Primjena MVVM arhitekture u razvoju Android aplikacije

Šebetić, Ivan

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:308074

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-11-25**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij računarstva

PRIMJENA MVVM ARHITEKTURE U RAZVOJU ANDROID APLIKACIJE

Diplomski rad

Ivan Šebetić

Osijek, 2024

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Ivan Šebetić
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D-1247R, 07.10.2021.
JMBAG:	0165079352
Mentor:	prof. dr. sc. Damir Blažević
Sumentor:	prof. dr. sc. Tomislav Keser
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	prof. dr. sc. Damir Blažević
Član Povjerenstva 2:	Ivana Kovačević, univ. mag. ing. comp.
Naslov diplomskog rada:	Primjena MVVM arhitekture u razvoju Android aplikacije
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Opisati i objasniti MVVM arhitekturu za razvoj aplikacija. Usporediti s drugim suvremenim arhitekturama razvoja te na primjeru izrade Android aplikacije pokazati prednosti i izazove korištenju MVVM arhitekture. Izraditi aplikaciju.

Datum ocjene pismenog dijela diplomskog rada od strane mentora:	23.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	27.9.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	02.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 02.10.2024.

Ime i prezime Pristupnika:	Ivan Šebetić
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1247R, 07.10.2021.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Primjena MVVM arhitekture u razvoju Android aplikacije**

izrađen pod vodstvom mentora prof. dr. sc. Damir Blažević

i sumentora prof. dr. sc. Tomislav Keser

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

Sadržaj

1. UVOD

Razvoj mobilnih aplikacija često se suočava s različitim izazovima u vidu održivosti gdje aplikacije pri razvoju i rastu koda postaju izuzetno zahtjevne za održavanje, skalabilnosti gdje mobilne aplikacije moraju biti sposobne prilagoditi se većem broju korisnika te njihovim zahtjevima bez degradacije performansi, testiranja u kojem je potrebno osigurati da aplikacija radi ispravno neovisno o vrsti uređaja ili verziji operacijskog sustava, performansi i optimizaciji baterije gdje je ključ održati brzinu i efikasnost uz minimalnu potrošnju baterije te sigurnost kao problem zaštite korisničkih podataka. Tradicionalni pristupi često dovode do navedenih problema, dok *MVVM* (eng. *Model-View-ViewModel)* arhitektura nudi modularnost, bolju organizaciju te lakšu održivost same aplikacije[1].

Glavni zadatak ovog diplomskog rada je prikazati kako primjena *MVVM* arhitekture pruža prostor za razvoj, održivost i skalabilnost samih aplikacija kroz razvoj mobilne *Android* aplikacije za naručivanje hrane. Za primjer prikaza izazova i mogućnosti predmetnog modela (MVVM), razvijena se aplikacija za naručivanje hrane kao praktični dio diplomskog rada. Aplikacija sadržava osnovne funkcionalnosti poput pregleda jelovnika, filtriranja jela, dodavanja u košaricu te procesa narudžbe, čime je prikazan način primjene arhitekture u stvarnom okruženju.

Rad sadržava usporedbu *MVVM*-a s drugim postojećim arhitekturnim obrascima kao što su *MVP* i *MVC* te naglašavajući prednosti i nedostatke svakog od njih. Prvo poglavlje diplomskog rada definira osnovne arhitekturne obrasce koji su korišteni u razvoju mobilnih aplikacija. U drugom poglavlje navode se alati i tehnologije koji se u današnjem svijetu koriste pri razvoju *Android* aplikacija kroz teorijske primjere, a u daljnjem dijelu rada kroz treće i četvrto poglavlje je u praktičnom dijelu prikazana implementacija upravo navedenog obrasca te način korištenja same aplikacije.

Kroz implementaciju funkcionalnog prototipa prikazano je kako ovaj obrazac olakšava razdvajanje poslovne logike od korisničkog sučelja, što je i glavni cilj ovog obrasca čime se dobiva jasniji kod i lakše testiranje. Rad također prikazuje praktične prednosti *MVVM*-a u različitim, specifičnim situacijama mobilnog razvoja uključujući probleme navedene na početku uvoda. Rad je detaljno elaboriran kroz teorijski dio kao i praktični prikaz s ciljem sveobuhvatnog diplomskog rada.

1.1. Pregled područja

1.1.1. Pregled arhitekturnih obrazaca za izradu aplikacija

U razvoju aplikacija postoji nekoliko poznatih arhitekturnih obrazaca koji pomažu organizirati kod, omogućuju održavanje te skaliranje aplikacija. Među najčešće korištenim arhitekturama u razvoju mobilnih aplikacija su *MVC (eng. Model-View-Controller), MVP (eng. Model-View-Presenter)* i *MVVM (Model-View-ViewModel)*, na kojem se temelji i ovaj rad.

- *MVC (Model-View-Controller)*: Ovaj obrazac omogućuje razdvajanje podataka (Model), korisničkog sučelja (eng. *View*) i logike (eng. *Controller*). *MVC* je bio jedan od prvih široko korištenih arhitekturnih obrazaca, no u složenim aplikacijama *Controller* često postaje preopterećen poslovnom logikom, što otežava održavanje.
- *MVP (Model-View-Presenter)*: *MVP* je evolucija *MVC* obrasca gdje *Presenter* djeluje kao posrednik između *View* i *Model* komponenata. Ovaj obrazac nudi bolju organizaciju koda jer *Presenter* sadrži poslovnu logiku, ali može postati previše složen ako se koristi u većim projektima.
- *MVVM (Model-View-ViewModel)*: Ovaj obrazac pruža snažniju separaciju slojeva, omogućujući bolju testabilnost i modularnost. *ViewModel* je ključna komponenta koja omogućuje interakciju između pogleda (*View*) i podataka (*Model*) bez da su direktno povezani. Ovo je trenutno najkorišteniji obrazac za razvoj *Android* aplikacija, a kao takav odabran je i za ovaj diplomski rad. *MVVM* omogućuje jednostavnije povezivanje korisničkog sučelja s podacima, uz veću fleksibilnost u upravljanju korisničkim akcijama.

1.1.2. Pregled postojećih aplikacija za naručivanje hrane

Aplikacije za naručivanje hrane postale su nezaobilazan dio svakodnevnog života. Neke od najpoznatijih aplikacija na tržištu su *Glovo*, *Wolt*, i *Bolt Food*. Ove aplikacije nude široku paletu funkcionalnosti, poput pretraživanja restorana, filtriranja po vrsti hrane, opcija naručivanja i praćenja narudžbi u stvarnom vremenu.

U usporedbi s ovim aplikacijama, aplikacija koja je tema ovog rada predstavlja jednostavniji koncept koji služi prvenstveno kao prototip za prikaz funkcionalnosti temeljenih na *MVVM* arhitekturi. Iako aplikacija uključuje osnovne funkcionalnosti poput pregleda jelovnika, dodavanja jela u košaricu i praćenja narudžbi, nema sve napredne opcije kao komercijalne aplikacije. Njezina svrha je prikazati kako se može izgraditi aplikacija s fokusom na modularnost, skalabilnost i lakoću održavanja kroz primjenu *MVVM* obrasca.

Dok komercijalne aplikacije poput *Glova* i *Wolta* koriste kompleksne backend sustave s mogućnostima integracije s vanjskim API-jevima za praćenje narudžbi u stvarnom vremenu, ova aplikacija nudi jednostavniji pristup s lokalnom bazom podataka te mogućnošću filtriranja i pregledavanja jelovnika. Iako se može smatrati prototipom, pokazuje kako i manji projekti mogu koristiti arhitekturne obrasce za poboljšanje održivosti i fleksibilnosti aplikacija.

2. ARHITEKTURNI OBRASCI U RAZVOJU ANDROID APLIKACIJA

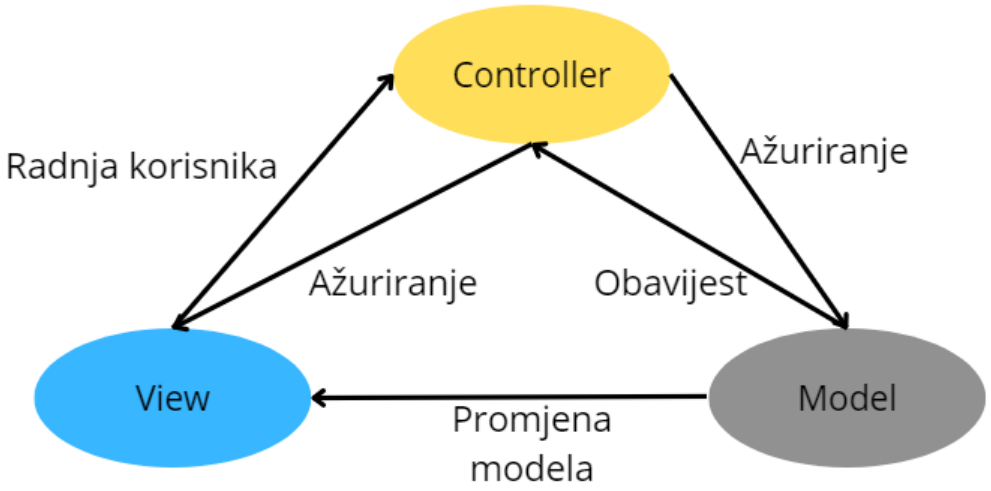
U ovom poglavlju proučeni su osnovni arhitekturni obrasci koji se koriste pri razvoju mobilnih aplikacija. Kroz usporedbu različitih modela, prikazano je kako *Model-View-ViewModel* arhitektura stoji u usporedbi s drugim, postojećim, arhitekturnim obrascima. Poseban fokus je stavljen na prednosti koje pruža spomenuta arhitektura u kontekstu održivosti i skalabilnosti aplikacija.

2.1. Uvod u arhitekturne obrasce

Arhitekturni obrasci predstavljaju temeljnu strukturu potrebnu za razvoj i dizajniranje softverskih sistema, a posebnu važnost imaju u razvoju mobilnih aplikacija gdje kroz skup različitih pravila, tehnika i obrazaca se prikazuje kako bi se aplikacija trebala odvijati. Oni omogućavaju programerima da organiziraju kod na način koji olakšava održavanje, skaliranje i daljnji razvoj kako aplikacije tako i koda. Korištenjem ispravnog obrasca programeri ne samo da mogu efikasno upravljati složenim aplikacijama već se omogućuje i ponovna upotreba koda te jednostavnija i efikasnija suradnja unutar razvojnog tima. Pri odabiru ispravnog obrasca potrebno je obratiti pozornost na nekoliko stvari od kojih su neki odabir platforme na kojoj će se aplikacija razvijati, publika kojoj se plasira proizvod, funkcionalnosti i značajke same aplikacije te vještine tima koji radi na razvoju aplikacije[2]. Značaj arhitekturnih obrazaca proizlazi iz njihove sposobnosti da riješe probleme s kojima se programeri suočavaju pri razvoju i izgradnji složenih sistema gdje se nude različita, provjerena rješenja prateći već definirana pravila. U *Android* svijetu značajnu ulogu imau brzina razvoja i performanse aplikacije, ispravan odabir obrasca može uvelike utjecati na uspješnost samog projekta.
Korištenje arhitekturnih obrazaca u *Android*u nije samo preporučljivo, već je i nužno kako bi se u potpunosti mogla iskoristiti sama *Android* platforma sa svojim specifičnostima kao što su životni ciklusi komponenti, upravljanje memorijom i interakcija s korisnikom, a upravo obrasci pomažu u suočavanju s navedenim specifičnostima, te također, s obzirom na široku paletu uređaja koji se pokrenu na *Android* operacijskom sustavu, pomoću obrazaca omogućeno je stvaranje aplikacija koje su konzistentne te pouzdane u tako širokom spektru uređaja [3].
Kroz ostatak poglavlja je detaljno proučen najčešći i najpoznatiji arhitekturni obrasci koji se koriste pri razvoju *Android* aplikacija, a to su *Model-View-Controller, Model-View-Presenter* te *Model-View-ViewModel*. Svaki od obrazaca je analiziriran kroz njiovu primjenu u *Android* razvoju, navedene su prednosti i nedostatci svakog od njih te su međusobno uspoređeni. Kroz detaljnu analizu obrazaca omogućeno je bolje razumijevanje svakog od njih te je odabir ispravnog obrasca olakšan što je ključ za razvoj dobro strukturiranih, a lako održivih aplikacija.

2.1. Model-View-Controller

Jedan od najstarijih, ali i najpopularnijih arhitekturnih obrazaca u razvoju softvera je *Model-View-Controller*. Posebno je popularan u razvoju kako web tako i mobilnih aplikacija zbog načina raspodjele logike na 3 osnovne komponente čime se pojednostavljuje održavanje i upravljanje kodom [4].



Slika 2.1. Prikaz MVC obrasca

U *MVC*-u model predstavlja podatke i poslovnu logiku aplikacije te kao što slika 2.1. prikazuje model upravlja podacima, pravilima poslovanja, logikom izračuna te pruža metode za ažuriranje stanja podataka unutar aplikacije. *Model* obavještava *Controllera* o promjenama u podacima što je prikazano strelicom „Obavijest“. *View* komponenta je sučelje kroz koje korisnici integriraju s aplikacijom, odnosno, korisničko sučelje gdje se prikazuju podaci korisniku i šalju se korisničke radnje, npr. klikom na gumb se šalje obavijest *Controlleru*. Te *View* također prikazuje ažurirane informacije iz *Modela* što je prikazano strelicom „Promjena modela“. *Controller* djeluje kao posrednik između *View* i *Model* komponente na način da interpretira korisničke akcije dobivene od *Viewa*, upravlja podacima u *Modelu* te ažurira *View* s novim podacima što je prikazano strelicom „Ažuriranje“.

Prednosti *MVC* arhitekture:

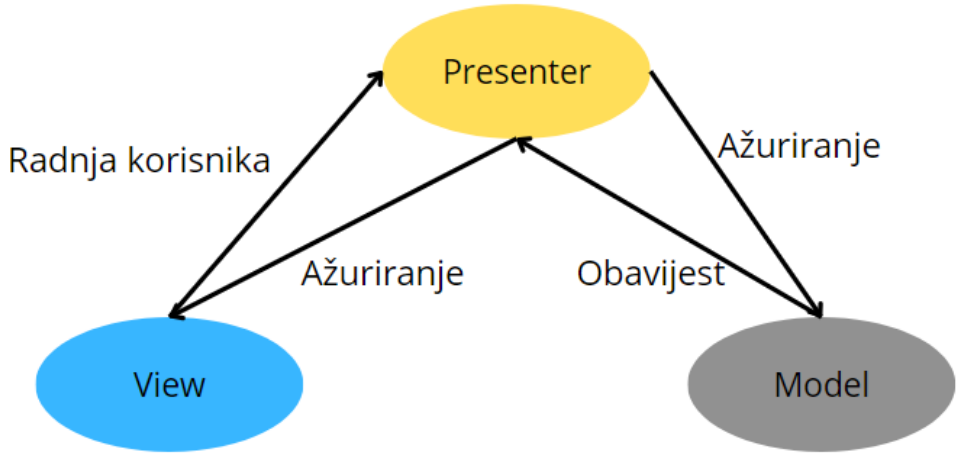
- Razdvajanje odgovornosti: omogućena je jasna granica između logike upravljanja, podataka te korisničkog sučelja čime je olakšano upravljanje kodom, održavanje i testiranje aplikacija jer promjene u jednom segmentu ne utječu izravno na druge segmente.
- Ponovna upotreba koda: s obzirom na odvojenost *Modela*, *View*a i *Controller*a, olakšano je ponovno korištenje i razvijanje komponenti neovisno jedna o drugoj te se mogu ponovno koristiti u nekoliko različitih značajki.
- Prilagodljivost sučelja: kako je korisničko sučelje *View* odvojeno od poslovne logike, lakše ga je ažurirati, nadograditi ili u potpunosti izmijeniti bez potrebe za izmjenom ostatka aplikacije.

Nedostaci *MVC* arhitekture:

- Složenost: *MVC* u velikim aplikacijama s mnogo dinamičnih elemenata i interakcija može postati poprilično složen te kontroleri postanu preopterećeni logikom što u koncu dovede do poteškoća u održava nju.
- Rizik od zastarjelosti: S obzirom na razvoj tehnologija i arhitekturnih obrazaca koji adresiraju neke od slabosti *MVC*-a kao što su *MVVM* i *MVP*, *MVC* u određenom trenutku može postati manje privlačan za moderne aplikacije, posebno one koje zahtijevaju fleksibilnije upravljanje stanjima.

2.2. Model-View-Presenter

Mode-View-Presenter je arhitekturni obrazac koji se razvio iz *MVC* obrasca iz potrebe za boljim odvajanjem sučelja od poslovne logike, a pogotovo u slučajevima kad kompleksnost interakcije korisnika postaje prevelika za *MVC* obrazac što je navedeno kao jedan od nedostataka *MVC*-a. S obzirom na separaciju uloga unutar arhitekture *MVP* obrazac je do nedavno bio jedan od vodećih obrazaca za razvoj web i mobilnih aplikacija [5].



Slika 2.2. Prikaz MVP obrasca

Model je jezgra aplikacije te sadrži podatke s kojima korisnik radi, a sadrži u sebi poslovnu logiku te interakciju s bazom podataka ili nekim vanjskim serverima. Ukoliko dođe do promjene u podacima, *Model* šalje obavijest *Presenteru* da ažurira *View* s najnovijim podacima. *View* označava sučelje kroz koje korisnik interaktira sa samom aplikacijom te je zadužen za prikaz podataka te slanje korisničkih radnji kao što su klik na neki gumb ili unos teksta te se to sve šalje *Presenteru*. U ovom obrascu *Presenter* djeluje kao posrednik te je glavni za ažuriranje podataka tako što šalje zahtjev *Modelu* za njihovu promjenu što se na kraju prikazuje u *Viewu*.

Prednosti *MVP* obrasca:

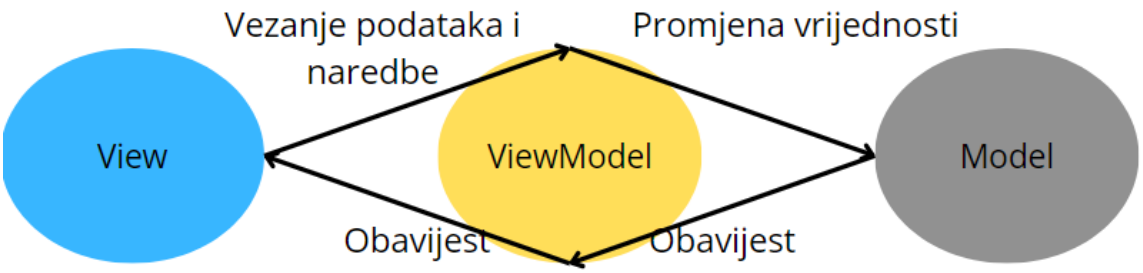
- Razdvajanje odgovornosti: logika prikaza, prezentacije i poslovna logika su jasno razdvojene unutar *MVP* obrasca što olakšava čitljivost koda, lakše testiranje te jednostavnije održavanje.
- Lakše testiranje: s obzirom na odvojenost *Presentera* od *View* komponente nema ovisnosti o korisničkom sučelju te se pojednostavljuje jedinično testiranje.
- Fleksibilnost *View* komponente: u timovima gdje se UI (eng. *user interface*) često mijenja, neovisnost *View* komponente je od velikog značaja jer se može modificirati bez potrebe za izmjenom ostalih komponenti.
- Ponovna upotreba poslovne logike: s obzirom da je sva logika unutar *Presentera*, moguće ju je koristiti kroz različite *View* komponente.

Nedostatci *MVP* obrasca :

- Preopterećenost *Presentera*: pri većim projektima *Presenter* može postati preopterećen logikom što će naposlijetku dovesti do težeg održavanja i razumijevanja koda.
- Složenost održavanja veza: u velikim projektima koji sadrže više ekrana, održavanje veze između *View* i *Presenter* komponente može postati izazovno s čime se povećava prostor za pogreške.

2.3. Model-View-ViewModel

Model-View-ViewModel je omiljeni arhitekturni obrazac koji je posebno dizajniran kao odgovor na moderna korisnička sučelja. Nastao je kao proširenje postojećih *MVC* i *MVP* obrazaca s fokusom na vezivanje podataka čime se omogućuje jednostavniji i efikasniji razvoj aplikacija. Sličan je *MVP* obrascu, ali postoji razlika u komponenti koja sadrži poslovnu logiku.



Slika 2. 3. Prikaz MVVM obrasca

View komponenta predstavlja korisničko sučelje aplikacije koje je u *MVVM* obrascu odgovorno za prikaz informacija te kao i u prethodnim obrascima služi za prikupljanje korisničkih interakcija. Strelica prema *ViewModelu* ukazuje na mehanizam vezivanja podataka gdje se promjene unutar *ViewModela*, automatski odražavaju na *View*, a akcije na *Viewu* se prenost na *ViewModel*. *ViewModel* je u ovom obrascu posrednik koji sadrži logiku prezentacije te ima stanja potrebna za

View. Strelice između *ViewModela* i *Modela* kao i strelice između *ViewModela* i *Viewa* označavaju obavijesti koje se pri promjeni podataka u *Modelu* šalju ka *View* komponenti kako bi se UI mogao osvježiti s najnovijim podacima. *Model* predstavlja poslovnu logiku i podatke same aplikacije. Zadužen je za obradu, pohranu i dohvat poslovnih pravila. Slika jasno prikazuje odvajanje različitih aspekata aplikacije što je temelj *MVVM* obrasca [6].

Prednosti *MVVM* obrasca:

- Vezanje podataka: obrazac omogućuje automatsko i jednostavno vezivanje podataka između komponenti *View* i *ViewModel* što pojednostavljuje ažuriranje podataka elemenata korisničkog sučelja te se smanjuje potreba za ručnom sinkronizacijom podataka već se UI automatski ažurira kad se podaci promjene.
- Testabilnost i modularnost: kako ovisnost između *ViewModela* i *Viewa* ne postoji, razvijanje i testiranje aplikacija je puno jednostavnije.
- Recikliranje koda: s obzirom na odvojenost komponenti koja je već navedena, svaka komponenta se može ponovno koristiti u različitim slučajevima i projektima što dovodi u konačnici do smanjenja napora potrebnog za razvoj sustava.

Nedostaci *MVVM* obrasca:

- Složenost: u projektima gdje vezivanje podataka nije dobro podržano, *MVVM* može biti kompleksan za implementaciju što na koncu dovodi do zamršenog i teško održivog koda.
- Performanse: ukoliko vezivanje podataka nije pravilno implementirano, može doći do negativnog utjecaja na performanse aplikacije jer svaka promjena u podacima može potaknuti i UI ažuriranja iako nisu bila potrebna.

2.4. Usporedba arhitekturnih obrazaca

Model-View-Controller, Model-View-Presenter , i Model-View-ViewModel su arhitekturni obrasci koji se u današnje vrijeme koriste za lakše strukturiranje i organiziranje koda pri razvoju softvera. Svaki od navedenih obrazaca ima već navedene mane i prednosti, ali ipak postoje određene situacije kada je najprikladniji za određeni projekt.

MVC je najstariji obrazac od navedena tri te je najidealniji u aplikacijama koje imaju jednostavno korisničko sučelje gdje su korisničke interakcije direktno povezane s prikazom podataka. Jednostavno se razdvaja poslovna logika od korisničkog sučelja čime je olakšan razvoj i održavanje aplikacije, ali ukoliko bi aplikacija postala previše kompleksna moglo bi doći do preopterećenja *Controllera* što bi otežalo upravljanje.

MVP je stvoren kao odgovor na neke od postojećih nedostataka *MVC* obrasca tako što pruža još bolje odvajanje korisničkog sučelja i poslovne logike te je koristan za aplikacije koje imaju nešto složenije interakcije između korisnika i sustava. *Presenter* preuzima većinu poslovne logike čime se olakšava testiranje i održavanje, ali isto u određenom trenutku može doći do zasićenja. *MVVM* kao najnoviji i u posljednje vrijeme i najpopularniji obrazac s obzirom da aplikacije imaju sve bogatije korisničko sučelje s potrebom za dinamičkim ažuriranjem ima veliku prednost dvosmjernog vezivanja podataka čime se znatno olakšava ažuriranje korisničkog sučelja. *ViewModel* kao snažan posrednik između *Modela*i *Viewa* upravlja interakcijama i ažuriranjem čime se postiže visoka reaktivnost aplikacije. Iako može biti složen za implementaciju, njegove prednosti u složenim korisničkim interakcijama čine ga idealnim za današnje aplikacije. Uzevši u obzir sve do sad navedeno, *MVVM* obrazac je izabran kao odgovarajući za izradu aplikacije za naručivanje hrane zbog njegove sposobnosti da efikasno i jednostavno upravlja sučeljem te omogućava bolju organizaciju koda, lakše održavanje i ono što je najbitnije, daljnji razvoj.

3. ALATI I TEHNOLOGIJE U RAZVOJU ANDROID APLIKACIJA

Ovo poglavlje bavi se alatima i tehnologijama koji su ključni za razvoj *Android* aplikacija. Kroz poglavlje su razmotreni pojmovi kao što su operacijski sustav, razvojno okruženje, odabrani programski jezik, sustav za pohranu i upravljanje kodom. Detaljno je istražen svaki od alata te je prikazano na koji način doprinosi boljoj organizaciji i održivosti koda.

3.1. Operacijski sustav Android

Prema [7], *Android* OS je razvijen kao operacijski sustav za mobilne uređaje, tablete, televizore i automobile, a zasnovan je na Linuxu. Osnove razvoja su temeljene na Linux jezgri koja predstavlja apstraktni sloj između programskih i harvdverskih slojeva. Prema [8] *Android* Inc. osnovan 2003. godine s glavnim ciljem razvijanja programa za mobilne uređaje, ali kako su uvidjeli prostor u operacijskim sustavima, odlučili su raditi na razvijanju vlastitog te su kupljeni od strane *Google*-a 2005. godine. Od predstavljanja sustava 2007. godine, *Android* sustav slovi za jedan od najraširenijih operacijskih sustava uz Apple-ovu inačicu operacijskog sustava, a na popularnosti je dobio zahvaljujući njegovoj fleksibilnosti, prilagodljivosti i podršci od velikog broja proizvođača uređaja.

Neke od ključnih karakteristika *Android* operacijskog sustava su:

- Otvoreni izvorni kod: najvažnija karakteristika *Androi*da jest upravo otvorena izvorna kodna struktura koja omogućava programerima da prilagode i prošire sustav prema njihovim vlastitim potrebama.
- Velika tržišna raznolikost: u suradnji s velikim brojem proizvođača, *Android* operacijski sustav se nameće na velikoj većini dostupnih mobilnih uređaja što programerima nudi pristup širokoj bazi korisnika.
- Podrška za različite vrste uređaja: iako slovi kao operacijski sustav za mobilne uređaje, *Android* je svoje poslovanje proširio i na ostale uređaje kao što su televizori, automobili i nosivi uređaji.
- *Google* Play Store: aplikacija koja korisnicima nudi najveći mogući broj aplikacija za preuzimanje, a osim Play Store-a, podržava i druge kanale za preuzimanje aplikacija.
- Razvojni alati i resursi: Veliki broj razvojnih alata znatno olakšava programerima da efikasno razvijaju, testiraju i plasiraju na tržište aplikacije, a neki od alata su *Android* Studio, *Android SDK*(eng. *Software Development Kit*) i *NDK*(eng. *Native Development Kit*).

Kroz svoju raznolikost, otvorenost i dostupnost, *Android* je jedan od glavnih voditelja i postavljača temelja za mobilnu industriju u kojoj ne samo da potiče daljnji razvoj postojećih rješenja, već na tržište redovno plasira i tehnološke inovacije čime ostaje ključ u svijetu mobilnih tehnologija.

3.2. Android studio i razvojno okruženje

Android studio je integrirano razvojno okruženje koje se koristi za razvoj mobilnih aplikacija za operacijski sustav *Android* tvrtke *Google*, a izgrađeno je na softveru *IntelliJ IDEA* tvrtke *JetBrains* [9]. Osim za razvoj mobilnih aplikacija, na njemu se mogu razvijati i aplikacije za tablete, TV uređaje te automobile. Preuzimanje *Android* studija je u potpunosti besplatno te je dostupan za različite operacijske sustave kao što su Windows, Linux te *macOS*. Od programskih jezika moguće je programirati u jezicima *Kotlin* i *Java,* a *Kotlin* je u posljednje vrijeme preporučeni jezik za programiranje aplikacija. Prva stabilna verzija je objavljena nakon više od godinu dana nakon njegove najave 16. svibnja 2013. godine.

Neke od ključnih značajki *Android* Studija su:

- Emulator: *Android* Studio dolazi s unaprijed konfiguriranim emulatorom koji omogućuje brzo i lako testiranje aplikacija bez potrebe za hardverskim uređajima.
- Integrirani *Android* SDK Manager: osigurava da su svi razvojni alati uvijek ažurirani s najnovijim dostupnim API-jima te omogućuje lako preuzimanje dostupnih alata i dokumentacije.
- *Visual Layout Editor*: omogućava jednostavno kreiranje korisničkog sučelja metodom *Drag and drop* bez potrebe za dodatnim, ručnim kodiranjem XML-a.
- Podrška za *Kotlin* i *Javu: Android* Studio nudi podršku za ova 2 jezika iako je *Kotlin* preferirani programski jezik za razvoj *Android* aplikacija, ali nudi podršku i za *Javu* čime programerima daje dodatnu fleksibilnost pri razvoju aplikacija.

S obzirom na veliku količinu dodatnih alata koje *Android* Studio nudi, programeri imaju različite mogućnosti prilikom kreiranja vlastitih aplikacija kao što su praćenje performansi aplikacije, problemi s memorijom ili brzinom izvođenja određenih zadataka čijom se detekcijom aplikacija može maksimalno optimizirati. Prilikom kreiranja projekta u *Android* Studiju, automatski se kreira i *Gradle* datoteka koja se koristi za automatizaciju gradnje aplikacije te uvelike omogućava dodavanje vanjskih biblioteka. *Android* je poznat po svojim inačicama OS-a koje imaju naziv prema desertima dok je *Android* Studio svoje inačice nazivao po životinjama, a zadnja verzija jest *Koala.*

3.3. Programski jezik Kotlin

Kotlin [10], moderni programski jezik razvijen je od strane *JetBrainsa* 2011. godine kao „*open source"* projekt te je brzo stekao popularnost u svijetu *Android* razvoja nakon što ga je *Google* priznao 2019. godine kao preferirani jezik za razvoj aplikacija. *Kotlin* je moćan programski jezik koji nudi pregršt alata koji *Kotlinu* daju prednost u odnosu na *Java*-u. Ono što je velika prednost jest upravo interoperabilnost s Javom što znači da se može koristiti zajedno s postojećim *Java* kodom čime se olakšava ulazak *Kotlina* u postojeće projekte koji su rađeni u *Java* programskom jeziku. Neki od prije spomenutih alata su asinkrono programiranje, ekstenzijske funkcije, lambda izrazi te mnogi drugi alati koji pomažu pri pisanju kodova koji su lakši za testiranje, manji skloni greškama, a i sam jezik je izražajan što znači da u trenutku kompajliranja i pisanja koda svaki izraz mora biti poznat kompajleru čime se dolazi do ranog otkrivanja pogrešaka gdje i sam kompajler u određenim trenucima predlaže rješenje.
Što se tiče povijesti i razvoja samog jezika, naziv dolazi od otoka *Kotlin* koji se nalazi u Finskom zaljevu, a želja za takvim imenom proizlazi upravo iz imena jezika od kojeg se *Kotlin* i razvio, a to je *Java* koja je dobila ime po indonezijskom otoku Java. Prva stabilna inačica *Kotlin* programskog jezika plasirana je u veljači 2016. godine, a trenutna najnovija inačica jest 2.0.20 gdje se sa svakom novom inačicom dodaju nove funkcionalnosti kako bi se programeru što više olakšalo pisanje koda. Iako se *Kotlin* uvijek spominje kao nasljednik *Jave* postoje još neki jezici koji su imali svoj utjecaj na razvoj *Kotlina*, kao što su *Scala* gdje je *Kotlin* preuzeo mogućnost kombiniranja objektno orijentiranog i funkcionalnog programiranja, a zadržao brzinu kompajliranja *Jave*, te je od *Groovya Kotlin* preuzeo sposobnost pisanja već spomenutog izražajnog koda kako bi sintaksa bila orijentirana na produktivnost.

3.4. Github sustav

Github sustav je vodeća platforma koja služi za hosting i upravljanje softverskim kodom. Radi se o servisu koji omogućuje ugošćivanje(eng. *hosting*) Git repozitorija u oblaku. Osnovan 19.listopada 2007. godine stekao je veliku popularnost te trenutno ima preko 100 milijuna korisnika, preko 420 milijuna kreiranih repozitorija te preko 28 milijuna javnosti dostupnih repozitorija što ih čini najvećim sustavom za pohranjivanje koda. Github se koristi za međusobnu suradnju programera gdje programeri mogu izmjenjivati kodove, pohranjivati svoje radove te dobiti pregled svojih projekata od strane drugih programera [11].

Neke od mnogih značajki GitHub-a:

- Repozitorij: omogućeno je korisnicima stvaranje repozitorija gdje se kod pohranjuje i njime se upravlja. Unutar repozitorija sprema se cijela povijest izmjene projekta čime se omogućuje lako praćenje promjena te je omogućena suradnja na kodu bez rizika od gubitka koda.
- Grananje i spajanje: razdvajanje, odnosno grananje(eng. *branching)* i spajanje(eng. *merging*) repozitorija na više grana omogućuje programerima da rade na određenom dijelu projekta bez utjecaja na ostatak projekta što stvara prostor za eksperimentiranje.
- *Pull request*: prije samog integriranja promjena projekta, koristeći *Pull request* programerima je omogućeno da obavijeste ostatak tima o promjenama koje su izvršili te se promjene mogu pregledati prije same integracije čime se mogu izbjeći neželjeni problemi pri izmjenama i nadogradnjama.

U današnjem svijetu je nezamislivo zamisliti projekt koji je finaliziran bez korištenja GitHub servisa kako za pojedinca tako i za veće timove. Ne samo da je olakšano upravljanje kodom već je potaknuta i međusobna suradnja programera gdje se međusobno mogu dijeliti različite ideje i projekti čime se uvelike povećava efikasnost programera i unaprjeđuje se kvaliteta koda koji se razvija.

4. DIZAJN I IMPLEMENTACIJA PROGRAMSKOG RJEŠENJA

U ovome poglavlju fokus je na dizajnu aplikacije i implementaciji programskog rješenja koristeći MVVM arhitekturu. Prikazan je proces planiranja strukture aplikacije, organizacija koda te povezivanje komponenti *Modela, Viewa* i *ViewModela*. Poglavlje obrađuje ključne tehničke odluke donesene tijekom izrade aplikacije te kako one utječu na učinkovitost i održivost rješenja.

4.1. Specifikacija funkcionalnih zahtjeva

Kako bismo lakše shvatili bit i rad aplikacije potrebno je jasno i temeljito definirati ciljeve i kriterije koje aplikacija treba zadovoljiti. Ciljevi i kriteriji se formaliziraju kroz funkcionalne i nefunkcionalne zahtjeve koje je potrebno razumjeti kako bi aplikacija u potpunosti bila funkcionalna te kako bi se zadovoljile potrebe korisnika. Aplikacija za naručivanje hrane namijenjena je krajnjim korisnicima, odnosno osobama koje žele naručiti hranu iz restorana na brz, jednostavan i učinkovit način, ali u budućim, proširenim verzijama aplikacija može pružiti potporu i za nove korisnike. Glavni korisnici aplikacije uključuju:

- Kranji korisnici: osobe koje žele pregledati ponudu, cijene, odabrati jelo i na koncu ga i naručiti hranu. Korisnici traže brzu, jednostavnu aplikaciju koja je intuitivna s obzirom da sadrži osnovne informacije vezane za ponudu jela.
- Restorani: iako trenutna verzija aplikacije ne podržava posebnu ulogu restorana, u budućnosti bi se aplikacija mogla nadograditi na način da se restoranima omogući vlastito upravljanje jelima, narudžbama i korisničkim podacima čime bi restorani automatizirali proces naručivanja hrane.

Kad korisnik pokrene aplikaciju prikazuje mu se kompletna ponuda jela koju restoran nudi uz mogućnost filtriranja jela ovisno o kategoriji u koju su jela svrstana. Svako od tih jela sadrži dodatne informacije koje korisnik dobiva pritiskom na odabrano jelo gdje dobiva informacije o sastojcima što može u isto vrijeme služiti i kao lista alergena, dobiva kratak opis jela, cijenu i ono što je u današnje vrijeme bitno, realnu fotografiju onoga što može očekivati pri narudžbi odabranog jela. Kad se korisnik odluči za određeno jelo, jednostavnim pritiskom na gumb u obliku kolica koji simbolizira kupovinu, proizvod se automatski dodaje u postojeću košaricu u koju korisnik u svakom trenutku ima mogućnost pristupa. Nakon što korisnik završi s odabirom jela, odlazi u košaricu gdje stoje sva jela za koja se odlučio i sva su prikazana na način kao i što su na prvom zaslonu osim što unutar kartice imaju i gumb kolica koji služi za uklanjanje jela. Ukoliko se u određenom trenutku korisnik odluči ukloniti neko od jela iz košarice to radi na način da ili direktno kroz košaricu pritisne gumb za uklanjanje jela čime se bez potrebe za osvježavanjem zaslona to jelo i uklanja sa zaslona, ili

ulaskom u samo jelo te pritiskom na gumb u gornjem desnom kutu uklanja jelo iz košarice. Nakon što korisnik bude zadovoljan sa sadržajem košarice, potvrđuje svoju narudžbu pritiskom na „*Order"* gumb čime se odabrana jela uklanjaju iz košarice, odlaze u povijest narudžbi gdje korisnik opet ima uvid u sve što je u određenoj narudžbi imao. Cijela aplikacija ostvarena je na *MVVM* arhitekturi koja je i temelj ovog diplomskog rada gdje je jasno podijeljena odgovornost između različitih segmenata aplikacije čime je aplikacija jednostavna za održavanje, proširivanje i testiranje. Unutar aplikacije *Model* je zadužen za podatke aplikacije kao što su informacije o jelima i narudžbama te je odgovoran za pristup i manipulaciju podacima, bilo iz lokalne baze podataka ili iz udaljenih izvora. *View* je sučelje s kojim korisnik komunicira te je zadužen za prikaz podataka i obavještavanje *ViewModela*o korisničkim akcijama kao što su pritisci na različite, navedene, gumbe. *ViewModel* kao posrednik između navedenih *View* i *Modela*upravlja podacima i logikom poslovanja, obrađuje akcije koje korisnik inicira, dohvaća podatke te ih prosljeđuje na korisničko sučelje čime se na jednostavan način upravlja promjenama stanja kao što su jela u košarici, a podaci su u svakom trenutku ažurni. Korištenjem *MVVM* arhitekture u ovom projektu osigurane su odvojene odgovornosti tako što se razdvajaju poslovna logika, korisničko sučelje i podaci. Olakšano je proširenje aplikacije s novim funkcionalnostima bez potrebe za promjenama postojećeg koda te je testiranje pojednostavljeno zbog odvojenih, već navedenih, odgovornosti.

Funkcijski zahtjevi na aplikaciju su sljedeći:

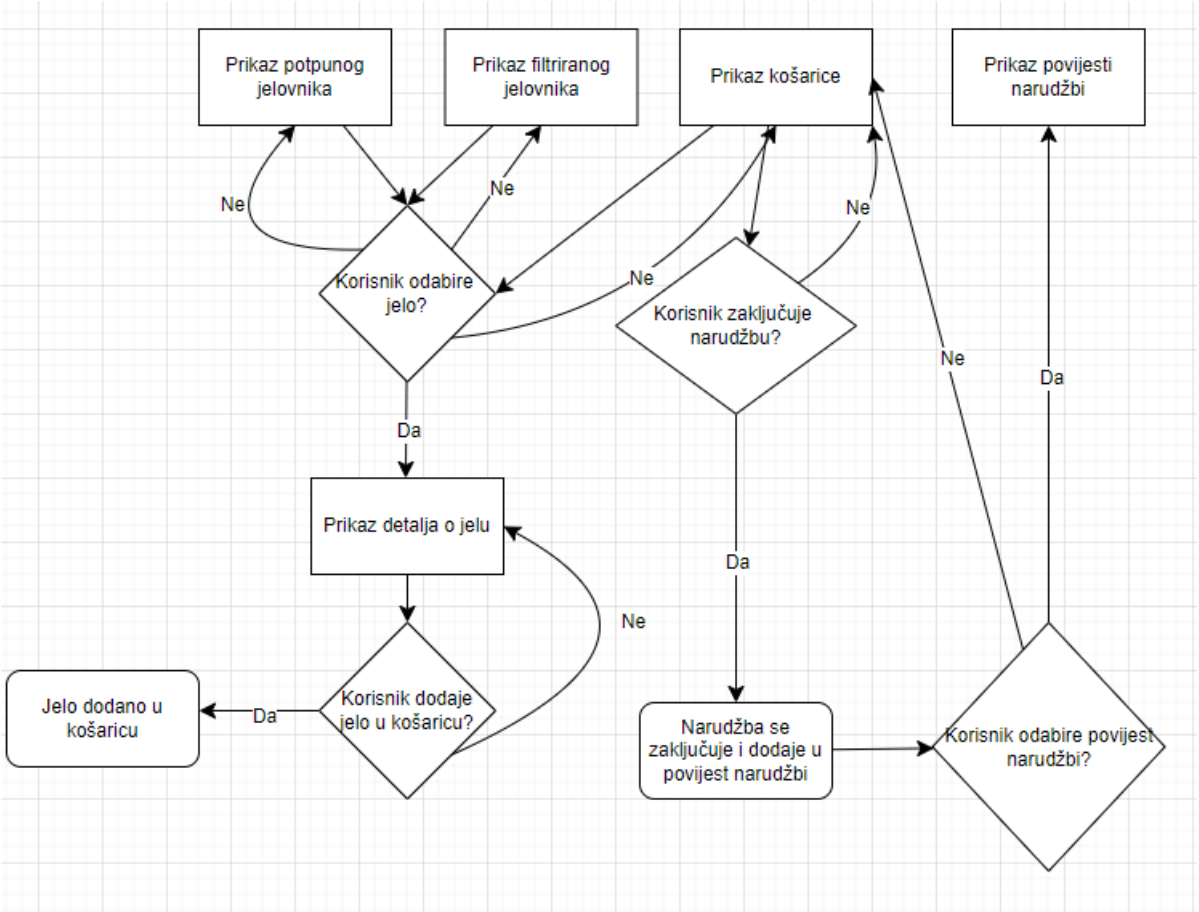
- Prikaz jelovnika gdje korisnik u svakom trenutku ima uvid u dostupna jela uz mogućnost filtriranja.
- Prikaz detalja o jelu gdje korisnik za svako pojedino jelo može vidjeti dodatne detalje o njemu kao što su cijena, sastojci, kratak opis jela i slika jela.
- Dodavanje jela u košaricu u trenutnu narudžbu.
- Prikaz košarice trenutne narudžbe gdje se nalazi popis svih odabranih jela.
- Uklanjanje jela iz košarice za trenutnu narudžbu.
- Zaključivanje narudžbe pritiskom na gumb „*Order"*.
- Pregled povijesti narudžbi s uvidom u sve prethodne narudžbe.

Osim funkcionalnih zahtjeva koji opisuju ključne funkcionalnosti aplikacije koje ona mora omogućiti korisnicima kako bi postigli sve svoje ciljeve, od pregleda jelovnika do pregleda narudžbe i finaliziranja iste, nefunkcionalni zahtjevi [12] su ključni za osiguravanje kvalitetnog korisničkog iskustva. Nefunkcionalni zahtjevi su oni zahtjevi koji su opisani sa sustavom ili prema sustavu,

odnosno objašnjavaju kako se programska podrška mora ponašati u određenim situacijama kao što je npr. veliki broj korisnika, brzina aplikacije, održavanje. Što se tiče konkretno ove aplikacije oni su sljedeći:

- Performanse: potrebna je brza reakcija aplikacije na korisničke akcije kao što su učitavanje jelovnika, dodavanje jela i zaključavanje narudžbe što je osigurano korištenjem asinkronih operacija kako bi se izbjeglo blokiranje korisničkog sučelja.
- Skalabilnost: aplikacija mora biti u mogućnosti podržati veći broj korisnika u slučaju rasta aplikacije i njenog proširenja kako u broju korisnika tako i u broju jela što je osigurano korištenjem lokalne baze podataka za brzi pristup podataka te povezivanjem s vanjskim API-jem za ažuriranja.
- Pouzdanost i stabilnost: treba osigurati stabilnost aplikacije prilikom korištenja i izbjeći nepotrebna rušenja prilikom uobičajene uporabe što se realizira korištenjem *try-catch* blokova za rukovanje iznimkama.
- Sigurnost: cilj je osigurati sigurno rukovanje podacima ukoliko bi se aplikacija unaprjeđivala u vidu osobnih podataka korisnika te informacije o narudžbama.
- Upotrebljivost: osigurana je jednostavnost aplikacije kako bi svim korisnicima bila jednostavna za korištenje.
- Kompatibilnost: osiguran je rad na svim modernijim *Android* uređajima što je testirano na različitim verzijama *Android* uređaja s različitim verzijama operacijskog sustava.
- Održavanje i proširivost: jednostavnost aplikacije za održavanje i mogućost za proširenje je ključ ovog zahtjeva, a to se osigurava korištenjem *MVVM* arhitekture.

Dijagram tijeka aplikacije vizualno prikazuje interakciju između korisnika i aplikacije na način da prikazuje sve korake i opcije koje se pružaju korisniku pri korištenju aplikacije. Pomaže u razaznavanju različitih *MVVM* komponenti, te njihovu međusobnu komunikaciju i odgovore na korisničke akcije. Dijagram prikazan na slici olakšava razumijevanje toka podataka i samih procesa unutar aplikacije.



Slika 4.1. Prikaz dijagrama aktivnosti

Iz prikazanog dijagrama na slici 4.1. jasno se vide svi uvjeti i opcije koje se pružaju korisniku ovisno o njegovim odabirima gdje iz početnog zaslona s prikazom svih jela korisnik ima opciju filtriranja jelovnika ili pregleda cijelog jelovnika te može odabrati određeno jelo. Ukoliko se odluči za tu akciju, otvara se zaslon s prikazom detalja jela s dodatnim informacijama i novom opcijom dodavanja jela u košaricu. Ukoliko doda jelo u košaricu ono se sprema u bazu. Korisnik nakon toga pregledava košaricu u kojoj ima opciju zaključivanja narudžbe nakon što doda jelo u košaricu te se finalizirana narudžba prikazuje u povijesti narudžbi.

4.2. Dizajn korisničkog sučelja

Pri razvoju mobilne aplikacije veliku važnost zauzima posvećenost korisničkom sučelju i korisničkom iskustvu, gdje se korisničko iskustvo odnosi na vizualni izgled te elemente aplikacije s kojima korisnik može vršiti određene akcije, dok samo korisničko iskustvo označava cjelokupni dojam korisnika tijekom interakcije s aplikacijom[13]. I jedan i drugi aspekt su od iznimne važnosti za stvaranje ne samo funkcionalne, već i intuitivne, privlačne i jednostavne aplikacije za upotrebu.

Unatoč tome što je ova aplikacija jednostavan prototip koji se koristi za prikaz realizacije *MVVM* arhitekture, osnovni principi dizajna korisničkog sučelja su i dalje primijenjeni te ne samo da osiguravaju funkcionalnost aplikacije već i korisniku omogućuju pozitivno iskustvo tijekom interakcije. Daljnji rad na aplikaciji bi svakako uključivao dublje razmatranje estetskih elemenata te dodatnih UX promjena i poboljšanja kako bi se korisničko iskustvo unaprijedilo.

Iako su aspekti poput fontova, boja i dodatnih, detaljnih animacija stavljeni u drugi plan kako bi se zadržala jednostavnost te fokus na arhitekturi i dalje su neke od osnova toga implementirane na sljedeći način.

Principi dizajna korisničkog sučelja i njihova realizacija u projektu su sljedeći:

- Konzistentnost

  Opis: korištenje istih dizajnerskih uzoraka omogućava korisnicima lakše razumijevanje i korištenje aplikacije, a konzistentnost se očituje u korištenju istih komponenti kao što su gumbi, liste i prikazi detalja.

  Realizacija: svi ekrani aplikacije su dizajnirani koristeći *Jetpack Compose* biblioteku, koja omogućuje traženu dosljednost. Svi gumbi koriste istu, osnovnu stilizaciju, a sama navigacija je dosljedna kroz cijelu aplikaciju. Korisnici u svakom trenutku znaju šta mogu očekivati kad koriste različite funkcionalnosti aplikacije.

- Intuitivnost

  Opis: bez potrebe za dodatnim uputama korisnici razumiju kako koristiti danu aplikaciju te korisnici mogu jednostavno naučiti kako se mogu izvršavati osnovni zadaci poput pregledavanja jelovnika, filtriranja, dodavanja jela u košaricu te u konačnici naručivanja i finaliziranja narudžbe.

  Realizacija: početni zaslon korisnicima pruža uvid u sva jela s jasnim informacijama o svakom jelu te korištenjem jednostavnih ikona poput ikone za dodavanje jela u košaricu, korisnicima je odmah jasno što koji gumb radi. Korištenje poznatih navigacijskih gumbova kao strelice za povratak također doprinosi intuitivnosti.

- Jednostavnost

  Opis: pružanje jasnog, jednostavnog i minimalističkog sučelja, te uklanjanje nepotrebnih složenosti upravo ukazuju na jednostavnost što je glavna karakteristika prototip aplikacija čiji je cilj pokazati funkcionalnosti bez distrakcija na dizajn.

  Realizacija: jednostavno sučelje, minimalan broj elemenata, prikaz ključnih informacija i fokus na bitnim stvarima označava jednostavnost u ovom projektu.

Pri dizajniranju glavnih zaslona također su zadržani spomenuti principi.

Početni zaslon(jelovnik):

- Prikaz svih jela u obliku kartica gdje svaka kartica prikazuje osnovne informacije o jelu.
- Mogućnost filtriranja jela prema kategorijama.

Detalji jela:

- Prikaz povećane slike jela, sastojci, cijena i kratak opis jela.
- Gumb za dodavanje jela u košaricu.
- Navigacijski gumb za povratak na početni zaslon.

Košarica:

- Prikaz svih jela koja su dodana u košaricu te uklanjanje odabranog jela.
- Prikaz cijele narudžbe.
- Mogućnost pregleda detalja svakog jela u košarici.

Povijest narudžbi:

- Prikaz popisa sa svim prethodnim narudžbama.

4.3. Arhitektura i dizajn sustava

4.3.1. Uvod u arhitekturu i dizajn sustava

Arhitektura aplikacije[14] jedan je od ključnih faktora kad se govori o dugoročnoj održivosti aplikacije, njenoj skalabilnosti i jednostavnosti održavanja. Glavni ciljevi arhitekture razvijene aplikacije za naručivanje hrane su sljedeći:

- Modularnost: jasna odvojenost i modularnost komponenti čime se različiti dijelovi aplikacije mogu neovisno razvijati i unaprjeđivati gdje promjene u prikazu podataka neće utjecati na poslovnu logiku aplikacije čime se automatski umanjuje rizik od nepredviđenih problema.
- Skalabilnost kao već navedeni nefunkcionalni zahtjev: unatoč jednostavnom prototipu, aplikacija je spremna za daljnja proširenja u budućnosti što uključuje dodavanje novih funkcionalnosti, komunikacija s vanjskim serverima ili povećanje opsega podataka s kojima aplikacija upravlja. Upravo odabirom ispravne, *MVVM* arhitekture, osigurana je lakša prilagodba te proširenje aplikacije bez potrebe za većim izmjenama postojeće strukture.
- Jednostavnost održavanja: kvalitetan odabir arhitekture uvelike olakšava održavanje koda, popravke postojećih grešaka i bugova te uvođenje nekih novih značajki. Jasnom podjelom između različitih slojeva *Model, View* i *ViewModel* doprinosti jednostavnosti koda čime se programeri lakše mogu orijentirati i navigirati kroz kod. Brže identificiranje problema što uvelike umanjuje vrijeme potrebno za održavanje aplikacije.
- Reaktivnost i odziv: Korisničko iskustvo kao ključni element uspješne aplikacije uvelike ovisi od brzine odziva na određene korisničke akcije gdje *MVVM* arhitektura, zajedno s korištenjem korutina omogućuje reaktivno ažuriranje zaslona čime se osigurava da korisnici u svakome trenutku vide najnovije podatke bez kašnjenja.

Kako bi se realizirali svi zadani ciljevi u pogledu arhitekture, odabirom *MVVM* arhitekture uspješno su razdvojene odgovornosti, održana je jednostavnost te se pruža opcija za daljnje proširenje aplikacije. Osim navedenih prednosti *MVVM* arhitekture ona također pruža i podršku za reaktivno programiranje gdje aplikacije reagiraju na promjene u stanju podataka u stvarnom vremenu tako što se koriste biblioteke kao što su *LiveData* i *StateFlow* koje omogućuju promatranje i reakciju na promjene u podacima, a upravo *MVVM* arhitektura je najprigodnija za implementaciju spomenutih biblioteka iz razloga što *ViewModel* može koristiti te alate kako bi obavijestio *View* o promjenama koje se događaju. *LiveData* [15] je komponenta iz *Android* arhitekturnih komponenti koja omogućuje promatranje podataka unutar *ViewModela*te mu je glavna prednost „svjesnost“ životnog ciklusa čime je automatski prilagođeno ponašanje aplikacije u skladu sa životnim ciklusom aktivnosti ili

fragmenta. Npr. ukoliko je korisničko sučelje neaktivno, *LiveData* prestaje emitirati podatke kako ne bi došlo do curenja memorije ili kako ne bi došlo do prekomjernog korištenja resursa. *StateFLow* je dio *Kotlin*ovih korutina te nudi sličnu funkcionalnost kao i *LiveData* s nekoliko razlika. *StateFlow* je dio *Kotlin*ove *flow* biblioteke koja se koristi za upravljanje tokova podataka te u sebi može imati samo jedno stanje, odnosno vrijednost u bilo kojem trenutku što je idealno u scenarijima kad je potrebno samo najnovija verzija podataka. Za razliku of *LiveData*-e, *StateFlow* nije svjestan životnog ciklusa te zahtjeva ručno upravljanje u kontekstu života aplikacije, ali nudi veću fleksibilnost koristeći korutine. U konkretnom primjeru aplikacije korišten je *StateFlow* u više situacija gdje je bilo potrebno ažuriranje podataka u stvarnom vremenu koje je pratio *ViewModel te obavještavao UI.*

```
17      private val _mealListUIState: MutableStateFlow<MealListUIState> =
18          MutableStateFlow(MealListUIState.Loading)
```

Slika 4.2. Prikaz korištenja StateFlowa

4.3.2. Struktura MVVM arhitekture u aplikaciji

MVVM arhitektura odabrana je za izradu ovog prototipa kako bi se osigurala jasna podjela između podataka, poslovne logike i korisničkog sučelja te kako bismo osigurali da su sve komponente sustava dobro organizirane i lako održive što je glavni cilj *MVVM* arhitekture. U ovome konkretnom prototipu *MVVM* je korišten ne samo u teorijskom smislu već i kako bi se riješili neki izazovi na koje aplikacija za naručivanje hrane može naići kao što su promjene stanja pri interakciji s korisnikom. Svaki sloj unutar *MVVM* arhitekture ima specifičnu ulogu i odgovornost:

- *Model*: predstavlja dio arhitekture koji je odgovoran za rukovanje podacima što uključuje dohvaćanje, pohranu te manipulaciju podacima. U ovoj aplikaciji model se sastoji od entiteta *Meal* i *Order* koji su klase koji upravljaju pohranom podataka. U daljnjem dijelu prikazana je klasa Meal koja je osnovni model podataka za jelo unutar aplikacije te svako od jela ima atribute kao što su ime, sastojci, cijena, slika i kategorija.

```kotlin
@Entity
@Serializable
data class Meal(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    @SerialName("Name")
    val name:String,
    @SerialName("Ingredients")
    val ingredients:String,
    @SerialName("Price")
    val price:String,
    @SerialName("Image")
    val image:String,
    @SerialName("Category")
    val category:String,
    @SerialName("Description")
    val description:String,
    var isInOrder:Boolean = false
)
```

Slika 4.3. Prikaz Meal klase

- *ViewModel*: služi kao posrednik između *Viewa* i *Modela*s ulogom držanja stanja podataka te logike poslovanja. Obavještava *View* o promjenama iako ne zna ništa o *Viewu*. U nastavku je pružen kod za jedan od *ViewModela*u aplikaciji, a radi se o *MealListViewModelu* koji upravlja stanjem popisa jela tako što dohvaća sva jela koristeći metode iz repozitorija koji je zadužen za interakciju s API-jem ili lokalnom bazom podataka. Inicijalizacijom UI stanja koje predstavlja trenutno stanje popisa jela, *View* pravilno reagira te prikazuje odgovarajuće sučelje. Nakon uspješnog dohvaćanja podataka, podaci se postavljaju u „*Success*“ stanje te se prikazuje lista jela, a ukoliko dođe do greške u dohvaćanju jela pokazuje se odgovarajuća poruka.

```kotlin
class MealListViewModel(
    private val mealListRepository: MealListRepository,
    private val orderRepository: OrderRepository
) : ViewModel() {

    private val _mealListUIState: MutableStateFlow<MealListUIState> =
        MutableStateFlow(MealListUIState.Loading)
    val mealListUIState = _mealListUIState.asStateFlow()


    fun getMealList(filter:String) {
        viewModelScope.launch {
            try {
                val data = mealListRepository.getMealList()
                if (data.isEmpty()) {
                    _mealListUIState.value = MealListUIState.EmptyMealList
                } else {
                    _mealListUIState.value = MealListUIState.Success(data.filter {
                        it.category.contains(filter)
                    })
                }
            } catch (e: Exception) {
                println(e.message)
                _mealListUIState.value = MealListUIState.Error
            }
        }
    }
}

```

Slika 4.4. Prikaz MealListViewModela

- *View*: odgovoran za prikaz podataka korisniku te za interakciju s korisnikom u vidu dohvaćanja korisničkog unosa. U ovoj aplikaciji *View* su *Composable* komponente koje prikazuju UI elemente kao što su popis jela, detalji jela, košarica. *View* osluškuje promjene stanja iz *ViewModela*te ažurira UI ovisno o njima. Prema [16], *Composable* komponente su posebne vrste funkcije koje su temelj *Jetpack Compose* biblioteke koja služi za izgradnju korisničkog sučelja. Označene anotacijom „@composable“ što označava definiranje i sastavljanje sučelja na deklarativan način što znači da umjesto pisanja imperativnog koda u smislu „promijeni tekst kad se klikne gumb“, koristeći *Composable* funkcije definira se kako tekst izgleda u kojem stanju, a *Jetpack Compose* je zadužen za ažuriranje tog teksta onda. U daljnjem prikazu koda koji prikazuje *MealDetailScreen* vidi se kako se prikazuju detalji od odabranom jelu te se omogućava korisniku da jelo doda ili ukloni iz košarice.

```kotlin
@Composable
fun MealDetailScreen(meal: Meal, navigator: DestinationsNavigator) {
    val mealDetailsViewModel = getViewModel<MealDetailsViewModel>()
    val isFavourite = rememberSaveable() {
        mutableStateOf(meal.isInOrder)
    }
    Scaffold(
        topBar = {
            TopAppBar(title = { Text(text = "${meal.name}, ${meal.price}") }, actions = {
                IconButton(
                    onClick = {
                        if (isFavourite.value) {
                            meal.isInOrder = false
                            isFavourite.value = false
                            mealDetailsViewModel.deleteMeal(meal)

                        } else {
                            meal.isInOrder = true
                            isFavourite.value = true
                            mealDetailsViewModel.insertMeal(meal)
                        }
                    }) {
                    Icon(
                        painter = painterResource(
                            id = if (isFavourite.value) {
                                R.drawable.baseline_shopping_cart_24
                            } else {
                                R.drawable.outline_shopping_cart_24
                            }
                        ),
                        contentDescription = null
                    )
                }
            },
                navigationIcon = {
                    IconButton(onClick = { navigator.navigateUp() }) {
                        Icon(imageVector = Icons.Default.ArrowBack, contentDescription = null)
                    }
                })
```

Slika 4.5. Prikaz MealDetailScreen funkcije

4.3.3. Impementacija ovisnosti i upravljanje ovisnostima

U današnjim, modernim, *Android* aplikacijama, kao što je u ovome primjeru aplikacija za naručivanje hrane, upravljanje ovisnostima(eng. *Dependency Injection*) igra ključnu ulogu u razvijanju aplikacije na modularan i skalabilan način. Korištenje *Dependency Injectiona* uz pomoć alata kao što je *Koin* pomaže u rješavanju usko povezanih klasa čime se olakšava testiranje i održavanje aplikacije. Prema

[17], *Dependency Injection* označava dizajnerski obrazac pomoću kojeg se ovisnosti između objekata jasno definiraju i lako upravljaju. U tradicionalnom pristupu, ovisnosti se često kreiraju unutar klasa što dovodi do uske povezanosti čime se otežava testiranje i proširivost aplikacije. *Koin* [18] je jednostavan, lagan i praktičan alat za ubrizgavanje ovisnosti u *Kotlin*u. Unutar aplikacije za naručivanje hrane *Koin* se koristi za upravljanje ovisnostima kao što su *ViewModel*-i, repozitoriji, DAO sučelja te mrežni servis. U daljnjem dijelu rada je kroz kod prikazana konfiguracija i korištenje *Koina*.

Moduli u *Koin*-u definiraju ovisnosti koje su aplikaciji potrebne. Svaki modul određuje kako će se određeni objekti kreirati i pružiti tamo gdje su potrebni. Unutar prototip aplikacije za naručivanje jela koristi se nekoliko modula za organiziranje različitih tipova ovisnosti.

```
7   val dbModules = module {
8       single {
9           get<MealDatabase>().mealDao()
10      }
11      single {
12          get<MealDatabase>().orderDao()
13      }
14      single {
15          Room.databaseBuilder(
16              get(),
17              MealDatabase::class.java,
18              name: "meal_database"
19          ).fallbackToDestructiveMigration()
20              .build()
21      }
22  }
```

Slika 4.6. Prikaz modula baze podataka

Ovaj modul definira na koji način se stvaraju instance baze podataka „*MealDatabase*“ te pristupni objekti podataka „*mealDao*“ i „*orderDao*“. Korištenjem metode *single* osigurava se kreiranje samo jedne instance svakog objekta tijekom životnog ciklusa aplikacije.

```kotlin
val networkModules = module {
    single {
        get<Retrofit>().create(MealService::class.java)
    }
    single {
        Json {
            ignoreUnknownKeys = true
        }
    }
    single {
        Retrofit
            .Builder()
            .baseUrl("https://seeba56.github.io")
            .addConverterFactory(get<Json>().asConverterFactory("application/json".toMediaType()))
            .build()
    }
}
```

Slika 4.7. Prikaz modula za mrežni sloj

Ovaj modul predstavlja mrežni sloj gdje se koristi Retrofit za API pozive. Definira se način stvaranja *„MealService“* instance koja komunicira s vanjskim API-jem, te JSON konfiguracija je postavljena za serijalizaciju i deserijalizaciju podataka.

```kotlin
val repositoryModule = module {
    single {
        MealListRepository(get())
    }
    single {
        OrderRepository(get(),get())
    }
}
```

Slika 4.8. Prikaz modula za repozitorije

Modul za repozitorije služi za definiranje ovisnosti za repozitorije koji su zaduženi za dohvaćanje i pohranu podataka te im se injektiraju potrebne ovisnosti iz modula gdje *„MealListRepository“* koristi mrežni servis, odnosno *„MealService“*, dok *„OrderRepository“* koristi DAO sučelja za upravljanje podacima iz baze.

```
11  val viewModelModule = module {
12      viewModel { MealListViewModel(get(),get()) }
13      viewModel { MealDetailsViewModel(get()) }
14      viewModel {OrderScreenViewModel(get())}
15      viewModel { OrderListViewModel(get()) }
16  }
```

Slika 4.9. Prikaz modula za ViewModel-e

Modul viewModelModule definira kako se kreiraju svaki od *ViewModela*. *Koin* omogućava jednostavno injektiranje ovisnosti u *ViewModel*-e što čini upravljanje stanjem i poslovnom logikom jednostavnim i efikasnim.

4.4. Implementacija baze podataka

Prema [19], *Room* baza podataka je dio *Android Jetpack* biblioteke koja pruža apstraktni sloj iznad SQLite baze podataka čime se značajno olakšava rad s lokalnim bazama podataka unutar aplikacija. Ono što je velika prednost *Room* biblioteke jest to što omogućava rad s bazom na objektno orijentiran način tako što koristi entitete odnosno objekte te koristi *DAO* sučelja za interakciju s podacima. Automatizacijom velikog broj zadataka vezanih uz rad s SQL-om aplikaciju čini jednostavnom i laganom za održavanje u vidu upravljanja podacima.

Room se sastoji od 3 ključne komponente:

- Entiteti: klase koje u bazi podataka predstavljaju tablice te je svaki entitet povezan za jednu tablicu.
- *Data Access Object*: sučelje koje sadrži metode za upravljanje podacima u bazi kao što su dohvaćanje, ažuriranje i brisanje podataka.
- Baza podataka: apstraktna klasa koja povezuje spomenute entitete i *DAO*-e te omogućuje upravljanje bazom podataka.

Konkretan primjer apstraktne klase baze podataka unutar aplikacije za naručivanje jela jest apstraktna klasa MealDatabase koja definira entitete koje baza podataka sadržava te metode za dohvaćanje DAO sučelja.

```
@TypeConverters(ListConverter::class)
@Database(entities = [Meal::class, Order::class], version = 5)
abstract class MealDatabase:RoomDatabase() {
    abstract fun mealDao(): MealDao
    abstract fun orderDao(): OrderDao
}
```

Slika 4.10. Prikaz apstraktne klase

Anotacija *@Database* specificira koje entitete, odnosno tablice, baza podataka sadržava, a u ovome konkretnom primjeru radi se o *Meal* i *Order* entitetima. Nasljeđivanjem *RoomDatabase* klasa postaje glavno mjesto za upravljanje bazom, a metode *MealDao()* i *OrderDao* služe za dobivanje instanci *MealDao* i *OrderDao* čime je omogućen pristup podacima pomoću tih sučelja. Spomenuta sučelja koriste se za definiranje metoda kojima se omogućuje interakcija u bazi podataka te umjesto ručnog pisanja SQL upita, Room automatski generira upite na temelju metoda koje su definirane u sučelju. U nastavku je dan primjer *MealDao* sučelja iz aplikacije.

```
@Dao
interface MealDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMeal(meal: Meal)

    @Delete
    suspend fun deleteMeal(meal: Meal)

    @Query("SELECT * from meal")
    fun getAllMeals():Flow<List<Meal>>

    @Query("DELETE FROM meal")
    suspend fun deleteAllMeals()

}
```

Slika 4.11. Prikaz MealDao sučelja

Anotacija *@Dao* označava da je sučelje dio *Room* baze podataka te da sadrži metode za upravljanje podacima, a metode su sljedeće:

- *insertMeal()* : metoda koja služi za umetanje jela u bazu podataka, a ako već postoji zapis s jednakim primarnim ključem on će biti zamijenjen zahvaljujući

OnConflictStrategy.REPLACE.

- *getAllMeals()* : služi za dohvaćanje svih jela iz baze podataka te ih vraća kao *Flow* čime je omogućeno reaktivno praćenje podataka.
- *deleteMeal()* : metoda koja je zadužena za brisanje određenih jela iz baze.
- *deleteAllMeals()* : metoda koja briše sva jela iz baze podataka.

Entiteti su kao što je rečeno, klase koje predstavljaju tablice unutar baze podataka te je svaki od njih mapiran u jednu tablicu, a svako polje entiteta je stupac u toj tablici. U našem primjeru imamo 2 entiteta, a u nastavku je prikazan *Meal* entitet.

```
8   @Entity
9   @Serializable
10  data class Meal(
11      @PrimaryKey(autoGenerate = true)
12      val id: Int = 0,
13      @SerialName("Name")
14      val name:String,
15      @SerialName("Ingredients")
16      val ingredients:String,
17      @SerialName("Price")
18      val price:String,
19      @SerialName("Image")
20      val image:String,
21      @SerialName("Category")
22      val category:String,
23      @SerialName("Description")
24      val description:String,
25      var isInOrder:Boolean = false
26  )
```

Slika 4.12. Prikaz Meal entiteta

Anotacija *@Entity* označava da je klasa *Meal* entitet, odnosno kao što je spomenuto, tablica u bazi. Spomenuta polja u bazi su u ovome slučaju *name* u koji se pohranjuje ime jela, *ingredients* u koji se pohranjuju sastojci, *price* u kojeg se pohranjuje cijena, *image* u kojeg se pohranjuje URL slike jela, *category* u kojeg se pohranjuje kategorija jela za olakšano filtriranje te *isInOrder* kao *Boolean* tip podatka koji služi za praćenje je li jelo dodano u narudžbu. Primarni ključ, *@PrimaryKey* je postavljen kao primarni ključ s opcijom automatskog generiranja vrijednosti ID-a za svaki novi unos u bazu. Korištenjem *Room* baze podataka omogućen je skalabilan i modularan pristup za rad s podacima čime se zadržala jednostavnost održavanja i proširenja aplikacije.

4.5. Upravljanje podacima i integracija API-ja

4.5.1. Uvod u upravljanje podacima i integraciju API-ja

U današnjim mobilnim aplikacijama jedno od najvažnijih aspekata jest upravljanje podacima te integracija s vanjskim izvorima podataka. Često postoje zahtjevi za podacima koji dolaze iz različitih, nerijetko vanjskih API-ja ili lokalnih servisa, a sve s ciljem pružanja ažurnih i dinamičkih informacija korisniku. U konkretnom primjeru aplikacije za naručivanje hrane korišten je vanjski API koji dohvaća dostupne informacije o jelima. Nakon što se informacije dohvate one se prikazuju korisniku unutar aplikacije u vidu jelovnika te se dodaju u košaricu gdje se finalizira narudžba. API omogućuje aplikaciji komunikaciju s vanjskim serverima kao što su baze podataka, web serveri i ostali. U primjeru aplikacije za naručivanje hrane radi se o dinamičkim podacima koji se povlače s vanjskog izvora te se prikazuju korisniku, a zahvaljujući API-ju, informacije prikazane u aplikaciji ostaju ažurirane s najnovijim informacijama, bez potrebe da lokalnom pohranom svih podataka na uređaju korisnika. Glavna prednost korištenja API-ja je ta što je omogućeno centralizirano upravljanje podacima u vidu ažuriranja podataka na serveru čime se automatski osvježavaju i podaci prikazani u aplikaciji bez potrebe za promjenom koda unutar aplikacije što zna biti od presudne važnosti u određenim aplikacijama kako bi se promjene prikazivale u stvarnom vremenu. Osim same integracije API-ja, s podacima je potrebno upravljati na odgovarajući način, a kako bi korisnici podacima mogli pristupati i izvan mrežnom načinu, korištena je Room baza podataka kako bi se podaci spremili lokalno. Pristup poznat kao cache-first omogućuje bolje korisničko iskustvo jer aplikacija ne ovisi u potpunosti o mreži. Koristeći Room bazu podataka, korisnicima je omogućena interakcija i pregled podataka bez obzira na kvalitetu mreže gdje se podaci povlače iz API-ja, pohranjuju se lokalno te se prikazuju korisniku, a sav proces dohvaćanja i spremanja podataka se odvija u pozadini. Tehnologije koje su korištene za upravljanje podacima i API integraciju su sljedeći:

- Retrofit: biblioteka korištena za upravljanje mrežnom komunikacijom čime se omogućuje jednostavno dohvaćanje podataka iz API-ja te njihova daljnja obrada.
- *Kotlin*x.serialization: biblioteka kojom je omogućena serijalizacija i deserijalizacija podataka čime se JSON odgovori iz API-ja serijaliziraju u *Kotlin* objekte koje aplikacija koristi.
- Room: lokalna baza podataka za skladištenje podataka koji dolaze iz API-ja. Osigurava rad aplikacije u izvan mrežnom načinu.

4.5.2. Definiranje API-ja i struktura podataka

Kao što je u uvodu objašnjeno, API se koristi za komunikaciju s vanjskim sustavima na standardiziran način, a u aplikaciji za naručivanje jela se dohvaćaju podaci, dodatni detalji o jelima koja su dostupna korisnicima. Korištenjem *HTTP GET* zahtjeva, dohvaćaju se podaci u JSON formatu. Aplikacija podatke dohvaća koristeći *Retrofit* koji je biblioteka za mrežnu komunikaciju, a dohvaća ih s URL-a koji je prikazan u kodu

```
6   interface MealService {
7       @GET("/data/data.json")
8       suspend fun getMealList():List<Meal>
9   }
```

Slika 4.13. Prikaz MealService sučelja

Sučelje definira metodu *getMealList* koja vraća listu objekata tipa Meal te koristi *HTTP GET* metodu za dohvaćanje podataka s API-ja. Nakon uspješnog odgovora API-ja, odgovor se serijalizira u *Kotlin* objekte.


```
{
  "Name": "Monster burger",
  "Ingredients": "triple beef patty, cheddar, bacon, lettuce, onion, aurora dressing",
  "Price": "10€",
  "Image": "https://img.freepik.com/premium-psd/qottab-isolated-transparent-background_191095-31767.jpg?w=740",
  "Category": "Burger",
  "Description": "A massive burger with triple beef patties, cheddar cheese, bacon, lettuce, and aurora dressing."
},
```


Slika 4.14. Prikaz JSON odgovora

API vraća podatke o jelima u JSON formatu prikazanom na slici 4.13. gdje svako jelo se sastoji od različitih atributa kao što su naziv, cijena, slika, kategorija i opis. Svaki zapis ovog odgovora se mapira u klasu *Meal* koja je prikazana na slici 4.3.. Pri pokretanju aplikacije podaci se dohvaćaju koristeći metodu *getMealList* koja je definirana u *MealService* sučelju. Nakon toga se podaci obrađuju koristeći već prikazan *MealListRepository* koji ih pohranjuje ili prosljeđuje *ViewModelu*.

4.5.3. Integracija s Retrofitom

Retrofit je popularna biblioteka korištena u svijetu *Android* razvoja kojom se pojednostavljuje rad s RESTful web servisima. Prema [20], Retrofit služi za upravljanje HTTP zahtjevima te pretvaranje podataka koje aplikacija prima u aplikaciji korisne, *Kotlin* objekte. Retrofit je korišten kako bi se dohvatili podaci s vanjskog API-ja. Neke od glavnih koristi korištenja Retrofita su sljedeće:

- Jednostavna konfiguracija: kroz jednostavno sučelje omogućeno je lako definiranje API poziva.
- Serijalizacija podataka: podržavajući različite formate poput JSON-a mora omogućiti i njihovu serijalizaciju i deserijalizaciju koristeći serijalizacijske biblioteke kako bi se podaci pretvorili u *Kotlin* objekte.
- Asinkroni pozivi: mogućnost obavljanja HTTP zahtjeva u pozadini bez korištenja glavne niti uvelike poboljšava korisničko iskustvo.

Na slici 4.7. prikazano je postavljanje Retrofita unutar aplikacije. Konfiguriran je putem modula u kojem su definirani svi mrežni slojevi, uključujući URL za API te način serijalizacije podataka za koje se koristi *Kotlin*x.serialization za rad s JSON podacima.Korištenjem *MealService* sučelja omogućeno je dohvaćanje podataka koristeći *Kotlin* korutine kako bi se zahtjev obavljao u pozadini bez blokiranja glavne niti. Podaci koje aplikacija dohvaća putem Retrofita prosljeđuje kroz repozitorij koji je zadužen za mrežnu komunikaciju te osigurava ispravno rukovanje podacima. Repozitorij služi kao posrednik između mrežnog sloja i *ViewModela*.

```
7   class MealListRepository(private val mealService: MealService) {
8
9       suspend fun getMealList():List<Meal>{
10          return mealService.getMealList()
11      }
12
13  }
```

Slika 4.15. Prikaz MealListRepository repozitorija

Kad korisnik otvori aplikaciju ili odabere da se podaci osvježe, *ViewModel* koristi repozitorij kako bi započeo mrežni zahtjev putem Retrofita.

```
22      fun getMealList(filter:String) {
23          viewModelScope.launch {
24              try {
25                  val data = mealListRepository.getMealList()
26                  if (data.isEmpty()) {
27                      _mealListUIState.value = MealListUIState.EmptyMealList
28                  } else {
29                      _mealListUIState.value = MealListUIState.Success(data.filter {
30                          it.category.contains(filter)
31                      })
32                  }
33              } catch (e: Exception) {
34                  println(e.message)
35                  _mealListUIState.value = MealListUIState.Error
36              }
37          }
38      }
39  }
```

Slika 4.16. Prikaz getMealList funkcije

Korištenje *viewModelScopea* omogućeno je obavljanje mrežnog zahtjeva u pozadini te ukoliko je zahtjev uspješan, podaci dobiveni iz API-ja se prosljeđuju korisničkom sučelju, a ukoliko dođe do pogreške, *ViewModel* rukuje s pogreškom te obavještava korisnika o problemu.

5. RAZVOJ I FUNKCIONALNOSTI APLIKACIJE

Ovo poglavlje se bavi razvojem aplikacije i opisom glavnih funkcionalnosti. Predstavljeno je korisničko sučelje te osnovne mogućnosti aplikacije poput odabira jela, dodavanja jela u košaricu, upravljanja narudžbama te finalizacija narudžbi. Posebna pažnja je posvećena povezivanju funkcionalnosti s MVVM arhitekturom te načinom na koji aplikacija upravlja podacima u stvarnom vremenu.

5.1. Izgradnja korisničkog sučelja

Korisničko sučelje (eng. UI), jedan je od ključnih elemenata bilo koje aplikacije, kako web tako i mobilne s obzirom da korisnici stupaju u interakciju s aplikacijom putem njega. Iako je u prethodnom, 4. poglavlju spomenut tehnički aspekt dizajna korisničkog sučelja, u ovome dijelu fokus je na nekoliko ključnih principa i teorija koje stoje iza uspješnog UI dizajna te je prikazano na koji način je ono realizirano unutar aplikacije. Glavni princip koji je korišten unutar aplikacije bio je minimalistički dizajn s obzirom da je aplikacija prototip te služi za prikaz *MVVM* arhitekture, bilo je ključno osigurati jednostavno, ali opet, funkcionalno sučelje za intuitivno korištenje. Korištenjem *Jetpack Compose* tehnologije, postignut je deklarativni pristup izradi sučelja gdje se izgled i funkcionalnost aplikacija definiraju na temelju stanja aplikacije čime su omogućene brže i lakše prilagodbe bez velikih promjena u kodu. Neki od aspekata koji su uzeti u obzir prilikom izrade aplikacije su interaktivnost i prilagodljivost korisničkog sučelja tako što je omogućeno jednostavno pregledavanje i filtriranje jelovnika, dodavanje jela u košaricu, pregled košarice te finaliziranje narudžbe, sve putem nekoliko intuitivnih zaslona. Svaka promjena unutar sučelja odmah je reflektirana korištenjem *StateFlowa* i *ViewModela* čime je osigurano automatskog ažuriranje sučelja ovisno o promjenama podataka. Također pri razvoju sučelja korištene su *Composable* funkcije čijim se korištenjem omogućuje ponovno korištenje elemenata sučelja na više različitih mjesta unutar aplikacije čime se smanjuje dupliciraje koda, a samim time se povećava i održivost aplikacije. Na primjer, funkcije za prikaz stavki iz jelovnika ili detalja o narudžbi mogu se integrirati na različitim zaslonima bez pisanja već napisane logike pri prvoj implementaciji. Bitno je napomenuti kako je korisničko sučelje samo dio cjelokupnog korisničkog iskustva. Iako je fokus stavljen na jednostavnost i funkcionalnost postoje i drugi aspekti korisničkog iskustva kao što su brzina, responzivnost i prilagodljivost aplikacije na različitim uređajima. Iako razvijena kao prototip, aplikacija slijedi osnovne smjernice za responzivan dizajn čime je omogućeno korištenje aplikacije na različitim ekranima i uređajima.

U izgradnji sučelja primijenjen je deklarativni pristup kroz *Jetpack Compose* čime je omogućena fleksibilna i dinamična gradnja vizualnih elemenata. Glavna prednost takvog pristupa jest što je

omogućeno automatsko reagiranje na promjene bez potrebe za ručnim osvježavanjem elemenata. Korištenjem *StateFlow*-a i *MutableState*-a unutar objekata *Composable* funkcija čime se osigurava responzivnost na korisničke interakcije.

Jedan od glavnih ciljeva sučelja bilo je osigurati modularnost kako bi se isti elementi mogli koristiti na više mjesta unutar aplikacije što je ostvareno korištenjem *Composable* funckija čime se definiraju mali, izolirani dijelovi sučelja koji se na koncu sastavljaju u cjelinu. Primjer modularne funkcije dan je u nastavku, a prikazana funkcija koristi se za prikaz stavke iz jelovnika, a omogućuje prikaz slike, naziva i cijene jela.

```
@Composable
fun MealListItem(
    meal: Meal,
    onMealItemClick: (Meal) -> Unit
) {
    Column {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .background(MaterialTheme.colorScheme.background)
                .padding(vertical = 8.dp)
        ) {
            AsyncImage(
                model = meal.image,
                contentDescription = null,
                modifier = Modifier
                    .size(64.dp)
                    .clickable { onMealItemClick.invoke(meal) },

                )
            Column(
                modifier = Modifier
                    .padding(horizontal = 8.dp)
                    .clickable { onMealItemClick.invoke(meal) },
                verticalArrangement = Arrangement.SpaceBetween,

                ) {
                Text(text = meal.name, style = MaterialTheme.typography.labelLarge)
                Text(text = meal.price, style = MaterialTheme.typography.labelSmall)
            }
        }
        Divider(modifier = Modifier.fillMaxWidth(), thickness = 2.dp)
    }
```

Slika 5.1. Prikaz MealListItem funkcije

Prikazana funkcija koristi se na više mjesta u aplikaciji, uključujući zaslon za pregled jelovnika i narudžbi čime se olakšava održavanje i proširivanje sučelja tako što se svi aspekti sučelja povezani s prikazom jela mogu ažurirati na jednom mjestu.

Kako bi se ostvarilo spomenuto automatsko ažuriranje podataka na korisničkom sučelju korišten je *StateFlow* u svakom zaslonu koji prikazuje podatke iz baze ili API-ja kako bi se pri dodavanju jela u košaricu ili ažuriranju jela na popisu narudžbi omogućila ažurnost informacija u stvarnom vremenu. U prethodnom poglavlju na slici 4.4. je kod koji prikazuje kako se stanjem upravlja u *ViewModelu* za prikaz popisa jela. U prikazanom primjeru vidljivo je kako se podaci dohvaćaju iz repozitorija unutar *ViewModela*te se prosljeđuju na korisničko sučelje. Promjenom stanja *mealListUIState*,

Jetpack Compose automatsku ažurira sučelje prikazujući odgovarajuće elemente.

Na posljetku, ostalo je još prikazati na koji način je korisniku omogućena interakcija i komunikacija s aplikacijom kroz različite interaktivne elemente kao što su gumbi, liste i detalji jela. Kako bi korisnici vidjeli detalje jela potrebno je kliknuti na jelo kako bi vidjeli detalje vezano za jelo ili kako bi dodali jelo u košaricu potrebno je pritisnuti odgovarajući gumb. U nastavku je prikazan kod za dodavanje i uklanjanje jela iz košarice unutar *MealDetailScreen* funkcije.

```
39  fun MealDetailScreen(meal: Meal, navigator: DestinationsNavigator) {
40      val mealDetailsViewModel = getViewModel<MealDetailsViewModel>()
41      val isFavourite = rememberSaveable() {
42          mutableStateOf(meal.isInOrder)
43      }
44      Scaffold(
45          topBar = {
46              TopAppBar(title = { Text(text = "${meal.name}, ${meal.price}") }, actions = {
47                  IconButton(
48                      onClick = {
49                          if (isFavourite.value) {
50                              meal.isInOrder = false
51                              isFavourite.value = false
52                              mealDetailsViewModel.deleteMeal(meal)
53
54                          } else {
55                              meal.isInOrder = true
56                              isFavourite.value = true
57                              mealDetailsViewModel.insertMeal(meal)
58                          }
59                      }) {
60                      Icon(
61                          painter = painterResource(
62                              id = if (isFavourite.value) {
63                                  R.drawable.baseline_shopping_cart_24
64                              } else {
65                                  R.drawable.outline_shopping_cart_24
66                              }
67                          ),
68                          contentDescription = null
69                      )
70                  }
71              },
```

Slika 5.3. Prikaz MealDetailScreen funkcije

5.2. Integracija s backendom

U ovome dijelu objašnjena je komunikacija aplikacije s backend serverom koristeći RESTful API te koristeći *MVVM* arhitekturu za upravljanje podacima i asinkronim zahtjevima. Kao što su već objašnjene neke od stvari koje su navedene u ovom poglavlju, *Retrofit* omogućuje komunikaciju s API-jem, a *ViewModel*-i osiguravaju pravilno upravljanje podacima koji su dohvaćeni sa servera te proslijeđeni na korisničko sučelje.

Komunikacija između aplikacije i servera se odvija putem HTTP zahtjeva pri čemu se šalje GET zahtjev na RESTful API kako bi se dohvatili podaci u JSON obliku. Upravo zbog Retrofita je moguće dohvaćati odgovore u spomenutom, JSON obliku s obzirom na podršku za serijalizaciju i deserijalizaciju podataka gdje se oni pretvaraju u *Kotlin* objekte. U aplikaciji je korišten već prikazan *MealService* koji šalje GET zahtjev.
Kako bi se dohvaćenim podacima moglo pravilno upravljati, ključnim elementom *MVVM* arhitekture, *ViewModel*-om se njima i upravlja. Korišten je s ciljem pozivanja API-ja, dohvaćanja popisa jela te rukovanjem stanja aplikacije. Svi navedeni zadaci se obavljaju asinkrono kako ne bi došlo do blokiranja korisničkog sučelja.
Unutar *MVVM* arhitekture, repozitorij je iskorišten kao posrednik između *ViewModela*te vanjskih izvora poput API-ja. Na primjer, za obradu podataka koje aplikacija dohvaća s API-ja, korišten je već naveden *MealListRepository* koji obrađuje podatke prije prosljeđivanja *ViewModelu*. Kako bi se osiguralo reaktivno ažuriranje sučelja korištenjem *Jetpack Compose*-a, novi podaci dohvaćeni s API-ja se automatski ažuriraju putem *StateFlow*-a.

```
70          when (uiState.value) {
71              is MealListUIState.Success -> {
72                  MealListScreenContent(
73                      mealList = (uiState.value as MealListUIState.Success).list,
74                      onMealItemClick = {meal ->
75                          navigator.navigate(MealDetailScreenDestination(meal))},
76                      selectedOptionValue = selectedOption.value,
77                      onIconRadioButtonClick = { selectedOption.value = it },
78                      paddingValues = it
79                  )
80              }
81
82              MealListUIState.Loading -> {
83                  EmptyLisScreen()
84              }
85
86              MealListUIState.EmptyMealList -> {
87                  EmptyLisScreen()
88
89              }
90
91              MealListUIState.Error -> {
92                  ErrorScreen {
93                      mealListViewModel.getMealList(selectedOption.value)
94                  }
95
96              }
97          }
98      }
```

Slika 5.4. Prikaz funkcionalnosti za praćenje promjene stanja

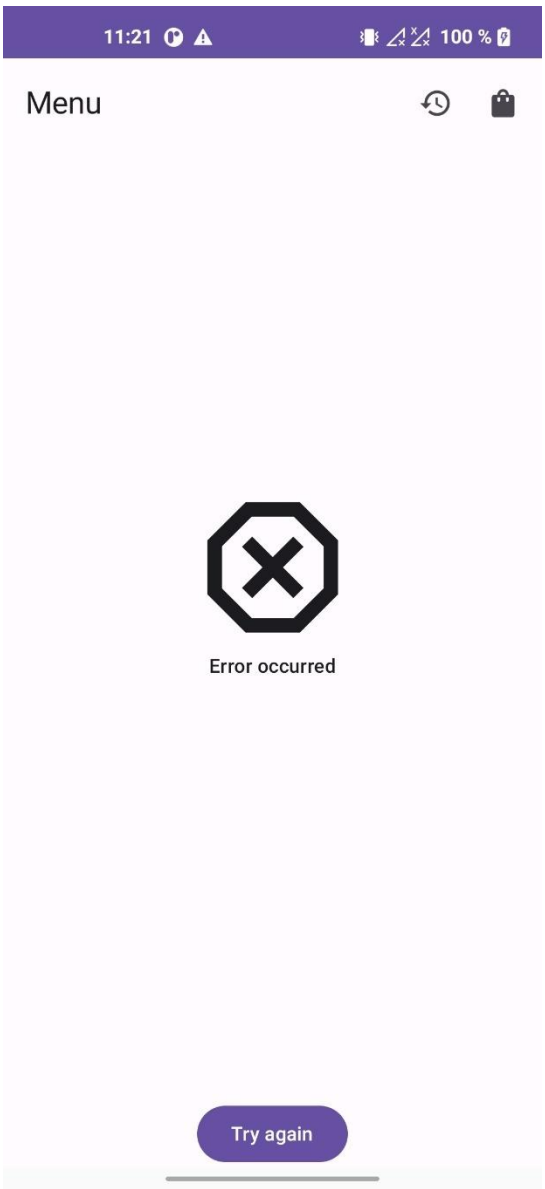
Prikazani dio koda je zadužen za praćenje promjene stanja u aplikaciji, gdje se ovisno o stanju, korisniku prikazuje ispravan prikaz, odnosno ukoliko dohvaćanje podataka uspije korisniku se prikazuje popis jela, a ukoliko dođe do određene greške prilikom dohvaćanja, korisnika se o tome obavještava odgovarajućom porukom.

5.3. Pregled jelovnika

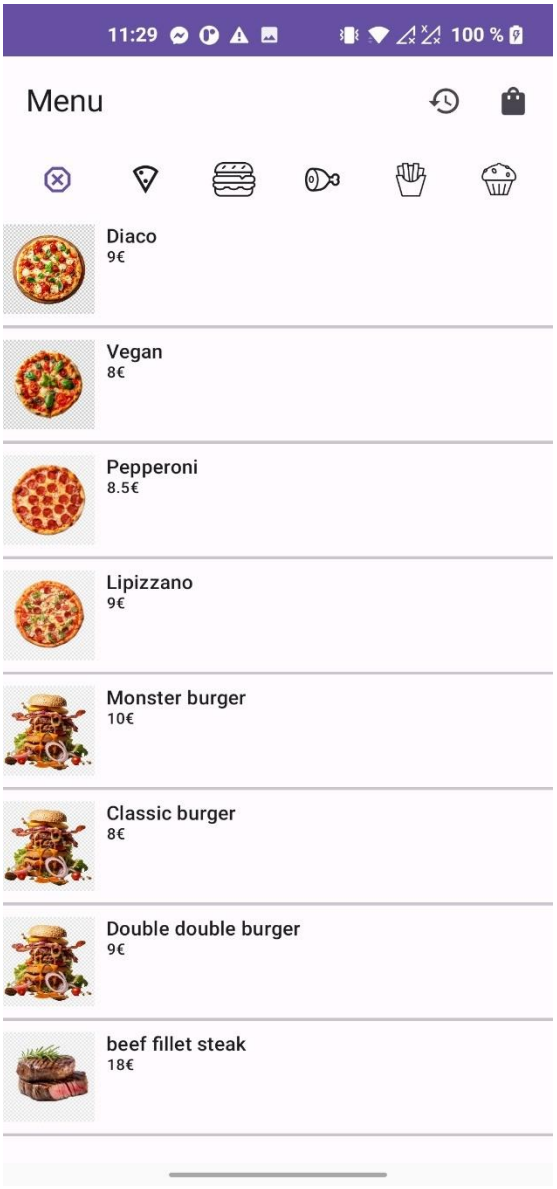
Pregled jelovnika je ključna funkcionalnost aplikacije kojom je korisnicima omogućen pregled dostupnih jela iz ponude. Funkcionalnost je implementirana na način da korisnicima nudi intuitivan i brz pristup informacijama o jelima uz mogućnost filtriranja jela prema različitim kategorijama.

5.3.1. Glavni zaslon pregleda jelovnika

Glavni zaslon aplikacije prikazuje popis svih jela dohvaćenih iz baze podataka putem API-ja. Korisniku su prikazane osnovne informacije o jelu kao što su ime, cijena i slika tog jela. U nastavku je prikazan kod za implementaciju glavnog zaslona kao i slika zaslona jelovnika u 2 slučaja, odnosno u slučaju uspješnog dohvaćanja jela, te u slučaju kad nema mreže te se jela ne uspiju dohvatiti.



Slika 5.5. Prikaz glavnog zaslona bez mreže



Slika 5.6. Prikaz glavnog zaslona s dohvaćenim jelima

Na prikazanim slikama 5.5. i 5.6. su prikazana oba slučaja kod dohvaćanja podataka, odnosno uspješno i neuspješno dohvaćanje, gdje je razlog neuspješnog dohvaćanja nepostojanje mreže gdje korisnik ima opciju ponovnog učitavanja jelovnika ukoliko se spoji na mrežu. Nakon uspješnog dohvaćanja podataka oni se prikazuju korisniku kao što se može vidjeti na slici 5.5. gdje je korisniku vidljiv popis jela s osnovnim informacijama, top bar za navigiranje između zaslona i traka koja služi za filtriranje jela.

```
48      Scaffold(
49          topBar = {
50              TopAppBar(
51                  title = { Text(text = "Menu") },
52                  actions = {
53                      IconButton(onClick = {navigator.navigate(OrderListScreenDestination)}) {
54                          Icon(
55                              painter = painterResource(id = R.drawable.baseline_history_24),
56                              contentDescription = null
57                          )
58
59                      }
60                      IconButton(onClick = {navigator.navigate(OrderScreenDestination)}) {
61                          Icon(
62                              painter = painterResource(id = R.drawable.baseline_shopping_bag_24),
63                              contentDescription = null
64                          )
65
66                      }
67                  })
68          }
```

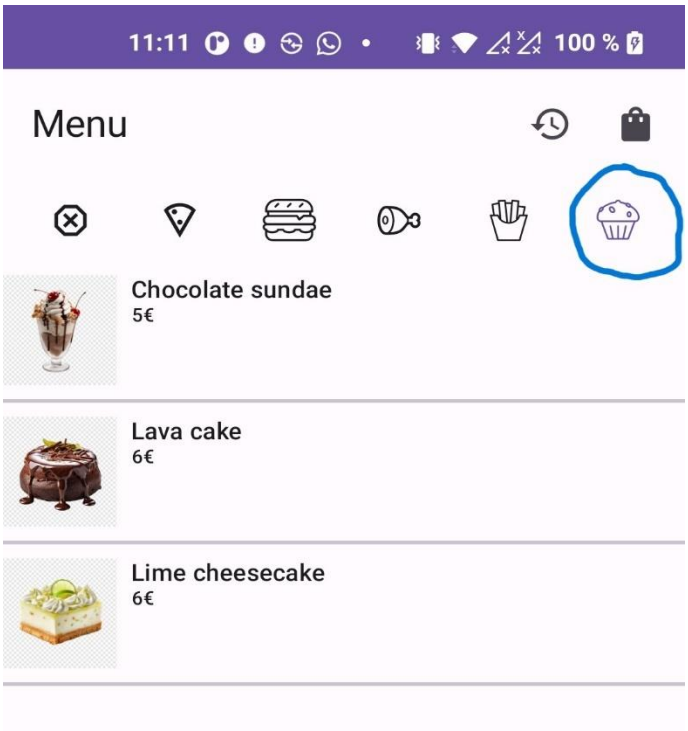
Slika 5.7. Prikaz koda za navigaciju

Na slici 5.7. prikazan je kod koji prikazuje top bar kojim je omogućena navigacija između zaslona kao što su povijest narudžbi te pregled košarice. Odmah ispod toga na zaslonu se nalaze gumbovi koji se koriste kako bi se filtrirala jela ovisno o kategoriji, a kod i slika zaslona filtriranih jela su prikazani u nastavku.

```
120     stickyHeader {
121         Row(
122             modifier = Modifier
123                 .fillMaxWidth()
124                 .background(MaterialTheme.colorScheme.background),
125             horizontalArrangement = Arrangement.SpaceEvenly
126         ) {
127             IconRadioButton(
128                 isSelected = selectedOptionValue == "",
129                 onClick = onIconRadioButtonClick,
130                 value = "",
131                 resDrawable = R.drawable.outline_dangerous_24
132             )
133             IconRadioButton(
134                 isSelected = selectedOptionValue == "Pizza",
135                 onClick = onIconRadioButtonClick,
136                 value = "Pizza",
137                 resDrawable = R.drawable.outline_local_pizza_24
138             )
139             IconRadioButton(
140                 isSelected = selectedOptionValue == "Burger",
141                 onClick = onIconRadioButtonClick,
142                 value = "Burger",
143                 resDrawable = R.drawable.fast_food_burger
```

Slika 5.8. Prikaz dijela koda filtriranje

Na slici 5.8. prikazan je dio koda koji se koristi za filtriranje jela tako da ovisno o korisnikovoj akciji, odnosno pritisku gumba koji označava pojedinu kategoriju jela se prikazuje odabrana kategorija jela.

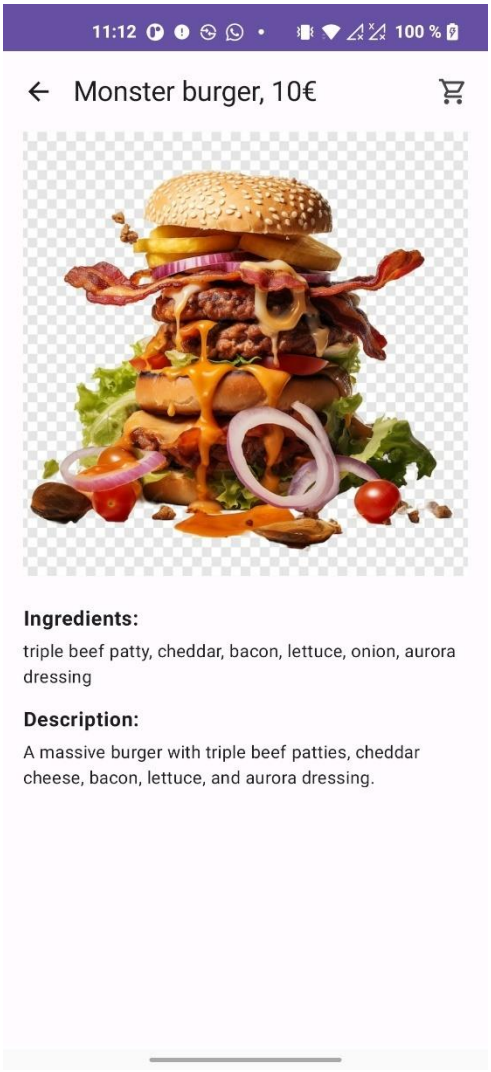


Slika 5.9. Prikaz filtriranog zaslona s odabranom kategorijom „Deserti“

Na slici 5.9. jasno se vidi kako je odabrana kategorija deserti gdje je ikona za deserte obojena drugom bojom u odnosu na ostale kategorije te su prikazana jela filtrirana tako da samo ona jela koja su kategorije „Dessert“ prikazana na popisu jela čime je korisniku olakšan pronalazak željenog jela.

5.3.2. Zaslon s detaljima

Pored osnovnog zaslona, korisnici mogu kliknuti na bilo koje jelo kako bi vidjeli detaljnije informacije o jelu, a prikazane su informacije poput naziva, cijene, slike, sastojaka te kratkog opisa svakog pojedinog jela. Glavna funkcionalnost zaslona s detaljima jest mogućnost dodavanja jela u košaricu te uklanjanje istog iz košarice. U nastavku je prikazan zaslon s detaljima te kod za taj zaslon. Kod je podijeljen u 2 cjeline gdje prvi dio prikazuje kod implementacije zaslona, odnosno UI komponentu, a drugi dio prikazuje funkcionalnost dodavanja i uklanjanja jela u i iz košarice.



Slika 5.10. Prikaz zaslona gdje jelo nije u košarici

Slika 5.11. Prikaz zaslona gdje jelo je u košarici

Na slikama 5.10. i 5.11. prikazana su 2 identična zaslona s glavnom razlikom što na slici 5.7. jelo „*Monster burger*“ nije dodano u košaricu, dok na slici 5.8. se jasno vidi pogledom na ikonu kolica da je jelo u ovome slučaju dodano u košaricu. Ukoliko korisnik želi dodati više istih jela u košaricu potrebno je vratiti se na prethodni zaslon te ponovno ući u detalje željenog jela i dodati ga jednakim postupkom u košaricu.

```
131  fun MealDetailScreen(meal: Meal, navigator: DestinationsNavigator) {
132      Scaffold(
133          topBar = {
134              TopAppBar(title = { Text(text = "${meal.name}, ${meal.price}") }, actions = {
135
136              },
137                  navigationIcon = {
138                      IconButton(onClick = { navigator.navigateUp() }) {
139                          Icon(imageVector = Icons.Default.ArrowBack, contentDescription = null)
140                      }
141                  })
142          }
143      ) {
144          MealDetailScreenContent(meal = meal, paddingValues = it)
145
146      }
147
```

Slika 5.12. Izdvojeni dio koda za implementaciju sučelja

Na slici 5.12. prikazan je dio koda koji se koristi za postavljanje top bara gdje su korisniku prikazani naziv jela i cijena te također i strelica za povratak natrag. Izdvojen je dio gdje je dodana košarica za dodavanje jela u košaricu koji će biti prikazan u nastavku. U nastavku je prikazan drugi dio koda koji se povezan s korisničkim sučeljem gdje se prikazuju sve informacije o jelu, a to su slika, sastojci i opis, dok su preostali detalji kako je već navedeno prikazani u top baru.

```
@Composable
private fun MealDetailScreenContent(meal: Meal, paddingValues: PaddingValues) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(
                top = paddingValues.calculateTopPadding(),
                start = 16.dp,
                end = 16.dp,
                bottom = 16.dp
            ),
    ) {
        AsyncImage(
            model = meal.image,
            contentDescription = null,
            modifier = Modifier
                .fillMaxWidth()
                .background(MaterialTheme.colorScheme.background)
                .padding(bottom = 24.dp)
        )
        Text(
            text = "Ingredients:",
            style = MaterialTheme.typography.bodyLarge,
            modifier = Modifier.padding(bottom = 8.dp),
            fontWeight = FontWeight.Bold
        )
        Text(
            text = meal.ingredients,
            style = MaterialTheme.typography.bodyMedium,
            modifier = Modifier.padding(bottom = 16.dp)
        )
        Text(
            text = "Description:",
            style = MaterialTheme.typography.bodyLarge,
            modifier = Modifier.padding(bottom = 8.dp),
            fontWeight = FontWeight.Bold
        )
```

Slika 5.13. Izdvojeni dio koda za implementaciju sučelja s detaljima jela

U nastavku je prikazan dio koda koji je zadužen za logiku upravljanja dodavanja i uklanjanja jela iz košarice. Funkcionalnost je realizirana pomoću *IconButton* gumba koji, ovisno o trenutnom stanju „*isInOrder*“ omogućava dodavanje ili uklanjanje jela iz košarice. *MealDetailsViewModel* se koristi za pozive funkcija *insertMeal()* i *deleteMeal()* koje ažuriraju odgovarajuće podatke.

```
fun MealDetailScreen(meal: Meal, navigator: DestinationsNavigator) {
    val mealDetailsViewModel = getViewModel<MealDetailsViewModel>()
    val isFavourite = rememberSaveable() {
        mutableStateOf(meal.isInOrder)
    }
    Scaffold(
        topBar = {
            TopAppBar(title = { Text(text = "${meal.name}, ${meal.price}") }, actions = {
                IconButton(
                    onClick = {
                        if (isFavourite.value) {
                            meal.isInOrder = false
                            isFavourite.value = false
                            mealDetailsViewModel.deleteMeal(meal)

                        } else {
                            meal.isInOrder = true
                            isFavourite.value = true
                            mealDetailsViewModel.insertMeal(meal)
                        }
                    }) {
                    Icon(
                        painter = painterResource(
                            id = if (isFavourite.value) {
                                R.drawable.baseline_shopping_cart_24
                            } else {
                                R.drawable.outline_shopping_cart_24
                            }
                        ),
                        contentDescription = null
                    )
                }
            },
        }
    ) {
        MealDetailScreenContent(meal = meal, paddingValues = it)

    }

```

Slika 5.14. Izdvojeni dio koda za implementaciju funkcionalnosti dodavanja i uklanjanja jela

5.4. Proces naručivanja

Nakon pritiska gumba za dodavanje jela u košaricu čija je implementacija prikazana u cjelini 5.3., sustav prvo poziva odgovarajuću funkciju unutar *ViewModela*, a u ovome slučaju je to *MealDetailsViewModel*, jelo se dodaje u bazu podataka te je u nastavku prikazano na koji način se pohranjuju u lokalnu bazu te prikazuju u košarici. U nastavku je dan primjer koda za dodavanje jela u košaricu.

```
9   class MealDetailsViewModel(private val orderRepository: OrderRepository):ViewModel() {
10      fun insertMeal(meal: Meal){
11          viewModelScope.launch {
12              orderRepository.insertMeal(meal)
13          }
14      }
15
16      fun deleteMeal(meal: Meal){
17          viewModelScope.launch {
18              orderRepository.deleteMeal(meal)
19          }
20      }
21  }
```

Slika 5.15. Izdvojeni dio koda za implementaciju funkcionalnosti dodavanja i uklanjanja jela

Unutar prikazane klase može se vidjeti da se funkcija *insertMeal()* poziva kako bi se jelo dodalo u bazu koristeći *orderRepository* koji upravlja lokalnim podacima u Room bazi podataka na sljedeći način.

```
class OrderRepository(private val mealDao: MealDao, private val orderDao: OrderDao) {

    fun getAllMeals():Flow<List<Meal>>{
        return mealDao.getAllMeals()
    }

    suspend fun deleteMeal(meal: Meal){
        mealDao.deleteMeal(meal)
    }

    suspend fun insertMeal(meal: Meal){
        mealDao.insertMeal(meal)
    }

    suspend fun insertOrder(order: Order){
        orderDao.insertOrder(order)
    }

    fun getAllOrders():Flow<List<Order>>{
        return orderDao.getAllOrders()
    }

    suspend fun deleteAllMeals(){
        mealDao.deleteAllMeals()
    }
}
```

Slika 5.16. Prikaz OrderRepository-ja

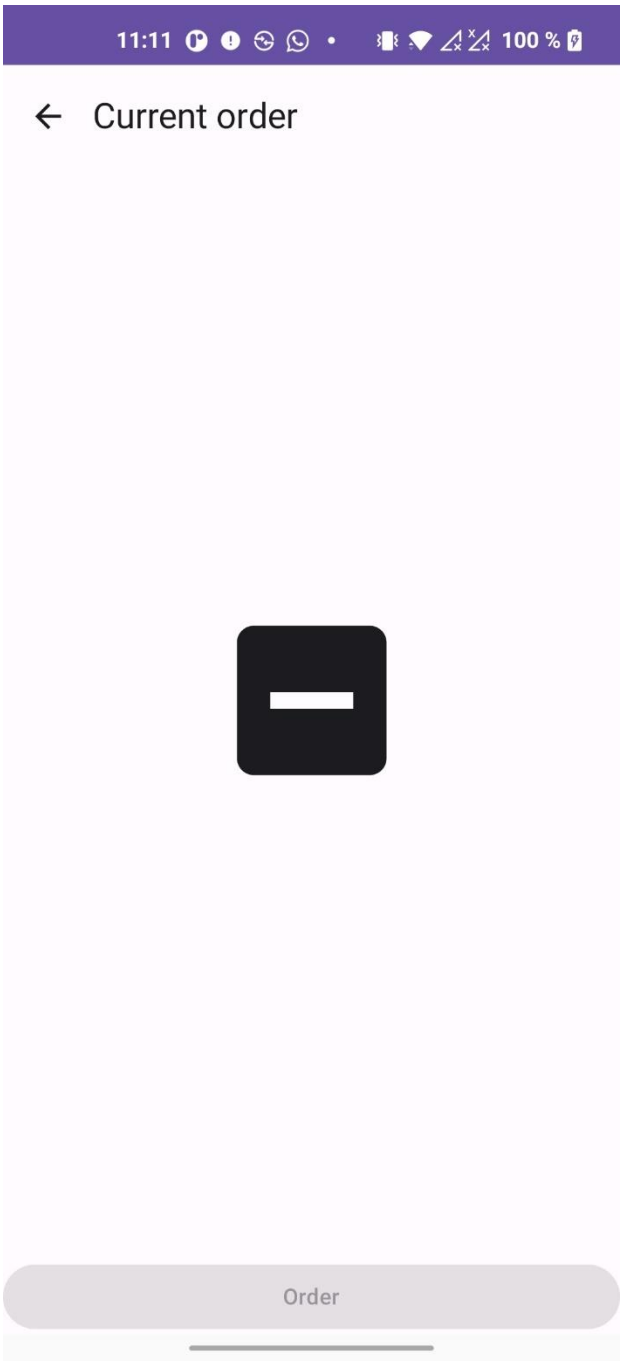
Na slici 5.16. prikazan je *OrderRepository* koji se unutar aplikacije koristi za upravljanje podacima o narudžbama i jelima u aplikaciji. Komunicira s bazom podataka preko *MealDao* i *OrderDao* sučelja. Kad korisnik doda jelo u košaricu, funkcija *insertMeal()* iz *MealDao* sučelja pohranjuje to jelo u bazu, a kad korisnik potvrdi narudžbu, funkcija *insertOrder()* iz *OrderDao* pohranjuje cijelu narudžbu u vidu liste jela s vremenskom oznakom. Nakon toga funkcija *deleteAllMeals()* prazni košaricu brišući sva jela iz nje, a narudžba se prebacuje u povijest narudžbi. Na taj način osigurano je efikasno upravljanje podacima u lokalnoj bazi čime se zadržava integritet narudžbi i omogućen je jednostavan prikaz korisničkih podataka.

Za prikaz jela u košarici nakon što ih korisnik odabere, korisnik može pregledati košaricu na zaslonu košarice. Zaslon košarice prikazuje sva jela koja su naručena te omogućuje korisniku da odluči hoće li potvrditi narudžbu ili će ukloniti neko od jela.

```
fun OrderScreen(navigator: DestinationsNavigator) {
    val orderScreenViewModel = getViewModel<OrderScreenViewModel>()
    val uiState by orderScreenViewModel.mealListUIState.collectAsState()
    val isOrderButtonEnabled = orderScreenViewModel.isOrderButtonEnabled.value
    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(text = "Current order") },
                navigationIcon = {
                    IconButton(onClick = { navigator.navigateUp() }) {
                        Icon(imageVector = Icons.Default.ArrowBack, contentDescription = null)
                    }
                }
            )
        },
        bottomBar = {
            Button(
                onClick = { orderScreenViewModel.lockOrder() },
                modifier = Modifier.fillMaxWidth(),
                enabled = isOrderButtonEnabled
            ) {
                Text(text = "Order")
            }
        }
    ) {
        when (uiState) {
            is MealListUIState.Success -> {
                OrderScreenContent(
                    mealList = (uiState as MealListUIState.Success).list,
                    onAddToOrder = { meal ->
                        meal.isInOrder = true

                    },
                    onRemoveFromOrder = { meal ->
                        orderScreenViewModel.removeMealFromOrder(meal)
                        meal.isInOrder = false
                    },
                    onMealItemClick = { meal ->
                        navigator.navigate(MealDetailScreenDestination(meal))
                    },
                     paddingValues = it
                )
            }

            MealListUIState.Loading -> {
                EmptyLisScreen()
            }

            MealListUIState.EmptyMealList -> {
                EmptyLisScreen()

            }

            MealListUIState.Error -> {
                ErrorScreen {
                    orderScreenViewModel.getAllOrderedMeals()
                }

            }
        }
    }
}
```

Slika 5.17. Prikaz OrderScreen funkcije

Na slici 5.17. u potpunosti je prikazan kod za *OrderScreen* gdje se na prikazanom zaslonu mogu pregledati naručena jela, ali se tu također nalazi i gumb za naručivanje hrane koji poziva funkciju *lockOrder()* koja finalizira narudžbu.



Slika 5.18. Prikaz prazne košarice

Slika 5.19. Prikaz košarice s dodanim jelom

Na slikama 5.18. i 5.19. su prikazane košarice na način da na prvoj slici niti jedno jelo nije dodano u košaricu te je onemogućeno izvršavanje narudžbe dok je na drugoj slici nakon dodanog jela u košaricu korisniku omogućeno naručivanje jela kako bi se ono spremilo u povijest narudžbi.

Nakon što korisnik potvrdi narudžbu, ona se sprema u povijest narudžbi koristeći funkciju *lockOrder()* koja se koristi za zaključavanje narudžbe i prebacivanje svih jela iz košarice u povijest narudžbi nakon čega se košarica prazni.

```
class OrderScreenViewModel(private val orderRepository: OrderRepository) : ViewModel() {
    private val _mealListUIState: MutableStateFlow<MealListUIState> =
        MutableStateFlow(MealListUIState.Loading)
    val mealListUIState = _mealListUIState.asStateFlow()
    private var mealList = listOf<Meal>()
    val isOrderButtonEnabled = mutableStateOf( value: false)

    init {
        getAllOrderedMeals()
    }

    fun getAllOrderedMeals() {
        viewModelScope.launch {
            try {
                orderRepository.getAllMeals().collect {
                    mealList = it
                    if (it.isEmpty()) {
                        _mealListUIState.value = MealListUIState.EmptyMealList
                        isOrderButtonEnabled.value = false

                    } else {
                        _mealListUIState.value = MealListUIState.Success(it)
                        isOrderButtonEnabled.value = true
                    }
                }
            } catch (e: Exception) {
                _mealListUIState.value = MealListUIState.Error
                isOrderButtonEnabled.value = false

            }

        }
    }

    fun removeMealFromOrder(meal: Meal) {
        viewModelScope.launch {
            orderRepository.deleteMeal(meal)
        }
    @RequiresApi(Build.VERSION_CODES.O)
    fun lockOrder() {
        viewModelScope.launch {
            orderRepository.insertOrder(
                Order(
                    mealList = mealList, time = LocalDateTime.now().toEpochSecond(
                        ZoneOffset.UTC
                    )
                )
            )
            orderRepository.deleteAllMeals()
        }
    }
}
```

Slika 5.20. Prikaz košarice s dodanim jelom

Na slici 5.20. prikazana je funkcija *lockOrder()* koja koristi *Kotlin*ove korutine kako bi se osiguralo da operacije koje se izvršavaju, a to je spremanje narudžbe u bazu, izvrše asinkrono čime se sprječava blokiranje glavnog korisničkog sučelja. Pozivom funkcije *insertOrder* iz *orderRepository*-ja narudžba se sprema u bazu na način da se stvara nova instanca *Order* objekta koji sadrži 2 stvari, a to je lista jela koju je korisnik odabrao u košarici te vrijeme narudžbe kad je narudžba finalizirana. Na koncu se poziva funkcija *deleteAllMeals* gdje se sva jela brišu iz košarice, odnosno iz lokalne baze podataka te se košarica prazni.

6. ZAKLJUČAK

Ovaj diplomski rad prikazao je razvoj jednostavne *Android* aplikacije za naručivanje hrane, s glavnim naglaskom na primjenu *MVVM*(*Model-View-ViewModel*) arhitekture. Kroz implementaciju aplikacije prikazano je kako *MVVM* olakšava razdvajanje poslovne logike od korisničkog sučelja te samim time omogućuje bolju modularnost, čišći kod te lakše održavanje i testiranje aplikacije.

Na samom početku rada teoretski su objašnjeni najčešći obrasci koji se koriste u razvoju *Android* aplikacija kao što su *MVC*(*Model-View-Controller*) te *MVP*(*Model-View-Presenter*), te je detaljno analizirana *MVVM* arhitektura. Usporedbom navedenih arhitekturnih obrazaca zaključeno je da *MVVM* ima prednost u složenijim aplikacijama zbog sposobnosti upravljanja podacima koristeći *ViewModel*-e te pristup raznim alatima koji su opisani u radu kao što je *StateFlow* i *LiveData.* U praktičnom dijelu rada prikazane su glavne funkcionalnosti aplikacije kao što su pregled ponude jela, dodavanje jela u košaricu, pregled detalja svakog pojedinog jela, proces naručivanja te na kraju i upravljanje s povijesti narudžbi. Aplikacija je implementirana koristeći moderne tehnologije kao što su *Jetpack Compose* koji je korišten za izgradnju korisničkog sučelja, *Retrofit* za komunikaciju s backendom te *Room* za lokalno upravljanje podacima.

Integracijom API-ja omogućeno je dohvaćanje podataka o jelima, dok je upravljanje ovisnostima implementirano korištenjem *Koina* čime je dodatno pojednostavljena organizacija i održavanje koda.

Zaključno, rad je prikazao kako primjena *MVVM* arhitekture u razvoju mobilnih aplikacija osigurava jasniju organizaciju koda, veću fleksibilnost u upravljanju podacima te bolju skalabilnost, čime je aplikacija održivija i adaptivnija na buduće nadogradnje. Aplikacija razvijena u ovome radu služi kao primjer funkcionalnog prototipa koji demonstrira ove prednosti te može poslužiti kao baza za daljnji razvoj složenijih aplikacija

SAŽETAK

U ovom diplomskom radu dizajnirana je i programski implementirana mobilna *Android* aplikacija za online naručivanje hrane. Aplikacija omogućava korisnicima pregled jelovnika, filtriranje jela prema kategorijama, dodavanje jela u košaricu te završavanje narudžbe s mogućnošću pregleda povijesti narudžbi. Razvoj aplikacije realiziran je u razvojnom okruženju *Android* Studio uz korištenje programskog jezika *Kotlin* i *Jetpack Compose* biblioteke za izradu korisničkog sučelja. Lokalno spremanje podataka ostvareno je pomoću Room baze podataka, dok je komunikacija s vanjskim API-jem implementirana korištenjem Retrofit biblioteke. Aplikacija je zasnovana na arhitekturi *Model-View-ViewModel* (*MVVM*) koja omogućava jasnu separaciju poslovne logike i korisničkog sučelja, poboljšava organizaciju koda te olakšava održavanje i testiranje aplikacije. Ovaj rad također demonstrira kako primjena *MVVM* arhitekture omogućava lakše upravljanje podatkovnim tokovima, bolju modularnost aplikacije i lakše integracije s backendom. Provedeno ispitivanje aplikacije pokazalo je uspješnu funkcionalnost kroz sve ključne procese, od upravljanja narudžbama do prikaza povijesti narudžbi.

Ključne riječi: mobilna *Android* aplikacija, *MVVM* arhitektura, online naručivanje hrane, *Kotlin*, Room, Retrofit.

ABSTRACT

APPLICATION OF MVVM ARCHITECTURE IN *ANDROID* APPLICATION DEVELOPMENT

This master thesis presents the design and implementation of a mobile *Android* application for online food ordering. The application allows users to browse the menu, filter meals by categories, add meals to the cart, and finalize orders with the option to view order history. The development was carried out in the *Android* Studio environment using the *Kotlin* programming language and the *Jetpack Compose* library for building the user interface. Local data storage is handled via the Room database, while communication with an external API is implemented using the Retrofit library. The application is based on the *Model-View-ViewModel* (*MVVM*) architecture, which facilitates a clear separation between business logic and the user interface, improving code organization and simplifying maintenance and testing. This thesis also demonstrates how *MVVM* architecture enables easier management of data flows, enhances the modularity of the application, and simplifies backend integration. Testing of the application showed successful functionality across all key processes, from managing orders to displaying order history.

Keywords: mobile *Android* application, *MVVM* architecture, online food ordering, *Kotlin*, Room, Retrofit.

LITERATURA

[1] G. Smith, "Understanding MVVM: A Guide for Android Developers", Journal of Mobile Technology, br. 10, sv. 10, str. 20-30, srpanj 2018.

[2] "Understanding Architectural Patterns in Software Development", Journal of Software Architecture, br. 15, sv. 1, str. 22-35, 2020.

[3] "Effective Android Programming with Architectural Patterns", International Journal of Mobile Application Development, br. 5, sv. 2, str. 75-85, 2021.

[4] MVC Framework – Introduction) [online], dostupno na: https://www.tutorialspoint.com/MVC_framework/MVC_framework_introduction.htm [15. srpanj 2024].

[5] MVP (Model View Presenter) Architecture Pattern in Android with Example) [online], dostupno na: https://www.geeksforgeeks.org/MVP-Model-View-Presenterarchitecture-pattern-in-Android-with-example/ [18.srpanj 2024].

[6] Model-View-ViewModel (MVVM) [online], dostupno na: https://www.techtarget.com/whatis/definition/Model-View-ViewModel#:~:text=Model%2DView%2DViewModel%20(MVVM)%20is%20a%20software%20design,logic%20and%20user%20interface%20controls. [20. srpanj 2024].

[7] E. Mixon, „AndroidOS“ TechTarget Mobile Computing, [online]. Dostupno na: https://www.techtarget.com/searchmobilecomputing/definition/Android-OS. [22. srpanj 2024].

[8] German, Kent. 2. kolovoza 2011. A brief history of Android phones. CNET, [online], dostupno na: https://www.cnet.com/tech/mobile/a-brief-history-of-Android-phones/ [23 srpanj 2024.]

[9] Ducrohet, Xavier; Norbye, Tor; Chou, Katherine, [online],dostupno na : https://Android-developers.Googleblog.com/2013/05/Android-studio-ide-built-for-Android.html [28 srpanj 2024.]

[10] Get started with Kotlin,[online], dostupno na: https://Kotlinlang.org/docs/getting-started.html#install-Kotlin [30. srpanj 2024.]

[11] Ben Lutkevich, DEFINITION Github, [online], dostupno na: https://www.techtarget.com/searchitoperations/definition/GitHub [1. kolovoz 2024].

[12] J. Dalbey, „Nonfunctional Requirements“ [online], dostupno na: http://users.csc.calpoly.edu/~jdalbey/SWE/QA/nonfunctional.html [2. kolovoz 2024].

[13] S. Anderson, "Why Good User Interface Design is Important," UX Planet, Medium, 2019. [online], dostupno na: https://uxplanet.org/why-good-user-interface-design-is-important-8a836e8a1a8 [3. kolovoz 2024].

[14] A. N. Tikhomirov, Android Development Patterns: Best Practices for Professional Developers, Addison-Wesley, 2016.

[15] M. Buckingham, "StateFlow and LiveData: Building Reactive Apps with Kotlin", KotlinConf, br. 4, str. 24-30, listopad 2020.

[16] Google, "Compose for Android: Declarative UI Framework", Google Developers [online], , dostupno na: https://developer.Android.com/Jetpack/Compose [posjećeno 4. kolovoz 2024.]

[17] D. Šimek, Dependency Injection in Android with Dagger and Koin, Packt Publishing, 2020.

[18] [2] M. Škorvaga, "Getting Started with Koin: Simple Dependency Injection for Kotlin", Koin Documentation [online], dostupno na: https://insert-Koin.io/docs/quickstart/Kotlin [5. kolovoz 2024.]

[19] Android Developers, Room Persistence Library [online], dostupno na: https://developer.Android.com/training/data-storage/room [6. kolovoz 2024].

[20] B. Nzivu, „Section,SImple GET request using Retrofit in Android“ [online], dostupno na: https://www.section.io/engineering-education/making-api-requests-using-retrofit-Android. [8. kolovoz 2023].

ŽIVOTOPIS

Ivan Šebetić rođen je 1.4.1999. u Osijeku. Nakon završene Osnovne škole Ivan Goran Kovačić, upisuje opću gimnaziju Antun Gustav Matoš u Đakovu koju završava sva 4 razreda s vrlo dobrim uspjehom te 2018. godine upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer računarstvo. Preddiplomski studij uspješno završava 2019. godine,te nakon toga sljedeća 2 ljeta provodi na usavršavanju engleskog jezika u Americi na Work and Travel programu. Tečno govori engleski, te poznaje različite programske jezike kao što su C, C++, C#, Java, *Kotlin*.