vršiti nad pojedinim modelom prateći da naziv svake metode bude u skladu sa željenom funkcionalnošću. Svako sučelje je implementirano kroz klasu gdje se definira poslovna logika koja u ovom primjeru uz standardan dohvat, manipulaciju i mapiranje podataka, podrazumijeva sinkronizaciju između dvije baze usporedbom spremljenog serverskog ključa u lokalnoj bazi odnosno vrijeme ažuriranja podataka kako bi se utvrdilo koji su podatci najnoviji te prema tome odlučilo što gdje treba pohraniti. Nastavlja se pratiti *state* kroz svaki rezultat koji vraća metoda te se logira sve opcije kako bi se jednostavno ustanovilo greške u razvoju.

Nakon toga ide definiranje svih *usecase-ova* koji služe kao dodatan sloj apstrakcije i razdvajanja odgovornosti. *Usecase-ovi* predstavljaju specifične akcije ili operacije koje

korisnik može izvršiti u aplikaciji, a njihova implementacija u domain sloju omogućuje da izoliramo poslovnu logiku od detalja implementacije u *data* i *presentation* slojevima. Proces definiranja *usecase-ova* započinje analizom zahtjeva aplikacije i identificiranjem ključnih funkcionalnosti koje korisnik može izvršiti. Za svaku takvu funkcionalnost, kreira se zaseban *usecase*.

Na primjer, za aplikaciju City Go, neki od *usecaseova* mogu se pronaći na poveznici (https://github.com/XSSmachine/CityGo/tree/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/domain/src/main/java/com/example/domain/interfaces/userrequest_usecases).

Svaki *usecase* implementira se kao zasebna klasa koja obično ima samo jednu metodu, u u ovom radu nazvanu "*execute*", ali može i "*invoke*". Ova metoda prima potrebne parametre za izvršavanje određene operacije i vraća rezultat, omotan u *wrapper* klasu koja prati stanje rezultata (https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/model/src/main/java/com/hfad/model/RepoResult.kt).

Unutar *usecase* klase, implementira se poslovna logika specifična za tu operaciju. To može uključivati pozive prema repozitorijima, validaciju podataka, transformaciju podataka ili bilo koju drugu logiku koja je potrebna za izvršavanje te specifične operacije. *Usecaseovi* također mogu kombinirati pozive prema više repozitorija ili drugih *usecaseova* kako bi izvršili kompleksnije operacije. Važno je napomenuti da *usecaseovi* rade isključivo s domain modelima, a ne s entitetima iz data sloja. Ovo osigurava da domain sloj ostane neovisan o implementacijskim detaljima data sloja.

Presentation modul služi za prikaz podataka i interakciju s korisnikom. U ovom praktičnom radu podijeljen je u 3 modula: 'ui-home', 'ui-userrequests' te 'ui-users'. To je neka osobna preferencija te to nije nešto nužno za okvir arhitekture koji je ovdje opisan. Jednostavno su enkapsulirane cjeline koje imaju sličnosti kao što su obični korisnik i prijevoznik, te korisnički oglas te ponuda. Ovaj modul implementira MVVM arhitekturu, gdje se korisničko sučelje odvaja od poslovne logike prezentacije u obliku ViewModela. *View* je implementiran pomoću Jetpack Composea, modernog toolkit-a za izgradnju nativnog Android UI-a bez potrebe za korištenjem XML-a. Važnost prezentacijskog sloja leži u njegovoj ulozi kao mosta između korisnika i poslovne logike aplikacije. Odgovoran je za prikazivanje podataka korisniku na razumljiv i interaktivan način, kao i za prihvaćanje korisničkih unosa i

akcija. Ovaj sloj također upravlja stanjem tako da UI uvijek odražava trenutno stanje aplikacije.

Slijedi primjer implementacije, gdje je 'DetailUserRequestViewModel' (https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/presentation/ui-userrequests/src/main/java/com/example/userrequest/details/DetailUserRequestViewModel.kt), ključna komponenta koja upravlja poslovnom logikom vezanom uz prikaz detalja korisničkog oglasa. Ova klasa koristi nekoliko *usecase-ova* (https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/presentation/ui-userrequests/src/main/java/com/example/userrequest/details/DetailUserRequestViewModel.kt#L49) za dohvaćanje i manipulaciju podacima, demonstrirajući princip odvajanja odgovornosti.

ViewModel koristi Kotlin korutine za asinkrono izvođenje operacija, što osigurava da UI ostane responzivan tijekom dugotrajnih ili zahtjevnih operacija poput dohvaćanja podataka s mreže. Stanje UI-a se prikazuje se kroz LiveData objekte (https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/presentation/ui-userrequests/src/main/java/com/example/userrequest/details/DetailUserRequestViewModel.kt#L63), koji omogućuju reaktivno ažuriranje UI-a kada se podaci promijene.

DetailScreen composable funkcija

(https://github.com/XSSmachine/CityGo/blob/91f46c790048c2dd5629e2edc3cfc24d36ceab89/CityGo/presentation/ui-userrequests/src/main/java/com/example/userrequest/details/DetailUserRequestScreen.kt), predstavlja View komponentu u MVVM arhitekturi. Ona koristi *state hoisting* princip za upravljanje lokalnim stanjem, dok istovremeno promatra stanje iz ViewModela. Ova funkcija demonstrira kako se kompleksno korisničko sučelje može izgraditi korištenjem Jetpack Compose-a, uključujući prikaz slika, gumbe, tekstualna polja i dijaloge. Uz to valja naglasiti kako se jedan ViewModel može iskoristiti u više View komponenti te na taj način možemo povezati više komponenti koje dijele iste *state* podatke.

Posebno je interesantan primjer 'CreateOfferButton' funkcije, koja realizira način kako se može implementirati uvjetno prikazivanje UI elemenata bazirano na korisničkoj ulozi i stanju aplikacije. Ovo demonstrira fleksibilnost Composea u stvaranju dinamičkih korisničkih sučelja. Nadalje, ova implementacija također pokazuje kako se mogu koristiti *custom composable* funkcije poput 'RatingBar' za stvaranje složenih UI komponenti, što omogućava ponovno korištenje koda i lakše održavanje.

Ovakvom arhitekturom postiže se jasna separacija odgovornosti između pojedinih komponenti sustava. Modularnost arhitekture omogućuje paralelan razvoj i neovisno testiranje pojedinih dijelova sustava, što ubrzava razvoj i povećava kvalitetu koda. MVVM obrazac i *clean* arhitektura promoviraju korištenje najboljih praksi i principa objektno-orijentiranog dizajna, što rezultira čišćim i čitljivijim kodom.

5.5. Implementacija ključnih funkcionalnosti

Autentifikacija korisnika jedan je od ključnih aspekata aplikacije 'City Go'. Za implementaciju autentifikacije korištena je usluga Firebase Authentication, koja omogućuje sigurnu i skalabilnu provjeru identiteta korisnika. Korisnici se mogu registrirati i prijaviti u aplikaciju pomoću svog telefonskog broja, što je intuitivno i lako dostupno širokom krugu korisnika. Proces autentifikacije uključuje slanje SMS koda na korisnikov broj telefona, koji korisnik unosi u aplikaciju kako bi potvrdio svoj identitet. Firebase Authentication SDK pojednostavljuje integraciju autentifikacije u Android aplikaciju i pruža gotova rješenja za rukovanje procesom autentikacije, poput automatskog osvježavanja tokena i odjave korisnika.

Za praćenje statusa pri dohvaćanju podataka s poslužitelja koristi se sealed klasa Result, koja predstavlja tri moguća stanja: *Success*, *Error* i *Loading*. Svaki zahtjev prema poslužitelju omotan je u Result objekt, što omogućuje jednostavno praćenje statusa zahtjeva i prikaz odgovarajućeg korisničkog sučelja ovisno o trenutnom stanju. Na primjer, tijekom dohvaćanja podataka prikazuje se indikator učitavanja, u slučaju uspjeha prikazuju se dohvaćeni podaci, a u slučaju greške prikazuje se poruka o pogrešci. Ovakav pristup omogućuje konzistentno i predvidljivo rukovanje asinkronim operacijama te poboljšava korisničko iskustvo.

Offline funkcionalnost aplikacije omogućena je korištenjem lokalne baze podataka Room. Prilikom prvog pokretanja aplikacije, podaci o dostupnim uslugama dohvaćaju se s poslužitelja i pohranjuju u lokalnu bazu podataka. Sve daljnje operacije, poput pretraživanja i

filtriranja usluga, izvode se nad lokalnim podacima, što omogućuje rad aplikacije bez internetske veze. Kada je uređaj povezan na internet, lokalni podaci se sinkroniziraju s poslužiteljem u pozadini, kako bi se osigurala ažurnost podataka. Ovakav pristup pruža besprijekorno korisničko iskustvo, bez obzira na stanje mrežne povezivosti.

Jedna od ključnih funkcionalnosti aplikacije je mogućnost postavljanja oglasa za pružanje usluga selidbe i transporta. Korisnici mogu kreirati novi oglas unošenjem podataka poput opisa usluge, cijene i lokacije. Prilikom postavljanja oglasa, korisnici mogu dodati i slike koje ilustriraju njihovu uslugu. Slike se pohranjuju u Firebase Storage, skalabilno i sigurno rješenje za pohranu datoteka u oblaku. Nakon uspješnog postavljanja oglasa, podaci se sinkroniziraju s Firebase Realtime Databaseom i postaju vidljivi drugim korisnicima aplikacije u realnom vremenu.

Korisnici mogu pregledavati dostupne oglase i rezervirati usluge koje odgovaraju njihovim potrebama. Proces rezervacije uključuje odabir željenog oglasa, unos dodatnih informacija poput datuma i adrese te slanje zahtjeva pružatelju usluge. Rezervacije se pohranjuju u Firebase Realtime Databaseu, što omogućuje praćenje statusa rezervacije u realnom vremenu. Pružatelji usluga mogu vidjeti pristigle rezervacije te prihvatiti ili odbiti zahtjeve. O svakoj promjeni statusa rezervacije korisnici se obavještavaju putem *push* obavijesti, zahvaljujući integraciji s Firebase Cloud Messaging-om.

6. Rasprava

Razvijena aplikacija 'City Go' predstavlja rješenje za povezivanje korisnika s pružateljima usluga selidbe i transporta. Implementacijom ključnih funkcionalnosti poput autentifikacije korisnika, pretraživanja i filtriranja usluga, postavljanja oglasa, aplikacija značajno olakšava proces povezivanja korisnika s prijevoznicima te omogućava usmjerenu komunikaciju između njih kako bi postigli obostrani dogovor oko nekog posla.

Clean principi i MVVM iskorišteni su za realizaciju ovog programskog rješenja, istaknute su brojne prednosti što kroz teoriju, tako i u praksi. Ali treba razmotriti i potencijalne nedostatke..

Za pojedinačne programere odnosno manje timove ljudi koji razvijaju nešto jednostavnije aplikacije, Clean pristup nije nikako dobar odabir. Na samom početku razvoja, ovakav pristup donosi puno veći obujam klasa i koda koji je na spomenutom primjeru ništa više nego čisti gubitak vremena i resursa. Također ako uzmemo sve principe koji ovaj pristup zagovara te ga slijedimo kao jedino pravo rješenje, može se dogoditi velika zbrka u smislu kompleksnosti koda koja u daljnjim fazama rada otežava dodavanje novih funkcionalnosti, jer izmjene u postojećoj logici zahtijevaju promjenu u sva tri sloja što može biti teško u vremenskom smislu. Ali nije sve tako negativno, sama riječ pristup označava da je pravi način za implementaciju njenih načela zapravo shvaćanje dubljih ideja koje stoje iza pravila koja su spomenuta.

Prava prednost opisane strategije razvoja aplikacije leži u ideji stabilne i ponovljive isporuke proizvoda. Naime clean pristup razdvaja glavne funkcionalnosti i na taj način nakon što se stekne određeno iskustvo, vrlo lako je procijeniti vremenski okvir isporuke danog zadatka i sama kreativnost izrade je minimalna kada imamo već jasnu podjelu odgovornosti.

Ključne značajke praktične aplikacije uključuju autentifikaciju korisnika putem telefonskog broja, što pruža jednostavan i široko dostupan način prijave. Pregled dostupnih usluga omogućeno je kroz dohvat sa Firebase Realtime baze, dok je dohvat vlastitih oglasa realiziran kombinacijom lokalnog pred memoriranja podataka i sinkronizacije s poslužiteljem u realnom vremenu, osiguravajući ažurnost podataka i mogućnost rada bez internetske veze. Korisnici mogu postavljati oglase za potragom usluga, pri čemu se slike pohranjuju u Firebase Storage, a podaci oglasa sinkroniziraju s Firebase Realtime Databaseom. Objava oglasa se automatski prikazuje pružatelju usluge koji onda može poslati ponudu, a statusi oglasa i ponuda prate se u realnom vremenu.

Tijekom razvoja aplikacije korištene su moderne tehnologije i arhitekturni obrasci. Kotlin kao programski jezik pružio je sažetiju i sigurniju sintaksu u odnosu na Javu. Android Studio korišten je kao integrirano razvojno okruženje, pružajući snažne alate za razvoj, testiranje i otklanjanje pogrešaka. Korištenje Firebase servisa za backend usluge značajno je smanjilo potrebu za razvojem i održavanjem vlastitog poslužitelja.

Kao što je rečeno, trenutna verzija aplikacije predstavlja samo MVP odnosno najmanju jedinicu koja se može ponuditi korisnicima kako bi se dobila kvalitetna povratna informacija. Aplikacija ima jako puno mogućnosti za napredak. Naprimjer dobra ideja bi bila implementirati WorkManager library koji bi omogućio rezerviranje Job-a u slučaju da nema interneta prilikom predaje oglasa te bi onda POST metoda prema bazi čekala sve do konekcije s internetom. Pored toga korištenje Firebase Cloud Messaging omogućilo bi slanje *push* obavijesti korisnicima i prijevoznicima o novim ponudama odnosno oglasima te bi na taj način poboljšali UX. Uz to logičan slijed bi bila implementacija Google Maps API-a kako bi prilikom unosa adrese bio automatski ponuđen niz sličnih te da bi se u detaljima oglasa ili kod uspješnog dogovora, grafički prikazala ruta neke selidbe ili transporta. Za ovu implementaciju su već pripremljeni modeli koji sadržavaju polje za unos latitude i longitude što uvelike olakšava ovaj korak. Također pripremljen je teren za implementaciju ocjenjivanja kako bi se nagradila odnosno sankcionirala kvaliteta usluga ili ponašanja u aplikaciji.

Ovaj rad pruža solidnu osnovu za daljnji razvoj i nadogradnju na postojeću implementaciju 'City Go' aplikacije. Zahvaljujući mentoru koji je omogućio komentar od strane senior Android programera, dobivene su vrijedne povratne informacije i prijedlozi za poboljšanje. Ovi prijedlozi se dotiču Android komponenti i knjižnica trećih strana koje su u sklopu ove aplikacije implementirane unutar 'build.gradle' datoteka, no prijedlog bolje implementacije navodi 'version catalog' datoteke za centralizirano upravljanje knjižnicama i ovisnostima što smanjuje vjerojatnost za pogreškom te pogotovo u više modularnom projektu olakšava ažuriranje s jednog mjesta. Preporučeno je napraviti pregled svih korištenih knjižnica i tehnologija koje su korištene kako bi ih se ažuriralo ili zamijenilo s boljom opcijom. Nadalje, opisan je drugi način modularizacije koji zagovara da se pojedini moduli slažu konceptualno, gdje imamo obavezne poput app i data modul, gdje je sadržana osnovna logika aplikacije. No s druge strane bi se dodavali *feature scoped* moduli koji sadrže logiku specifičnu za određeni *usecase,* što omogućuje maksimalnu skalabilnost i ponovno korištenje komponenti. Sugerirana je i optimizacija dohvaćanja podataka iz baze korištenjem 'Flow-a', što bi omogućilo automatsko ažuriranje podataka bez potrebe za manualnim zahtjevima. Predloženo je i izvlačenje dimenzija i teksta u odgovarajuće XML datoteke, što je standardna praksa u Android razvoju. Ove sugestije, zajedno s prethodno spomenutim idejama za proširenje funkcionalnosti, pružaju jasan put za daljnji razvoj aplikacije. Implementacijom ovih preporuka, 'City Go' aplikacija bi unaprijedila svoju arhitekturu, performanse i održivost pružajući bolje korisničko iskustvo te u konačnici omogućiti konkurentnost na tržištu.

7. Zaključak

Razvoj Android aplikacije 'City Go' bio je izazovan i poučan proces. Primjena MVVM arhitekture i Clean pristupa omogućila je stvaranje robusne, skalabilne i održive aplikacije, koja ima svijetlu budućnost. Pored toga, programski jezik Kotlin, kao i Firebase servisi su ubrzali razvoj i smanjili kompleksnost implementacije. Ključni izazovi tijekom razvoja uključivali su dizajniranje intuitivnog korisničkog sučelja, implementaciju sigurne autentifikacije korisnika, osiguravanje pouzdane sinkronizacije podataka između uređaja i poslužitelja te optimizaciju performansi aplikacije. Rješavanjem ovih izazova stečeno je vrijedno iskustvo i znanje u razvoju mobilnih aplikacija. Rezultat ovog procesa je funkcionalna i korisnički orijentirana aplikacija koja rješava stvarne probleme i pojednostavljuje proces povezivanja korisnika s pružateljima usluga transporta. Aplikacija demonstrira potencijal mobilnih tehnologija u transformaciji tradicionalnih industrija i pružanju inovativnih rješenja.

Daljnji razvoj aplikacije mogao bi uključivati proširenje na druge platforme poput iOS-a, integraciju dodatnih usluga i funkcionalnosti te prilagodbu aplikacije specifičnim potrebama različitih tržišta. Kontinuirano praćenje trendova u mobilnim tehnologijama i korisničkim preferencijama ključno je za održavanje konkurentnosti i relevantnosti aplikacije u budućnosti. Ovaj rad prikazuje kako pažljivo planiranje arhitekture, odabir prikladnih tehnologija i pridržavanje najboljih praksi razvoja mogu rezultirati uspješnom i proširivom Android aplikacijom. Stečeno znanje i iskustvo primjenjivo je na širok raspon projekata i pruža čvrste temelje za daljnji profesionalni razvoj u području mobilnog razvoja.

Technologies and mobile app development architecture for the Android operating system on the example of application providing moving and transportation services.

Summary

This work aims to research and describe the development process, first through a theoretical part, and then through the actual creation of an Android application that will allow users to connect and easily communicate with moving and furniture transportation service providers. The technologies that will be covered and used in practice are: the Kotlin programming language, Android Studio development environment, Figma interface design tool, and the comprehensive Firebase application development platform. The external libraries used as part of the practical work are Jetpack Compose, Dagger Hilt, Glide, Gson, and Retrofit. These libraries have primarily enabled a different approach to interface creation without XML files, easy implementation of singletons for all components, and simple manipulation of data and images in communication with the database. The first part describes the purpose and goal of the work, followed by a theoretical section with an overview of approaches to Android application development throughout history, thus describing the theoretical part of individual technologies and approaches such as SOLID and clean architecture. The second part discusses the technologies used, their advantages and disadvantages, as well as the reason for their selection within the framework of this work. The last part concerns the concrete implementation through an Android application where we describe the development process from design to a functional application. The complete application code is available from a public repository on GitHub and can be found at the following link: https://github.com/XSSmachine/CityGo/tree/master

Key words: Android application, Kotlin, Android Studio, Figma, Firebase, Clean architecture

Popis literature

[1] N. Akhtar i S. Ghafoor, *Analysis of Architectural Patterns for Android Development*, 2021.

[2] Google, *Android developer,* 2024., Dohvaćeno iz: https://developer.android.com/guide.

[3] Lucidchart, *Software architecture vs design*, 2024., Dohvaćeno iz: https://www.lucidchart.com/blog/software-architecture-vs-design.

[4] Android developers, *App fundamentals*, 2024., Dohvaćeno iz: https://developer.android.com/guide/components/fundamentals.

[5] A. Dumbravan, *Clean Android Architecture*, 2022.

[6] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 2017.

[7] Huawei developers, *Kotlin SOLID principles tutorial examples*, 2024., Dohvaćeno iz: https://medium.com/huawei-developers/kotlin-solid-principles-tutorial-examples-192bf8c049dd.

[8] ProAndroidDev, *The Stable Abstractions Principle in Android Architecture*, 2024., Dohvaćeno iz: https://proandroiddev.com/the-stable-abstractions-principle-in-android-architecture-de2a4c33dddd.

[9] M. Yener i O. Dundar, *Expert Android Studio*, 2016.

[10] J. Annuzzi Jr, L. Darcey, i S. Conder, *Introduction to Android Application Development: Android Essentials*, 2014.

[11] Gradle, *Kotlin DSL,* 2024., Dohvaćeno iz: https://docs.gradle.org/current/userguide/kotlin_dsl.html.

[12] E. Hellman, *Android Programming: Pushing the Limits*, 2014.

[13] D. Jemerov i S. Isakova, *Kotlin in Action*, 2017.

[14] N. Miroshnychenko, *Resources for learning Clean Architecture in Android*, 2023., Dohvaćeno iz: https://nikolaymiroshnychenko.medium.com/resources-for-learning-clean-architecture-in-android-cf68cd1bdbd0. [Pokušaj pristupa 10. svibnja 2024].

[15] N. Jarad, *Clean architecture in Android and some thoughts*, 2023., Dohvaćeno iz: https://medium.com/@nileshjarad/clean-architecture-in-android-and-some-thoughts-1a9707e3761d.

[16] R. C. Martin, *The Clean Architecture*, 2012., Dohvaćeno iz: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[17] S. P. Kantanami, "Detailed Guide on Clean Architecture," 2023., Dohvaćeno iz: https://medium.com/@satyapavan/detailed-guide-on-clean-architecture-4b4b2f7f0c23.

[18] T. Czura, *The Benefits of Android Clean Architecture*, 2023., Dohvaćeno iz: https://www.netguru.com/blog/clean-architecture-in-android.

[19] A. Tyagi, *Better Android Apps using MVVM with Clean Architecture*, 2023., Dohvaćeno iz: https://medium.com/androiddevelopers/better-android-apps-using-mvvm-with-clean-architecture-c24c56e51c1a.

[20] K. Stefan, *Clean architecture & SOLID principles for Android in Kotlin*, 2023., Dohvaćeno iz: https://www.udemy.com/course/android-architecture-componentsmvvm-with-dagger-retrofit/.