MASARYKOVA
UNIVERZITA

FAKULTA INFORMATIKY

Mobilní aplikace pro on-line tržiště

Bakalářská práce

TOMÁŠ ONDRÁČEK

Brno, jaro 2026

MASARYKOVA
UNIVERZITA

FAKULTA INFORMATIKY

Mobilní aplikace pro on-line tržiště

Bakalářská práce

TOMÁŠ ONDRÁČEK

Vedoucí práce: Mgr. Luděk Bártek, Ph.D.

Katedra počítačových systémů a komunikací

Brno, jaro 2026


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Prohlašuji, že jsem při zpracování této práce využil nástroje umělé inteligence v souladu s principy akademické integrity. Tyto nástroje byly použity jako podpůrný prostředek (např. pro jazykové úpravy textu a konzultaci návrhu řešení). Za veškerý obsah práce nesu plnou odpovědnost.

Tomáš Ondráček

Vedoucí práce: Mgr. Luděk Bártek, Ph.D.

Poděkování

Děkuji vedoucímu práce, Mgr. Luďkovi Bártkovi, Ph.D., za cenné rady a vedení během psaní této práce. Dále děkuji své rodině a přátelům za jejich podporu a trpělivost.

Shrnutí

Tato bakalářská práce se zabývá návrhem a implementací mobilní aplikace pro podporu lokálního prodeje produktů menších producentů. Cílem práce je vytvořit nástroj, který umožní jednoduchou prezentaci nabídky producentů a její vyhledávání zákazníky na základě aktuální polohy uživatele.

V práci je provedena analýza existujících řešení zaměřených na lokální prodej a jejich omezení. Na základě této analýzy jsou stanoveny požadavky na navrhované řešení se zaměřením na snadnou použitelnost, mapové zobrazení a vyhledávání podle polohy uživatele.

Dále je navržena architektura aplikace a popsána její implementace pro platformu Android s využitím moderních vývojových nástrojů a architektonických principů. Výsledná aplikace umožňuje vytváření a správu obchodů, zobrazení obchodů v okolí uživatele a filtrování výsledků podle zvolených kritérií.

Součástí práce je také testování aplikace a její nasazení, včetně zhodnocení dosažených výsledků a návrhů pro další rozvoj. Výsledkem práce je funkční mobilní aplikace připravená pro použití reálnými uživateli.

Klíčová slova

mobilní aplikace, lokální prodej, drobní producenti, mapové vyhledávání, geolokační služby, návrh aplikace

Obsah

Seznam tabulek

Seznam obrázků

Úvod

V posledních letech roste zájem zákazníků o nákup potravin a dalších produktů od menších producentů, lokálních farem nebo domácích pěstitelů [60, 61]. Přestože se tento trend dále rozvíjí, dostupné nástroje pro prezentaci nabídky drobných a domácích producentů nejsou vždy zaměřeny na jednoduchou a samostatnou propagaci jejich produktů pro zákazníky v okolí.

Lokální prodej je často založen na tradičních postupech, jako jsou například reklamní poutače v místě prodeje nebo osobní doporučení. Tyto metody mohou mít omezený dosah. Producentům i zákazníkům by mohl pomoci nástroj, který umožní přehlednější a dostupnější prezentaci jejich nabídky.

Cílem této práce je navrhnout a implementovat aplikaci pro platformu Android, která se snaží řešit uvedenou problematiku. Aplikace umožní uživatelům vytvářet a spravovat vlastní obchody pro zobrazení jejich nabídky, zobrazovat obchody na mapě a filtrovat pomocí zvolených kritérií.

Struktura práce je následující: V první kapitole je představena motivace pro tuto aplikaci a přehled existujících řešení. Druhá kapitola se věnuje návrhu aplikace, včetně specifikace požadavků, návrhu diagramů a výběru technologií. Třetí kapitola popisuje architektonická rozhodnutí podporující vývoj a údržbu aplikace. Následují kapitoly věnované implementaci a testování. Poslední kapitola popisuje proces vydání a nasazení aplikace pro platformu Android. Práce je zakončena shrnutím dosažených výsledků a zhodnocením zvolených postupů.

1 Analýza

V oblasti lokálního prodeje potravin a dalších produktů se malé farmy a domácí producenti mohou potýkat s omezenými možnostmi, jak efektivně oslovit zákazníky ve svém okolí bez nutnosti prodávat své produkty prostřednictvím velkých řetězců nebo centralizovaných výkupních míst.

Přestože poptávka po lokálních produktech v dnešní době stále více roste [60, 61], způsoby jejich propagace nejsou vždy dostatečně efektivní. Výsledkem je, že zákazníci často neví o tom, jaké produkty jsou v jejich blízkosti dostupné, a producenti tak nemusí plně využít potenciální zájem zákazníků.

1.1 Motivace

Navrhovaná aplikace cílí právě na zmiňované drobné nebo domácí producenty. Typickým příkladem může být včelař prodávající med, zahrádkář nabízející přebytky zeleniny, producent domácích sirupů, chovatel prodávající domácí vejce nebo lokální vinař. Takoví producenti zpravidla nemají vlastní webové stránky a nechtějí řešit složité registrační či schvalovací procesy. Navrhovaná aplikace poskytuje jednoduchý způsob, jak být dohledatelní pro zákazníky v okolí.

Z pohledu producenta existují obvykle dvě možnosti distribuce. První možností je centrální výkup, který snižuje jejich podíl na zisku a eliminuje přímý kontakt se zákazníkem. Druhou možností je lokální propagace prostřednictvím reklamních poutačů v místě prodeje, osobního doporučení nebo příspěvků na sociálních sítích. Tato možnost však může mít často omezený dosah.

Typickým využitím aplikace může být:

- zákazník otevře aplikaci, zobrazí mapu okolí, vyfiltruje požadovanou kategorii a vzdálenost a následně kontaktuje producenta nabízejícího požadované produkty,
- producent vytvoří obchod, přidá nabízené produkty, uvede kontaktní informace a přidá fotografie produktů a místa prodeje.

1.2 Existující aplikace a projekty

V rámci analýzy byly zkoumány existující aplikace a projekty, které se zaměřují na podporu lokálních farmářů a malých producentů. Cílem bylo zjistit, zda již existuje nástroj zaměřený na malé nebo domácí producenty, který by splňoval požadavky na jednoduchou a přístupnou prezentaci nabídky jejich domácích produktů pro potenciální zájemce v jejich okolí. Analýza vychází z veřejně dostupné dokumentace jednotlivých platforem a jejich způsobu fungování.

1.2.1 Farmáři z regionu

Farmáři z regionu [25] je mapová aplikace, která sdružuje lokální farmáře a poskytuje informační katalog jejich produktů a provozoven [44]. Projekt umožňuje zobrazit jednotlivé farmy podle regionu a zprostředkovává základní údaje o jejich nabídce.

Platforma je zaměřená především na oficiální farmáře a výrobce. Obsahuje zemědělce, kteří o zařazení projevili zájem v rámci mapování produkce v Jihomoravském kraji. Další producenti se mohou přihlásit prostřednictvím formuláře a požádat o přidání do aplikace, nejedná se ale o otevřený marketplace, ve kterém by mohl kterýkoli uživatel okamžitě vytvářet vlastní obchody a spravovat svou nabídku. Proces přidání do aplikace je řízený a je zaměřený spíše na oficiální farmáře a výrobce než na drobné domácí producenty.

1.2.2 Scuk.cz

Scuk.cz [47] je česká online platforma zaměřená na podporu lokálních farmářů a malých až středních producentů potravin [43]. Scuk klade důraz na kvalitu a původ produktů, a proto je vstup do platformy podmíněn schválením ze strany provozovatele.

Platforma však není určena běžným domácím producentům, kteří chtějí jednoduše prezentovat svou nabídku. Uživatelé si nemohou samostatně vytvářet obchody ani spravovat vlastní nabídku bez zapojení do relativně složitého schvalovacího procesu [33]. Scuk navíc funguje jako nákupní platforma, obsahuje košík, objednávky a odběrná místa, takže jeho cílem je zprostředkování prodeje, nikoli pouze jednoduchá a přístupná prezentace nabídky.

Dalším rozdílem je způsob zobrazení prodejců. Scuk využívá především katalog a filtrování produktů, zatímco mapové zobrazení není centrálním prvkem platformy. To se liší od cíle této aplikace, která se soustředí právě na mapové zobrazení obchodů a snadnou prezentaci nabídky bez nutnosti formální registrace provozovny.

1.2.3 Facebook skupiny a Facebook marketplace

Facebook [23] nabízí možnost vytváření skupin, které mohou být často zaměřeny na konkrétní témata, o která má uživatel zájem. Tyto skupiny však postrádají možnost zobrazit si lokální nabídky a neumožňují přehledné zobrazení produktů. Příspěvky se navíc mohou ve skupině rychle ztratit nebo nemusí být aktuální.

Naopak Facebook Marketplace [24] umožňuje přehledné zobrazení produktu, ale je zaměřený spíše na jednotlivé inzeráty než na dlouhodobou či proměnlivou nabídku. Nenabízí ani přehlednou nabídku producenta a není zaměřený na lokální prodej potravin a podobných produktů.

1.2.4 Google Maps Places

Google Maps [31] sice umožňuje přehledné mapové zobrazení prostřednictvím služby Google Business Profile, ale ta je primárně určena pro oficiálně registrované podniky a vyžaduje proces ověření.

Tento model tak může představovat bariéru pro drobné nebo příležitostné producenty, kteří nepůsobí jako formálně registrované provozovny.

1.3 Výsledek analýzy

Analýza dostupných nástrojů ukázala, že existující řešení jsou zpravidla zaměřena buď na oficiálně registrované producenty, nebo na zprostředkování samotného prodeje prostřednictvím centralizované platformy. Jednoduchý nástroj umožňující drobným nebo domácím producentům samostatně vytvářet a spravovat prezentaci své nabídky s důrazem na mapové zobrazení a lokální dosah není v analyzovaných řešeních primárním cílem.

Dalším důležitým zjištěním je, že ačkoliv některá analyzovaná řešení poskytují mobilní aplikaci nebo jsou přizpůsobena pro zobrazení na mobilních zařízeních, jejich návrh není primárně zaměřen na scénář vyhledávání lokální nabídky drobných producentů na základě aktuální polohy uživatele. Vyhledávání lokálních produktů je přitom často spojeno s mobilním zařízením a jeho použitím v terénu, kde uživatel očekává rychlé zobrazení relevantních výsledků v okolí. Tento způsob použití je v existujících řešeních podporován pouze částečně, což vytváří prostor pro návrh specializovaného nástroje zaměřeného právě na tento scénář.

Tabulka 1.1 shrnuje srovnání analyzovaných platforem podle vybraných kritérií. Hodnocení vyjadřuje přítomnost dané vlastnosti (✓), její částečné naplnění (∼) nebo její absenci (✗).

Žádné z analyzovaných řešení přitom nesplňuje současně požadavky na podporu drobných producentů, mapové zobrazení jako primární prvek a jednoduchou správu nabídky bez nutnosti schvalovacího procesu. Tato skutečnost vytváří prostor pro návrh aplikace, která by byla zaměřena na snadnou prezentaci lokální nabídky drobných producentů, vyhledávání podle aktuální polohy uživatele a zároveň by zohledňovala použití na mobilních zařízeních.

Tabulka 1.1: Hodnotící matice analyzovaných platforem

Kritérium	Farmáři z regionu	Scuk.cz	Facebook	Google Maps	Vlastní aplikace
Samostatná registrace producenta	∼[a]	∼[b]	✓	∼	✓
Zaměření na drobné producenty	∼	∼	✗	✗	✓
Mapové zobrazení jako primární prvek	✓	✗	✗	✓	✓
Nutnost schvalovacího procesu	✓	✓	✗	∼[c]	✗
Zprostředkování přímého prodeje	✗	✓	∼[d]	✗	✗
Jednoduchá prezentace a správa nabídky	∼	✗	✗	∼	✓

Legenda: ✓ Ano, ∼ Částečně, ✗ Ne.
[a] Registrace probíhá formou žádosti o zařazení.
[b] Vstup do platformy podléhá schvalovacímu procesu.
[c] Založení profilu vyžaduje ověření podnikatelského subjektu.
[d] Prodej probíhá formou jednotlivých inzerátů, nikoli dlouhodobé nabídky.

2 Návrh aplikace

Návrh aplikace vychází z výsledků analýzy existujících řešení, která ukázala absenci nástroje umožňujícího drobným producentům jednoduchou a samostatnou prezentaci nabídky bez nutnosti komplexních schvalovacích procesů, s důrazem na mapové zobrazení a zaměřením na efektivní použití na mobilních zařízeních v terénu. Cílem návrhu aplikace je definovat její funkční rozsah, strukturu, uživatelské rozhraní a technologické řešení tak, aby odpovídaly potřebám cílových uživatelů a zaměření aplikace. K tomu byly využity standardní techniky softwarového inženýrství.

Kapitola postupně představuje specifikaci funkčních a nefunkčních požadavků, návrhové diagramy a návrh uživatelského rozhraní. Závěrem je zdůvodněna volba použitých technologií, které tvoří základ implementace aplikace.

2.1 Specifikace požadavků

Tato část specifikuje požadavky na navrhovanou aplikaci. Požadavky vymezují očekávané chování systému a jeho kvalitativní vlastnosti z pohledu uživatelů i provozu aplikace. Specifikace tvoří základ pro návrh architektury a implementaci aplikace.

2.1.1 Funkční požadavky

Funkční požadavky definují chování systému a služby, které poskytuje svým uživatelům [49]. Aplikace rozlišuje dva základní typy uživatelů: běžné návštěvníky aplikace a registrované uživatele.

- Prohlížení a vyhledávání obchodů
  - Systém umožňuje zobrazit obchody v mapovém nebo seznamovém zobrazení.
  - Systém umožňuje filtrovat obchody podle kategorie, maximální vzdálenosti od aktuální polohy a minimálního průměrného hodnocení.

  - Systém umožňuje zobrazit detail obchodu, včetně popisu, fotografií, nabídky produktů, otevírací doby a kontaktních údajů.

- Správa uživatelského účtu
  - Systém umožňuje vytvoření uživatelského účtu.
  - Systém umožňuje přihlášení a odhlášení uživatele.
  - Systém umožňuje registrovanému uživateli upravovat své osobní a kontaktní údaje.

- Správa obchodů
  - Systém umožňuje registrovanému uživateli vytvářet nové obchody.
  - Systém umožňuje registrovanému uživateli upravovat a odstraňovat obchody, jejichž je vlastníkem.
  - Systém umožňuje registrovanému uživateli přidávat a upravovat u svých obchodů název, popis, kategorie obchodu, fotografie, nabídku produktů a otevírací dobu.

- Recenze
  - Systém umožňuje registrovanému uživateli vytvářet recenze obchodů jiných uživatelů.

Aplikace neslouží jako e-shop a neumožňuje přímý nákup ani vytváření objednávek. Jejím cílem je poskytovat prostředek pro prezentaci nabídky domácích producentů a umožnit zájemcům snadno vyhledat obchody v jejich okolí. Aplikace nevstupuje do samotného obchodního vztahu mezi producentem a zákazníkem.

2.1.2 Nefunkční požadavky

Nefunkční požadavky popisují omezení a vlastnosti systému, které se netýkají přímo funkcionality, ale ovlivňují kvalitu, použitelnost a provoz systému [49].

- Aplikace je implementována jako nativní mobilní aplikace pro platformu Android [5] a využívá nativní API zařízení.

- Uživatelské rozhraní musí být přehledné, konzistentní a snadno použitelné i pro méně technicky zdatné uživatele.
- Systém musí poskytovat dostatečně rychlou odezvu uživatelského rozhraní při práci s mapovým i seznamovým zobrazením obchodů a dalšími funkcemi aplikace.
- Přístup k datům musí být omezen pouze na oprávněné uživatele prostřednictvím autentizace a řízení přístupových práv.
- Architektura aplikace musí umožňovat další rozšiřování funkcionality bez zásadních zásahů do existujícího kódu.

2.2 Diagram případů užití

Diagram případů užití (Use Case Diagram) slouží k přehlednému znázornění základních funkcí aplikace a interakcí mezi uživateli a systémem [49]. Diagram vychází ze specifikace funkčních požadavků a zachycuje chování aplikace z pohledu jednotlivých rolí uživatelů. Přehled aktérů a případů užití je znázorněn na obrázku 2.1.

V rámci aplikace jsou rozlišeny tři základní role: obecný uživatel (User), zákazník (Customer) a prodejce (Seller). Role zákazníka a prodejce jsou modelovány jako specializace obecného uživatele pomocí vztahu generalizace. Společné případy užití jsou tak sdíleny všemi specializovanými rolemi. Tyto specializované role jsou rozděleny podle toho, jakým způsobem chce uživatel aplikaci využívat a co je jeho cílem při používání aplikace. Primárním cílem zákazníka je procházet obchody a jejich nabídku, zatímco primárním cílem prodejce je vytvářet a spravovat obchody a prezentovat svou nabídku zájemcům.

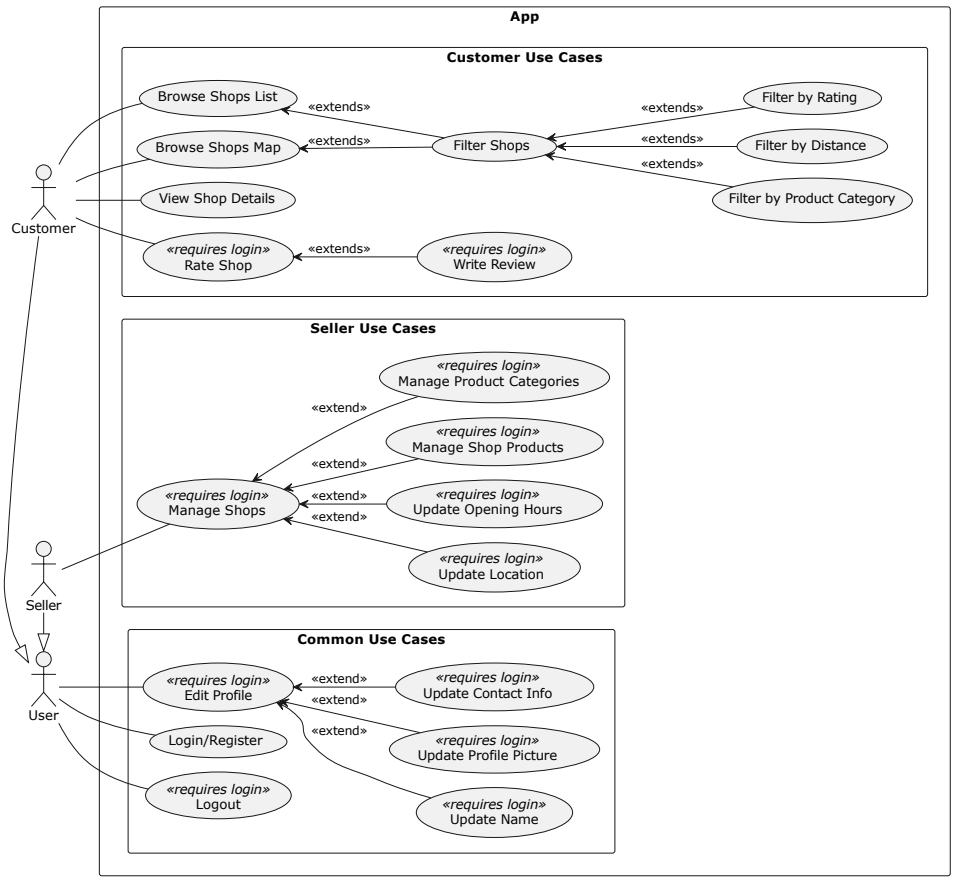
Obecný uživatel zahrnuje základní případy užití související s registrací, přihlášením, odhlášením a úpravou uživatelského profilu. Případ užití úpravy profilu je dále rozšířen o konkrétní podřízené případy užití (například aktualizace jména, profilového obrázku nebo kontaktních údajů). Případy užití odhlášení a úpravy profilu jsou dostupné pouze přihlášeným uživatelům.

Zákazník může procházet obchody v mapovém nebo seznamovém zobrazení, zobrazovat detail obchodu a filtrovat obchody podle

kategorií, vzdálenosti nebo hodnocení. Funkce filtrování obchodů je modelována pomocí vztahu «*extend*», kdy jednotlivé typy filtrů (podle kategorií, vzdálenosti nebo hodnocení) rozšiřují základní případ užití procházení obchodů. Pokud je zákazník přihlášen, může vytvářet recenze obchodů ostatních uživatelů.

Role prodejce zahrnuje případy užití související se správou obchodů. Případ užití správy obchodů je dále rozšířen o konkrétní operace, jako je aktualizace polohy, otevírací doby, správa nabídky produktů a kategorií. Tyto případy užití jsou rovněž podmíněny přihlášením uživatele.

Případy užití vyžadující přihlášení jsou v diagramu označeny stereotypem «requires login».



Obrázek 2.1: Diagram případů užití aplikace

2.3 Diagram tříd

Diagram tříd (Class Diagram) popisuje strukturu aplikace z pohledu hlavních entit, jejich atributů a vztahů mezi nimi [49]. Diagram vychází ze specifikace funkčních požadavků a diagramu případů užití a slouží jako podklad pro návrh datového modelu a implementaci aplikační logiky. Modeluje základní entity tak, aby systém mohl plnit požadované funkce a zajišťoval konzistenci dat. Přehled hlavních tříd a jejich vztahů je znázorněn na obrázku 2.2.

2.3.1 Entita uživatele

Jednou ze základních entit je uživatel (User), který reprezentuje registrovaného uživatele aplikace. Uživatel je identifikován jednoznačným identifikátorem a obsahuje základní osobní údaje, profilový obrázek a kontaktní informace.

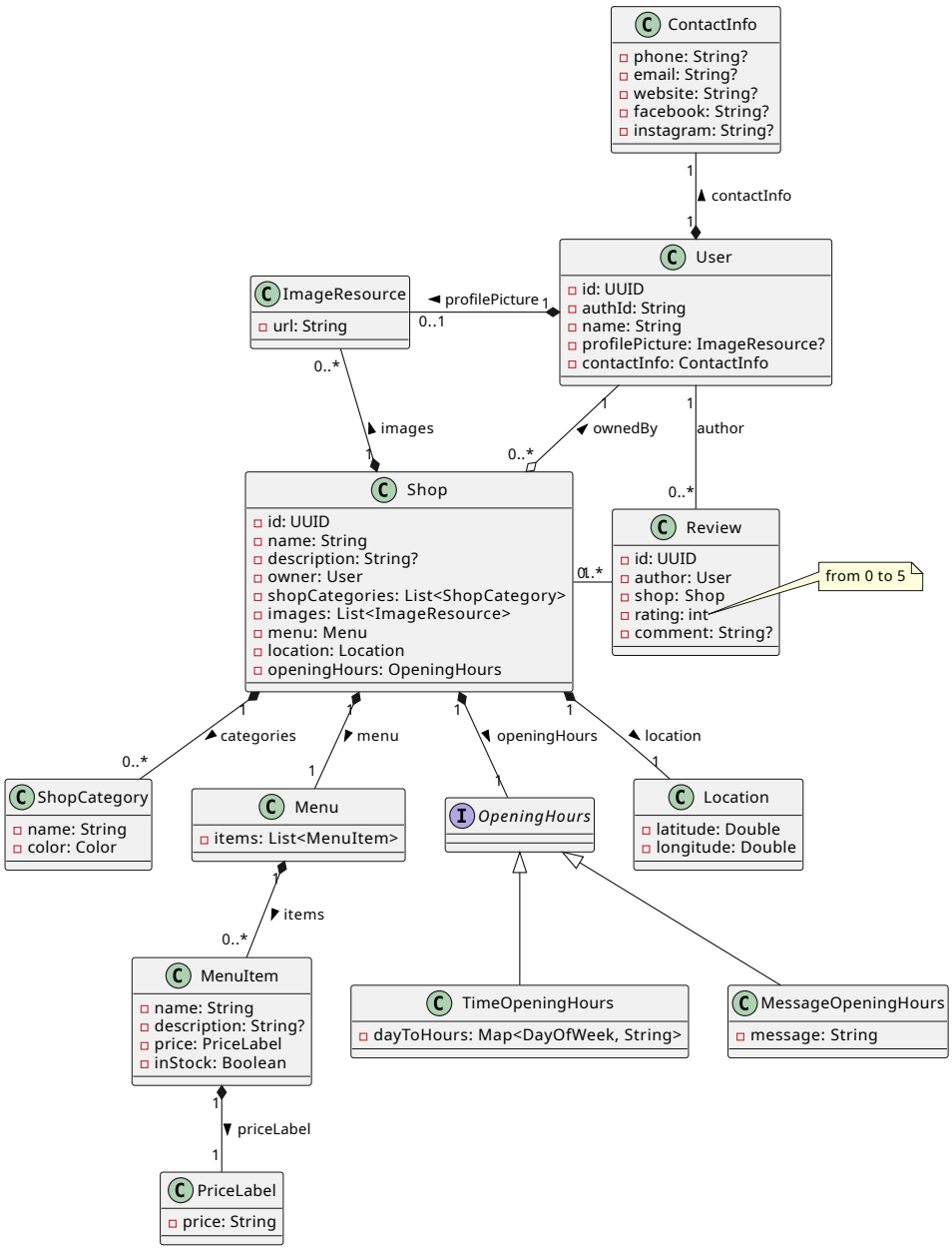
Identifikátor je typu UUID, což zajišťuje jeho jedinečnost napříč celým systémem. Tento identifikátor představuje doménovou identitu uživatele, která je oddělena od identifikátoru používaného autentizačním mechanismem. Ten je reprezentován samostatným atributem `authId`, což umožňuje flexibilitu v případě změny autentizačního systému nebo implementace vlastního mechanismu autentizace.

Kontaktní údaje jsou modelovány samostatnou třídou `ContactInfo`, která je s uživatelem svázána kompozicí, jelikož bez existence uživatele nemají samostatný význam. Profilový obrázek je volitelný a je reprezentován hodnotovým objektem `ImageResource`, který uchovává odkaz na externě uložený obrazový soubor.

2.3.2 Entita obchodu

Druhou klíčovou entitou je obchod (Shop), který reprezentuje obchod vytvořený registrovaným uživatelem, kde může nabízet své produkty. Obchod je také identifikován jednoznačným identifikátorem typu UUID. Obchod obsahuje atributy: id, název, popis, vlastník, seznam kategorií, obrázky, nabídku produktů, polohu a otevírací dobu. Základní atributy jako název a popis jsou typu `String` a popis je volitelný.

Obchod je svázán s uživatelem prostřednictvím asociace, která vyjadřuje vlastnictví obchodu konkrétním uživatelem.



Obrázek 2.2: Diagram tříd aplikace

Kategorie obchodu je reprezentována hodnotovým objektem typu `ShopCategory`, který obsahuje název kategorie a barvu pro vizuální odlišení. Vztah mezi obchodem a kategoriemi je modelován jako kompozice, protože kategorie jsou vlastněny konkrétním obchodem. Oproti možnosti modelovat kategorie jako samostatné entity sdílené mezi obchody poskytuje tento přístup větší flexibilitu jak při volbě názvu kategorie, tak i při volbě její barvy. Uživatel si tak může pro kategorii zvolit libovolnou kombinaci názvu a barvy. Požadované filtrování obchodů podle kategorií bude realizováno pomocí normalizovaných názvů kategorií ze všech obchodů.

Obrázky obchodu jsou reprezentovány jako seznam již zmiňovaných hodnotových objektů `ImageResource`.

Nabídka produktů je modelována hodnotovým objektem `Menu`, který obsahuje seznam položek nabídky typu `MenuItem`. Každá položka obsahuje název, volitelný popis, cenu a informaci o dostupnosti. Cena je reprezentována hodnotovým objektem `PriceLabel`, který ji uchovává jako řetězec. Uživatel tak může cenu popsat vzhledem k měně a jednotce v jaké se produkt nabízí (například „150 Kč/kg“). Vzhledem k tomu, že aplikace neslouží k přímému nákupu ani k provádění filtrování či řazení podle ceny produktů, cena je tímto způsobem modelována jako textový údaj určený pouze k informační prezentaci nabídky.

Otevírací doba obchodu je reprezentována rozhraním `OpeningHours`, které má dvě implementace: `TimeOpeningHours` a `MessageOpeningHours`.

První implementace uchovává otevírací dobu jako mapu dnů v týdnu na otevírací hodiny. Tato implementace se využívá v případě, že producent je dostupný v pravidelných časech. Druhá implementace uchovává otevírací dobu jako volitelnou zprávu v případě, že producent není dostupný v pravidelných časech a chce například uvést, aby ho zájemce nejdříve kontaktoval.

Poloha obchodu je modelována hodnotovým objektem `Location`, který obsahuje zeměpisnou šířku a délku jako desetinná čísla.

2.3.3 Entita recenze

Poslední entitou je recenze (Review), která reprezentuje hodnocení obchodu uživatelem. Recenze je identifikována jednoznačným identifi-

kátorem typu UUID a obsahuje atributy: id, autor, obchod, hodnocení a volitelný komentář. Recenze je svázána s uživatelem prostřednictvím asociace, která vyjadřuje autora recenze. S obchodem je recenze rovněž svázána asociací, jelikož recenze reprezentuje interakci mezi uživatelem a konkrétním obchodem, nikoli vnitřní součást těchto entit. Přestože recenze bez existence obchodu ztrácí svůj praktický význam, není modelována jako kompozitní součást obchodu, protože má vlastní identitu a samostatný životní cyklus.

2.4 Návrh uživatelského rozhraní

Návrh uživatelského rozhraní vychází z funkčních požadavků a diagramu případů užití. Cílem je vytvořit přehledné a konzistentní prostředí umožňující efektivní práci s aplikací.

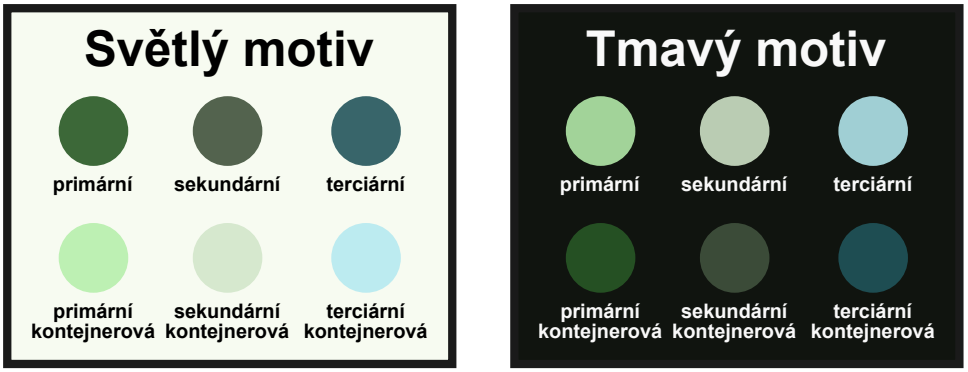
2.4.1 Barevné schéma

Při návrhu byly využity principy Material Designu [37], které zajišťují jednotný a moderní vzhled aplikace. Material Design je designový systém vyvinutý společností Google, který poskytuje sadu doporučených postupů a komponent pro tvorbu uživatelských rozhraní.

Barevná paleta byla vygenerována pomocí nástroje Material Theme Builder, který generuje barevné schéma v souladu s principy Material Designu [22]. Po zadání primární barvy nástroj vytvoří kompletní sadu barev pro světlé i tmavé téma aplikace, která působí konzistentně a zajišťuje dostatečný kontrast pro čitelnost textu a vizuální přitažlivost aplikace. Pro primární barvu byla zvolena tmavě zelená (#3C6838), která byla vybrána s ohledem na tematické zaměření aplikace na lokální potraviny a zemědělství.

Významnou barvou v Material Design paletě je barva *surface*, která je používána jako pozadí pro většinu obrazovek aplikace (na obrázku 2.3 je použita jako pozadí rámce „Světlý motiv“, respektive „Tmavý motiv“). Velmi významnými barvami jsou primární, sekundární a terciární a jejich kontejnerové varianty (zobrazeny na obrázku 2.3). Primární, sekundární a terciární barvy jsou používány pro zvýraznění důležitých prvků uživatelského rozhraní, jako jsou tlačítka, ikony a další interakční prvky. Kontejnerové varianty těchto

barev jsou používány pro větší plochy, které jsou třeba zvýraznit, jako jsou karty nebo větší tlačítka [14].



Obrázek 2.3: Barevná paleta aplikace

2.4.2 Rozvržení obrazovek

Návrh rozvržení obrazovek aplikace vychází z funkčních požadavků a diagramu případů užití.

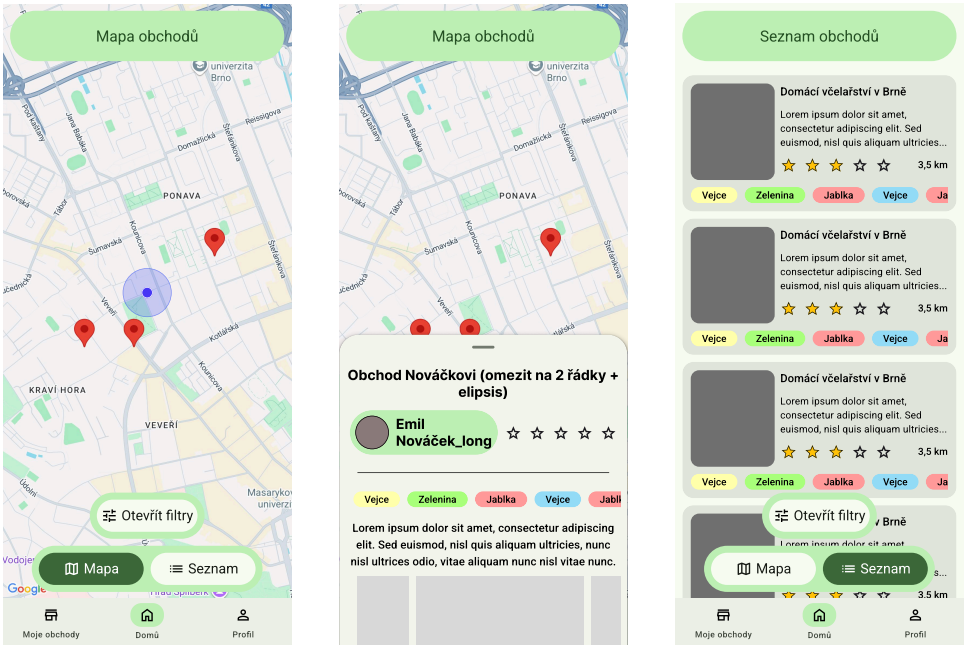
Ve spodní části základních obrazovek se nachází **navigační lišta** (obrázky 2.4), která obsahuje tři ikony představující tři hlavní obrazovky aplikace: „Moje obchody“, „Úvodní obrazovka“ a „Profil uživatele“. Díky ní může uživatel snadno navigovat mezi právě těmito základními obrazovkami aplikace. Sekce „Moje obchody“ je dostupná pouze přihlášeným uživatelům a ikona pro tuto sekci se zobrazuje pouze pokud je uživatel přihlášen. Ikona „Profil uživatele“ přesměruje uživatele na obrazovku profilu uživatele, nebo na obrazovku pro přihlášení v závislosti na tom, zda je uživatel přihlášen či nikoliv. Ikona „Úvodní obrazovka“ přesměruje uživatele vždy na úvodní obrazovku, která je výchozím zobrazením aplikace.

Úvodní obrazovkou aplikace je obrazovka, kde uživatel může zvolit mezi mapovým a seznamovým zobrazením obchodů.

Výchozím zobrazením je mapové zobrazení, kde mapa ukazuje polohu uživatele a okolní obchody jako značky na mapě (obrázek 2.4a). Při kliknutí na značku obchodu se zobrazí spodní panel s detaily obchodu (obrázek 2.4b).

V seznamovém zobrazení jsou obchody zobrazeny jako karty v seznamu, které obsahují pouze základní informace o obchodu, aby zůstaly přehledné, a přitom obchod dostatečně popisovaly (obrázek 2.4c). Zde se při kliknutí na kartu obchodu zobrazí detail obchodu na samostatné obrazovce.

Obě zobrazení mají společnou spodní nabídku, kde může uživatel přepínat mezi mapovým a seznamovým zobrazením a otevřít dialog pro filtry.



(**a**) Mapové zobrazení obchodů

(**b**) Detail obchodu v mapovém zobrazení

(**c**) Seznamové zobrazení obchodů

Obrázek 2.4: Přehled uživatelského rozhraní aplikace

Dialog filtrů obsahuje záložky pro nastavení požadovaných kategorií obchodů, maximální vzdálenosti od aktuální polohy a minimálního průměrného hodnocení obchodů (obrázek 2.5). V záložce filtru kategorií se názvy kategorií pro filtrování doporučují na základě jejich četnosti v existujících obchodech, aby uživatel mohl snadno vybrat nejčastěji používané kategorie.

Na obrazovce profilu může uživatel upravit své osobní údaje, jak bylo specifikováno ve funkčních požadavcích. Mezi tyto údaje patří

Obrázek 2.5: Dialog pro nastavení filtrů

jméno, kontaktní informace a profilový obrázek. V případě profilového obrázku může uživatel buď nahrát obrázek ze svého zařízení, nebo přímo vyfotit nový obrázek pomocí fotoaparátu zařízení. Uživatel se zde také může odhlásit z aplikace.

Na obrazovce **Moje obchody** má uživatel přehled o všech obchodech, které vytvořil. Obchody jsou zobrazeny jako karty v seznamu, podobně jako v seznamovém zobrazení na úvodní obrazovce. Každá karta má však navíc tlačítka pro úpravu a odstranění obchodu. Tato obrazovka také umožňuje uživateli spustit proces vytváření nového obchodu nebo úpravy existujícího.

Proces vytváření a úpravy obchodu je rozdělen do několika kroků a každý krok je zobrazen na samostatné obrazovce. Rozdělení procesu do více kroků bylo zvoleno s cílem snížit kognitivní zátěž uživatele [58]. Místo velkého množství informací a vstupních polí na jedné obrazovce jsou tak jednotlivé prvky procesu rozděleny do menších, významově seskupených částí (jako jednotlivé obrazovky procesu). Uživatel může mezi těmito kroky libovolně navigovat pomocí dvou spodních tlačítek „Předchozí“ a „Další“.

V prvním kroku uživatel zadává název a polohu obchodu na mapě (obrázek 2.6a). Jsou to kromě samotné nabídky nejvýznamnější údaje, které obchod popisují a kterých si zájemci všimnou jako první.

V druhém kroku uživatel přidává fotografie a popis obchodu (obrázek 2.6b). Fotografie lze přidat ze zařízení nebo přímo pořídit pomocí

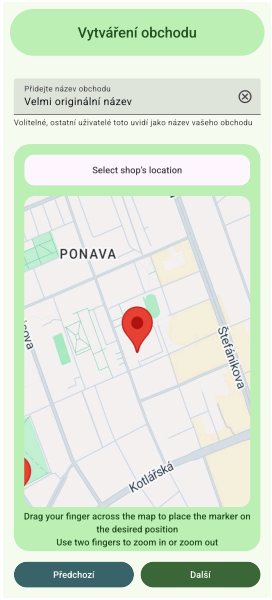
fotoaparátu. Fotografie může uživatel přidávat, mazat nebo měnit jejich pořadí.

Ve třetím kroku uživatel přidává kategorie obchodu a položky nabídky (obrázek 2.6c). Kategorie jsou reprezentovány názvem a barvou, kterou si uživatel může zvolit z předdefinované palety barev. Názvy kategorií se uživatelovi doporučují na základě jejich četnosti v existujících obchodech, aby uživatel mohl snadno vybrat nejpopulárnější kategorie a jeho obchod byl tak lépe dohledatelný. Každá položka nabídky je zobrazena jako karta v seznamu a kromě zobrazení údajů obsahuje také tlačítka pro úpravu a odstranění položky nabídky. Přidání nové nebo úprava stávající položky nabídky je realizována pomocí dialogu pro zadání všech potřebných údajů pro danou položku.

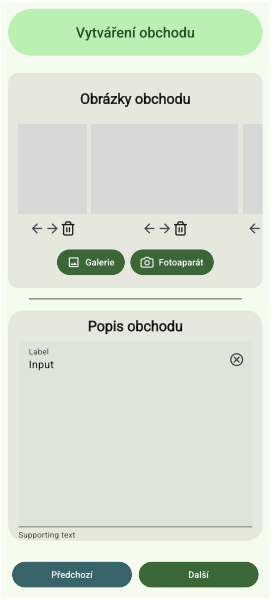
Ve čtvrtém kroku uživatel zadává otevírací dobu obchodu (obrázek 2.6d). Uživatel má na výběr mezi dvěma způsoby zadání otevírací doby: jako pravidelnou denní otevírací dobu, kde může uživatel ke každému dni v týdnu zadat otevírací hodiny (například „9:00-11:00, 13:00-17:00“), nebo jako nepravidelnou otevírací dobu formou volitelné zprávy, kde uživatel může zadat libovolný text (například „Kontaktujte mě předem telefonicky“).

Posledním krokem je shrnutí, kde může uživatel zkontrolovat všechny zadané údaje před vytvořením nebo úpravou obchodu.

Návrh uživatelského rozhraní byl využit především jako nástroj pro strukturování aplikace a plánování implementace jednotlivých obrazovek. Cílem bylo definovat hlavní scénáře použití a průchod uživatele aplikací, aby implementace probíhala systematicky a nevznikala ad hoc.

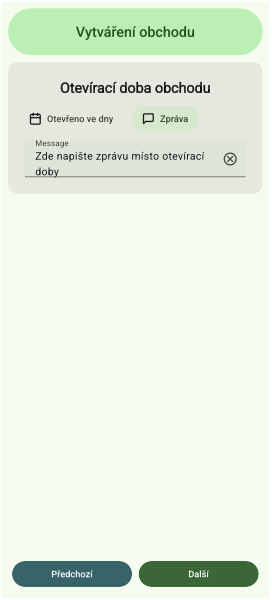


(**a**) Krok 1: Název a poloha obchodu



(**b**) Krok 2: Obrázky a popis obchodu

(**c**) Krok 3: Kategorie obchodu a nabídka produktů



(**d**) Krok 4: Otevírací doba obchodu – zpráva

Obrázek 2.6: Kroky vytváření a úpravy obchodu

2.5 Výběr technologií

Volba použitých technologií vychází z požadavků aplikace a zohledňuje faktory jako je efektivita vývoje, uživatelský zážitek, škálovatelnost a údržba aplikace. Cílem je zvolit technologie vhodné pro implementaci navrženého řešení s ohledem na rozsah bakalářské práce a požadovanou funkčnost aplikace. Volba jednotlivých technologií je dále zdůvodněna.

2.5.1 Volba cílové platformy

Při volbě platformy pro vývoj aplikace se obvykle nabízejí tři hlavní možnosti: mobilní aplikace, webová aplikace a desktopová aplikace.

Hlavními kritérii pro výběr platformy byly především přístup k nativním funkcím zařízení, možnost použití aplikace mimo domov a kvalita uživatelského zážitku v mobilním prostředí.

Navrhovaná aplikace využívá funkce, jako jsou fotoaparát zařízení, galerie obrázků, zjišťování aktuální polohy uživatele a mapové zobrazení okolí. Aplikace je navržena tak, aby byla využitelná i mimo domov, například při cestování nebo návratu z cesty, kdy může uživatel rychle vyhledat obchody v okolí své aktuální polohy. Zároveň obchody v aplikaci poskytují zájemcům kontaktní údaje na producenty, aby je mohli snadno kontaktovat například telefonicky, přes SMS nebo pomocí odkazu na sociální sítě. Mobilní zařízení tak přirozeně podporuje jak samotné použití aplikace, tak i následnou komunikaci s producenty.

Desktopová aplikace s ohledem na uvedené požadavky neodpovídá charakteru navrhované aplikace. Výběr platformy se tak zužuje na nativní nebo multiplatformní mobilní aplikaci a webovou aplikaci optimalizovanou pro mobilní zařízení.

Webová aplikace by sice zajistila širší dostupnost napříč zařízeními, avšak v prostředí mobilních zařízení by ve srovnání s nativní mobilní aplikací poskytovala méně vhodný uživatelský zážitek, a to jak z hlediska výkonu a odezvy, tak i z hlediska přístupu k nativním funkcím zařízení [32, 51].

Další možností je využití multiplatformních frameworků, jako jsou Flutter nebo React Native. Tato řešení však přidávají další vrstvu abstrakce a mohou zvyšovat složitost integrace s nativními funkcemi

zařízení, jako jsou práce s mapami, polohou uživatele nebo fotoaparátem [16].

Na základě uvedených kritérií bylo zvoleno nativní řešení pro platformu Android, které nejlépe naplňuje požadavky na přímý přístup k nativním funkcím zařízení a kvalitní uživatelský zážitek v mobilním prostředí. Právě tyto požadavky se řadí mezi primární kritéria pro výběr platformy navrhované aplikace.

V České republice je navíc Android nejrozšířenějším operačním systémem pro mobilní zařízení, což dále podporuje vhodnost této volby [39].

2.5.2 Programovací jazyk a UI framework

Pro vývoj mobilní aplikace pro platformu Android bylo nutné zvolit vhodný programovací jazyk a framework pro tvorbu uživatelského rozhraní. Mezi hlavní kritéria výběru patřila podpora platformy Android, efektivita vývoje, čitelnost a udržovatelnost kódu a podpora moderních přístupů k vývoji uživatelského rozhraní.

V současnosti se pro tvorbu uživatelského rozhraní v Android aplikacích využívá tradiční View systém založený na XML, v kombinaci s jazyky Java nebo Kotlin, a moderní deklarativní framework Jetpack Compose, který je založen na jazyce Kotlin [45].

Jako programovací jazyk byl zvolen Kotlin, který je moderním a oficiálně doporučovaným jazykem pro vývoj Android aplikací. Ve srovnání s jazykem Java nabízí Kotlin kratší a čitelnější syntaxi, lepší podporu moderních programovacích paradigmat a vyšší bezpečnost typů díky práci s nulovatelností (null-safety) [7].

Pro tvorbu uživatelského rozhraní byl zvolen framework Jetpack Compose, který umožňuje deklarativní přístup k návrhu UI. Tento přístup snižuje množství potřebného kódu, usnadňuje jeho údržbu a zrychluje vývoj aplikace [45]. Jetpack Compose je navíc přirozeně integrován se správou stavu aplikace, což usnadňuje práci s dynamickým obsahem a uživatelskými interakcemi.

Na základě uvedených kritérií byla zvolena kombinace jazyka Kotlin a frameworku Jetpack Compose, která splňuje požadavky na efektivní a moderní přístup k vývoji uživatelského rozhraní a v současnosti představuje doporučovaný způsob vývoje nativních aplikací pro platformu Android [10].

2.5.3 Designový systém

Designový systém aplikace ovlivňuje vzhled, dojem z aplikace a jednoduchost jejího používání. Při výběru designového systému byly zohledněny zejména požadavky na konzistenci uživatelského rozhraní, efektivitu vývoje a podporu moderních designových principů.

Pro nativní mobilní aplikaci pro platformu Android se v praxi využívají většinou dvě hlavní možnosti: Material Design 3 nebo vlastní designový systém [20].

Vlastní designový systém by zahrnoval vytvoření veškerých vlastních komponent, barevného schématu a vzorů pro uživatelské rozhraní. Tento přístup je časově náročný a vyžaduje značné úsilí při návrhu i implementaci. Použití vlastního designového systému je vhodné především pro větší projekty nebo společnosti, které potřebují přizpůsobit vzhled aplikace své značce.

Další možností pro Android aplikace je využití Material Designu, což je designový systém vyvinutý společností Google. Pro vývoj nových aplikací je doporučováno využití nejnovější verze Material Design 3, která přináší moderní vzhled, lepší podporu pro tmavý režim a vestavěné komponenty a vzory pro návrh uživatelského rozhraní [38].

S ohledem na uvedené požadavky byl zvolen Material Design 3, který naplňuje požadavky na moderní a konzistentní vzhled aplikace a zároveň usnadňuje vývoj uživatelského rozhraní díky předpřipraveným komponentám.

2.5.4 Serverová část aplikace

Součástí návrhu bylo rozhodnutí o způsobu realizace serverové části aplikace, který by odpovídal rozsahu bakalářské práce, požadavkům aplikace a možnostem jednotlivce jako vývojáře. Hlavními kritérii pro výběr řešení serverové části byly především rychlost vývoje a nasazení, náročnost implementace a provozu, dostatečná škálovatelnost řešení a vhodnost pro navržený datový model aplikace. V úvahu připadalo využití řešení typu Backend-as-a-Service (BaaS), reprezentovaného například platformou Firebase, nebo implementace vlastního serveru.

BaaS je model cloudové služby, který poskytuje vývojářům předpřipravené služby backendu [57]. Mezi klíčové služby pro navrhovanou aplikaci poskytované platformou Firebase patří Firestore databáze

pro ukládání dat a řízení přístupu k datům, Firebase Authentication pro správu uživatelských účtů, Firebase Storage pro ukládání souborů, jako jsou fotografie obchodů a profilové obrázky uživatelů a Firebase Crashlytics pro sledování chyb a výkonu aplikace [27]. Firebase představuje plnohodnotné řešení serverové části aplikace typu **Backend-as-a-Service** (BaaS).

Platforma Firebase zároveň využívá principy **serverless architektury** [13]. Serverless architektura znamená, že vývojář nepracuje přímo s fyzickými nebo virtuálními servery, ale využívá cloudové služby, které automaticky spravují infrastrukturu a škálovatelnost. Poskytované služby se tak automaticky přizpůsobují aktuálnímu zatížení aplikace. Díky tomuto modelu nemusí vývojář řešit provozní aspekty serveru, jako je nasazení, škálovatelnost a údržba serveru. V případě, že je potřeba implementovat vlastní serverovou logiku, Firebase nabízí službu **Cloud Functions**, která umožňuje spouštět kód na straně backendu formou serverless funkcí v reakci na definované události.

Alternativním řešením by bylo vytvoření vlastního serveru například pomocí frameworku Spring [50]. Tento přístup by poskytoval vyšší míru kontroly nad architekturou a umožňoval implementaci komplexnější aplikační logiky. Na druhou stranu však vyžaduje větší implementační úsilí a řešení provozních aspektů, jako je nasazení, škálovatelnost, bezpečnost a údržba serveru nebo i náklady na jeho provoz. Navrhovaná aplikace však nevyžaduje komplexní serverovou logiku ani vysokou škálovatelnost, a proto by implementace vlastního serveru představovala nepřiměřenou implementační a provozní režii vzhledem k rozsahu a cílům práce.

Řešení typu BaaS umožňuje soustředit se na vývoj aplikace, aniž by bylo nutné řešit především provozní aspekty. Náklady na provoz se odvíjejí od počtu uživatelů a využití služeb, což je výhodné zejména pro aplikace s nižším počtem uživatelů.

Zároveň je to řešení, které je vhodné pro malé vývojové týmy, protože umožňuje rychlejší implementaci díky předpřipraveným službám [3]. S ohledem na realizaci aplikace jednotlivcem a rozsah práce je toto řešení vhodné.

Další výhodou použití Firebase je jeho snadná integrace s platformou Android a jazykem Kotlin, protože poskytuje oficiální SDK a knihovny pro tyto technologie [26].

Nevýhodou řešení typu BaaS může být závislost na konkrétním poskytovateli služby (vendor lock-in) a omezené možnosti implementace komplexní serverové logiky.

Při vhodném návrhu architektury lze tyto nevýhody minimalizovat. Architektura aplikace je navržena tak, aby nebyla pevně svázána s konkrétní implementací datových zdrojů pomocí abstrahovaných rozhraní. To umožňuje případnou budoucí náhradu zvoleného způsobu realizace serverové části aplikace jinou technologií, například vlastním serverem. Komplexní serverovou logiku lze v případě potřeby implementovat pomocí služby Cloud Functions. V navrhované aplikaci však taková logika není zásadní součástí systému, a proto je uvedené omezení akceptovatelné a nebrání realizaci navržené funkčnosti.

Firebase poskytuje jako databázové řešení dokumentově orientovanou NoSQL databázi **Cloud Firestore**. Tento typ databáze ukládá data ve formě dokumentů seskupených do kolekcí [12].

Dokument je datový záznam, který obsahuje páry klíč-hodnota [11]. Hodnoty mohou být jednoduché i složené datové typy (například pole nebo vnořené objekty). Kolekce jsou skupiny dokumentů, které sdílejí společný kontext (například kolekce uživatelů, kolekce obchodů, kolekce recenzí).

V navrhované aplikaci je datový model tvořen omezeným počtem hlavních entit, jako jsou obchody, uživatelé a recenze, které obsahují další vnořené nebo odvozené informace (například seznam kategorií, nabídku produktů, otevírací dobu nebo polohu obchodu). Firestore umožňuje ukládat tyto související informace přímo v rámci jednoho dokumentu nebo v podřízených kolekcích, což snižuje potřebu složitých spojení (joins), která jsou typická pro relační databáze.

Databáze je navržena pro práci v reálném čase a umožňuje efektivní synchronizaci dat mezi klientskou aplikací a backendem. Firestore SDK zároveň podporuje lokální offline cache dat na straně klienta a jejich automatickou synchronizaci při opětovném připojení k síti. Tato vlastnost je vhodná zejména pro mobilní aplikace, kde nelze vždy garantovat stabilní připojení k internetu.

Společně s bezpečnostními pravidly (security rules [28]) je možné zajistit řízení přístupu k datům na úrovni jednotlivých dokumentů podle přihlášeného uživatele.

Tato databázová technologie tak odpovídá požadavkům aplikace z hlediska struktury dat, výkonu, škálovatelnosti i vývoje.

Na základě uvedených kritérií pro serverovou část aplikace bylo zvoleno řešení BaaS, konkrétně platforma Firebase, které umožňuje efektivní implementaci navrženého datového modelu, nevyžaduje správu serverové infrastruktury a zároveň poskytuje dostatečnou škálovatelnost pro potřeby navrhované aplikace. Zvolené řešení tak odpovídá současnému rozsahu a cílům bakalářské práce i možnostem jednotlivce jako vývojáře, přičemž nevylučuje další rozvoj aplikace v případě zvýšených nároků na logiku na straně serveru nebo škálovatelnost.

2.5.5 Autentizace uživatelů

Autentizace uživatelů slouží k ověření identity uživatele, která je následně využita při řízení přístupu k datům aplikace. V rámci navrhovaného řešení je autentizace realizována pomocí služby Firebase Authentication, která je součástí zvoleného backendového řešení typu Backend-as-a-Service [27].

Mezi alternativní možnosti autentizace patří implementace vlastního autentizačního mechanismu nebo využití jiných spravovaných poskytovatelů identity, jako jsou například Auth0 nebo Amazon Cognito [9, 4].

Vlastní implementace by poskytovala vyšší míru kontroly nad autentizačním procesem, ale zároveň by vyžadovala návrh, implementaci a správu bezpečnostních mechanismů, jako je správa přihlašovacích údajů, generování autentizačních tokenů nebo ochrana proti neoprávněnému přístupu. Využití externích poskytovatelů identity naopak umožňuje delegovat tyto mechanismy na specializovanou službu, která poskytuje standardizované postupy autentizace a správy identity.

Zvolený přístup umožňuje využít existující infrastrukturu pro správu uživatelských účtů a bezpečné ověřování identity. Autentizační identita uživatele je v návrhu oddělena od doménového identifikátoru, což umožňuje nezávislost doménového modelu na konkrétní technologii autentizace. V případě potřeby je tak možné v budoucnu změnit způsob autentizace bez nutnosti zásadních změn v doménovém modelu a logice aplikace.

Detailní popis implementace autentizačního mechanismu a jeho propojení s řízením přístupu k datům je uveden v kapitole 4.4.1.

2.5.6 Mapové služby

Klíčovým prvkem navrhované aplikace je mapové zobrazení okolních obchodů. Hlavními kritérii pro výběr mapové služby byly míra přizpůsobitelnosti mapového zobrazení, výkon při práci s větším množstvím zobrazovaných prvků, uživatelský zážitek, kvalita dokumentace, náklady na provoz a náročnost integrace do prostředí Android aplikace. Rozhodování probíhalo mezi třemi hlavními možnostmi: Google Maps, Mapbox a OpenStreetMap.

Google Maps nabízí velmi rozsáhlou dokumentaci a snadnou integraci s platformou Android a frameworkem Jetpack Compose [35]. Omezením je však menší míra přizpůsobení mapového vzhledu a absence vestavěných nástrojů pro efektivní práci s větším množstvím zobrazovaných prvků bez využití externích nástrojů.

Mapbox představuje populární alternativu ke službě Google Maps. Poskytuje rovněž kvalitní dokumentaci a snadnou integraci s platformou Android a navíc nabízí vyšší míru přizpůsobitelnosti mapových stylů a podporu pro efektivní zobrazení většího množství prvků na mapě [34].

OpenStreetMap je open source projekt, který poskytuje volně dostupná mapová data. Jeho hlavní výhodou je otevřenost a absence licenčních poplatků, avšak ve srovnání s komerčními řešeními neposkytuje ucelené SDK pro platformu Android a jeho integrace je zpravidla složitější [2].

S ohledem na uvedená kritéria a zkušenosti získané v průběhu vývoje aplikace byla jako hlavní mapová služba zvolena platforma Mapbox, která poskytuje vyšší přizpůsobitelnost mapového vzhledu a lepší výkon při práci s větším množstvím zobrazovaných obchodů. Služba Google Maps je v aplikaci využita pouze okrajově pro vybrané funkce, kde byla její integrace jednodušší a plně dostačující.

3 Architektura kódu

Před zahájením implementace je důležité navrhnout a naplánovat architekturu kódu. Dobře navržená architektura pomůže ušetřit značné množství času a námahy v budoucnu, například při přidávání nových funkcí nebo opravování stávajícího kódu. Zároveň přispívá ke zvýšení jeho čitelnosti a udržovatelnosti. Naopak špatně navržená architektura může vést k problémům, jako je náročné zjišťování chyb a pomalejší vývoj.

Architektura aplikace byla navržena především s cílem jasně oddělit odpovědnosti jednotlivých částí kódu, čímž podporuje dosažení výše uvedených vlastností. Při návrhu byly převzaty vybrané principy běžně používané v moderních architektonických přístupech, které jsou dále rozvedeny v následujících podkapitolách.

V rámci této kapitoly je používán pojem *doména aplikace*, kterým je v kontextu této práce myšlena oblast reálného světa, kterou aplikace modeluje a jejíž pravidla a procesy realizuje. Doménová logika pak představuje část aplikační logiky odpovědnou za implementaci těchto pravidel, nezávisle na konkrétních technologiích či uživatelském rozhraní. Toto pojetí vychází z principů doménově řízeného návrhu [21].

3.1 Architektonický vzor MVVM

Jako základní architektonický vzor byla zvolena architektura Model-View-ViewModel (MVVM) [40, 29]. Jedná se o doporučený vzor pro vývoj nativních Android aplikací, který je podporován samotným Android frameworkem a prosazován společností Google, která stojí za vývojem platformy Android [45].

Alternativami k architektonickému vzoru MVVM jsou například Model-View-Presenter (MVP) nebo Model-View-Intent (MVI) [55]. U vzoru MVP Presenter přímo manipuluje s uživatelským rozhraním, což není plně v souladu deklarativním přístupem frameworku Jetpack Compose, který je využíván v této aplikaci. MVI je velmi dobře kompatibilní s deklarativním přístupem frameworku Jetpack Compose, avšak jeho použití bývá spojeno s vyšší komplexitou návrhu a správy stavu aplikace. Některé jeho principy (jako například jednosměrný

tok dat [45] nebo *Reducer pattern* [46]) jsou však využitelné i v rámci MVVM vzoru a přispívají k lepší předvídatelnosti toku dat v aplikaci a byly při implementaci také využity (viz. sekce 4.3.2).

V navrhované aplikaci je architektonický vzor MVVM využit k jasnému oddělení odpovědností mezi uživatelským rozhraním, logikou obrazovek a doménovou logikou. Vzor MVVM za tímto účelem rozděluje aplikaci do tří hlavních komponent: Model, View a ViewModel.

View představuje uživatelské rozhraní a je zodpovědné pouze za zobrazování dat a předávání uživatelských událostí. V této aplikaci je View realizováno pomocí frameworku Jetpack Compose a neobsahuje žádnou aplikační ani doménovou logiku.

Model reprezentuje data a doménovou logiku aplikace, včetně komunikace s repozitáři pro získávání dat. Podle MVVM nesmí být nijak závislý na uživatelském rozhraní ani jeho konkrétní implementaci a nesmí obsahovat logiku týkající se zobrazení. V navrhované aplikaci tuto roli plní doménová vrstva, jejíž struktura a principy jsou podrobně popsány v sekci 3.3.

ViewModel slouží jako prostředník mezi View a Modelem. Je zodpovědný za správu stavu uživatelského rozhraní, reakci na uživatelské události a komunikaci s Modelem za účelem získání nebo aktualizace dat.

ViewModel obsahuje aktuální stav uživatelského rozhraní a vystavuje ho ve formě, kterou může View snadno sledovat a reagovat na jeho změny. V navrhované aplikaci jsou pro správu stavu obrazovek využívány asynchronní datové toky, do kterých při změně dat ViewModel publikuje nový stav jakožto neměnný (immutable) objekt [8]. View pak tento stav pozoruje a na jeho základě vykresluje uživatelské rozhraní. View naopak předává uživatelské události (například kliknutí na tlačítko) ViewModelu, který na ně reaguje a případně aktualizuje stav nebo komunikuje s Modelem. Tento přístup je označován jako **jednosměrný tok dat** (unidirectional data flow) [45] a přispívá k lepší předvídatelnosti a stabilitě aplikace.

Ve ViewModelu se nachází pouze logika týkající se uživatelského rozhraní, ne však jeho samotné vykreslování. Doménová logika a správa dat jsou plně odděleny v Modelu. Stejně tak logika spojená s navigací mezi obrazovkami, zobrazení vyskakovacích oken či upozornění pro uživatele se nachází pouze ve View, protože je závislá na konkrétní implementaci uživatelského rozhraní.

ViewModely v této aplikaci nevykonávají žádné přímé akce nad uživatelským rozhraním, ale vystavují tzv. **vedlejší efekty** (side effects) [48]. Efekty jsou jednorázové události, které vyvolávají akce v uživatelském rozhraní, jako například navigace nebo zobrazení vyskakovacích oken. View vystavované efekty interpretuje a na jejich základě provádí příslušné akce v uživatelském rozhraní.

Jak bylo zmíněno výše, vzor MVVM je přímo podporovaný frameworkem Android. ViewModely jsou v Android frameworku tzv. lifecycle-aware, což znamená, že automaticky reagují na změny životního cyklu aplikace, například při změně konfigurace zařízení (otočení obrazovky) nebo při přechodu aplikace na pozadí. Díky tomu nedochází ke ztrátě stavu uživatelského rozhraní a k opakovanému načítání dat při těchto změnách [54].

Zvolený způsob využití architektonického vzoru MVVM se v praxi osvědčil jako přehledný a umožnil jasné oddělení uživatelského rozhraní od doménové logiky aplikace.

3.2 Struktura projektu podle funkčních celků

Projekt je strukturován podle funkčních celků (feature-based struktura), kdy jednotlivé části aplikace odpovídají konkrétním oblastem domény nebo uživatelské funkcionality [15, 36]. Tento přístup umožňuje jasné oddělení odpovědností, lepší orientaci v kódu a omezuje provázanost mezi jednotlivými částmi aplikace.

Každý celek je v projektu organizován jako samostatná jednotka a typicky obsahuje následující vrstvy:

- **domain** – doménové modely, use-case třídy (třídy případů užití) a rozhraní repozitářů (viz sekce 3.3),
- **data** – konkrétní implementace repozitářů a práce s datovými zdroji, mapování návratových hodnot do doménových výsledků (viz sekce 3.4),
- **di** – definice závislostí a jejich konfigurace pro daný funkční celek,
- **sample** – ukázková data využívaná při testování a ladění aplikace.

Aplikace je rozdělena do několika hlavních funkčních celků. Mezi nejdůležitější patří například:

- **auth** – registrace a přihlášení uživatelů,
- **shop** – práce s obchody, jejich vytváření, úprava a zobrazování,
- **review** – hodnocení a recenze obchodů,
- **user** – správa uživatelských údajů,
- **core** – logika jádra aplikace a architektonické abstrakce,
- **common** – společné pomocné třídy a funkce.

Celek **shop** je založen především na entitě *Shop* a s ní související doménovou logiku. Ta je realizovaná pomocí samostatných use-case tříd pro vytváření, úpravu a mazání obchodů a různé možnosti zobrazování obchodů (všechny obchody podle vzdálenosti, nejbližších X obchodů, obchody daného uživatele, ...). K tomu tento celek obsahuje i zmiňované rozhraní repozitářů (v doménové vrstvě) a jejich implementace (v datové vrstvě) a ukázková data (v balíčku *sample*) využívaná pro testování a náhled uživatelského rozhraní.

Celek **common** obsahuje sdílenou logiku, která není přímo spojená s konkrétní doménou, ale je využívána v různých částech aplikace. Jedná se převážně o jednoduché bezstavové pomocné funkce a třídy. Patří sem například třídy pro práci s barvou nebo logika pro formátování dat, validaci, porovnávání textů apod.

Celek **core** obsahuje logiku jádra aplikace, na které je závislá většina ostatních funkčních celků. Patří sem například abstrakce dat z repozitářů a doménových chyb operací (viz sekce 3.4) nebo obecná rozhraní využívaná napříč aplikací, jako je kontrakt pro perzistentní entity. Tento celek není závislý na žádném jiném funkčním celku aplikace. Většina ostatních funkčních celků je naopak závislá na tomto celku, čímž je zajištěno, že sdílená logika je centralizována a jednotná.

Uživatelské rozhraní je organizováno samostatně ve funkčním celku **app** v podsložce *ui*, kde jsou jednotlivé prvky rozděleny podle obrazovek nebo samostatných částí uživatelského rozhraní (komponent). Kromě toho složka ui obsahuje také podsložky **ui/common**, která obsahuje sdílené prvky uživatelského rozhraní (tlačítka, vstupní

pole apod.), a **ui/core**, která obsahuje základní komponenty a centrální logiku pro práci s uživatelským rozhraním (navigace a navigační lišta, vyvolávání vyskakovacích oken, informační lišty apod.). Funkční celek **app** je jako jediný celek přímo závislý na platformě Android, zatímco ostatní funkční celky jsou navrženy tak, aby jejich doménová logika nebyla pevně svázána s konkrétní platformou a mohla být při úpravách znovu použita i v jiných typech aplikací.

Jednotlivé funkční celky jsou v projektu realizovány jako samostatné Gradle moduly, což přispívá k lepší izolaci závislostí a zkrácení doby sestavení projektu.

3.3 Oddělení doménové a datové vrstvy

Jedním ze základních principů architektury aplikace je oddělení doménové a datové vrstvy [6, 21, 36]. Cílem tohoto rozdělení je zajistit, aby doménová logika aplikace nebyla závislá na konkrétní implementaci ukládání nebo získávání dat.

Doménová vrstva (domain layer) obsahuje veškerou logiku týkající se samotné domény aplikace. To zahrnuje definice doménových modelů, use-case třídy a rozhraní repozitářů.

Doménové modely reprezentují klíčové entity domény aplikace, jako jsou například uživatelé, obchody nebo recenze. Jedná se o čisté datové třídy bez závislostí na konkrétních technologiích nebo knihovnách a obsahují pouze atributy potřebné pro reprezentaci dané entity v kontextu domény aplikace.

Konkrétní doménová logika je implementována v use-case třídách, které představují hlavní vstupní bod pro její volání z ViewModelů. Jedná se často o komplexní nebo znovupoužitelnou logiku, jako je například validace dat, doménová pravidla, výpočty, koordinace více operací nad daty, nebo naopak jen jednoduché zprostředkování volání repozitářů.

Komunikace s backendem probíhá přes rozhraní repozitářů definovaných v doménové vrstvě. Rozhraní repozitářů byla umístěna do této vrstvy, protože tvoří hranici mezi doménovou logikou a datovými zdroji a výsledky jejich operací jsou orientovány na doménové modely a chyby.

Datová vrstva (data layer) je zodpovědná za konkrétní implementaci repozitářů a práci s externími datovými zdroji (cloudová databáze, autentizační služba apod.). Datová vrstva obsahuje datové modely, které reprezentují strukturu dat v konkrétních datových zdrojích. Datové modely se mohou lišit od doménových modelů, protože jsou přizpůsobeny specifickým požadavkům daného datového zdroje. V datové vrstvě tak probíhá získávání a ukládání dat, mapování datových modelů na doménové modely a naopak, a zpracování technických chyb vznikajících při komunikaci s datovými zdroji. Datová vrstva kromě základních operací zapouzdřuje také složitější způsoby práce s daty, které jsou závislé na konkrétním datovém zdroji, například stránkování výsledků nebo dotazy nad geografickými daty.

Díky definici rozhraní repozitářů v doménové vrstvě a jejich implementaci v datové vrstvě je závislost mezi těmito vrstvami jednosměrná a směřuje výhradně od datové vrstvy k doménové vrstvě. Toto rozdělení umožňuje snadnou výměnu nebo úpravu datových zdrojů bez nutnosti měnit doménovou logiku aplikace.

3.4 Zpracování výsledků a chyb v doménové vrstvě

V rámci návrhu architektury aplikace byl kladen důraz na jednotný a předvídatelný způsob práce s výsledky operací a chybovými stavy. Namísto používání výjimek napříč aplikační logikou bylo cílem reprezentovat výsledky volání, včetně chybových stavů, pomocí návratových hodnot. Díky tomu je zajištěno, že volající kód musí explicitně řešit všechny možné výsledky, včetně chyb, což přispívá k robustnější a předvídatelnější aplikaci.

Pro tento účel byl navržen typ **DomainResult**, který reprezentuje výsledek libovolné operace v doménové vrstvě. DomainResult může být buď úspěšný a obsahovat požadovaná data, nebo neúspěšný a obsahovat doménově orientovanou chybu (DomainError). Použití typu DomainResult je v této aplikaci jednotné napříč celou aplikační logikou. Rozhraní repozitářů i jednotlivé use-case třídy zpravidla vracejí výsledek operace ve formě DomainResult, čímž je zajištěno, že volající kód nemůže chybový stav ignorovat.

Chybové stavy jsou v aplikaci reprezentovány pomocí typu **DomainError**. Jedná se o hierarchii doménově orientovaných chyb, které

popisují význam chyby z pohledu aplikace, nikoli z pohledu konkrétní technologie nebo datového zdroje. Doménové chyby jsou členěny podle jednotlivých oblastí aplikace, například na chyby související s autentizací, správou obchodů nebo prací s fotografiemi. Tím je zajištěno, že chyby mají jasný význam v kontextu domény aplikace a mohou být jednoznačně interpretovány vyššími vrstvami aplikace, zejména ViewModely a uživatelským rozhraním.

V datové vrstvě jsou technické chyby vznikající při komunikaci s datovými zdroji zachyceny a převedeny na doménově orientované chyby (DomainError), které jsou následně vráceny ve formě neúspěšného DomainResult. Úspěšná volání jsou vrácena jako úspěšný DomainResult obsahující požadovaná data.

V doménové vrstvě se tyto výsledky dále zpracovávají, kombinují s výsledky jiných operací a slouží k řízení aplikačního toku na základě úspěšných i chybových stavů. Volajícímu kódu jsou data opět předány ve formě DomainResult, díky čemuž nedochází ke ztrátě informací o chybových stavech a je zajištěno jejich explicitní zpracování.

3.5 Tok dat v aplikaci

Tok dat v aplikaci je navržen tak, aby byl jednoznačný, předvídatelný a snadno sledovatelný napříč jednotlivými vrstvami aplikace. Aplikace využívá princip jednosměrného toku dat, kdy data proudí od uživatelského rozhraní směrem k doménové a datové vrstvě a výsledky operací jsou následně předávány zpět do uživatelského rozhraní.

Typický tok dat v aplikaci probíhá následujícím způsobem:

1. Uživatel vyvolá akci v uživatelském rozhraní, například kliknutím na tlačítko.

2. Uživatelská událost je předána ViewModelu, který zpracovává logiku dané obrazovky.

3. ViewModel na základě události zavolá odpovídající use-case třídu v doménové vrstvě.

4. Use-case třída provede doménovou logiku a v případě potřeby komunikuje s repozitářem.

5. Repozitář získá nebo uloží data prostřednictvím datové vrstvy a vrátí výsledek ve formě DomainResult.

6. Use-case třída zpracuje DomainResult a vrátí ho zpět ViewModelu.

7. ViewModel zpracuje výsledek operace a aktualizuje stav uživatelského rozhraní.

8. Uživatelské rozhraní automaticky reaguje na změnu stavu a znovu se vykreslí.

ViewModel vystavuje stav uživatelského rozhraní pomocí reaktivních datových toků, které uživatelské rozhraní průběžně pozoruje. Díky tomu není nutné explicitně řídit aktualizaci uživatelského rozhraní – změna stavu automaticky vede k jeho opětovnému vykreslení.

Stav uživatelského rozhraní je v aplikaci reprezentován neměnnými (immutable) datovými objekty [8]. Při změně stavu je vždy vytvořena nová instance stavu, nikoli modifikována stávající. Tento přístup zjednodušuje sledování změn, eliminuje vedlejší efekty a přispívá k lepší předvídatelnosti chování aplikace.

Kombinace jednosměrného toku dat, reaktivního zpracování stavu a explicitního zpracování chyb pomocí DomainResult vytváří konzistentní a snadno udržovatelný model toku dat napříč celou aplikací.

3.6 Vkládání závislostí

Pro řízení závislostí mezi jednotlivými částmi aplikace je využíván princip vkládání závislostí (Dependency Injection) [17]. Cílem tohoto přístupu je oddělit vytváření objektů od jejich používání, snížit provázanost jednotlivých tříd a zjednodušit testování aplikace.

V aplikaci je pro Dependency Injection použit framework Hilt, který je oficiálně doporučovaným řešením pro platformu Android [18]. Hilt poskytuje automatickou správu životního cyklu objektů, integraci s ViewModely a snižuje množství pomocného (boilerplate) kódu potřebného pro konfiguraci závislostí [19].

Pomocí Dependency Injection jsou do jednotlivých tříd vkládány jejich závislosti, například repozitáře do use-case tříd nebo use-case

třídy do ViewModelů. Díky tomu nejsou třídy přímo závislé na konkrétních implementacích svých závislostí, ale pouze na jejich rozhraních.

Závislosti jsou definovány v samostatných konfiguračních modulech, které určují, jaké konkrétní implementace určité funkcionality mají být v aplikaci použity. Tento přístup umožňuje snadnou výměnu implementací, například při testování nebo při změně způsobu práce s datovými zdroji.

Použití Dependency Injection v kombinaci s architekturou aplikace přispívá k lepší čitelnosti kódu, omezení provázanosti mezi vrstvami a jednodušší údržbě aplikace v budoucnu.

3.7 Shrnutí architektury

Navržená architektura aplikace vychází z požadavků kladených na přehlednost, rozšiřitelnost a dlouhodobou udržovatelnost kódu. Jednotlivé architektonické principy byly zvoleny s ohledem na charakter aplikace, rozsah bakalářské práce a použité technologie.

Použití architektonického vzoru MVVM umožnilo jasné oddělení uživatelského rozhraní, logiky obrazovek a doménové logiky aplikace. Struktura projektu podle funkčních celků přispěla k lepší orientaci v kódu a omezení provázanosti mezi jednotlivými funkčními celky. Oddělení doménové a datové vrstvy zajistilo nezávislost doménové logiky na konkrétní implementaci datových zdrojů. Abstrahování výsledků operací a chyb pomocí typu DomainResult přispělo k předvídatelnému toku dat napříč aplikací a vynutilo explicitní řešení chybových stavů.

Architektura aplikace nevznikla v konečné podobě od počátku vývoje, ale vyvíjela se postupně spolu s narůstající komplexitou systému. S přibývajícími funkčními celky a třídami se ukázalo, že jasné oddělení odpovědností a modularizace projektu jsou nezbytné pro udržení kontrolovatelnosti kódu. Zvolený návrh se v praxi osvědčil při implementaci i následném rozšiřování aplikace a poskytuje stabilní základ pro její další vývoj.

4 Implementace aplikace

Tato kapitola popisuje implementaci navržené architektury a klíčových částí aplikace. Vzhledem k rozsahu výsledného řešení se zaměřuje především na zásadní a pro tento systém specifické prvky, které mají největší význam z hlediska návrhu a struktury aplikace.

Pozornost je věnována zejména implementaci reprezentace výsledků operací, práci s geografickými dotazy nad databází a stránkování jejich výsledků, správě stavu uživatelského rozhraní a bezpečnostním mechanismům. Vybrané části ilustrují, jak se návrhová rozhodnutí popsaná v předchozí kapitole promítla do konkrétní realizace aplikace.

4.1 Implementace doménové vrstvy

Doménová vrstva představuje jádro aplikační logiky a její implementace vychází z návrhu popsaného v předchozí kapitole (3).

Obsahuje zejména doménové modely realizované jako neměnné datové třídy, jejichž atributy co nejvíce odpovídají skutečné doméně aplikace, use-case třídy (viz sekce 4.1.2), rozhraní repozitářů a typ `DomainResult` pro reprezentaci výsledků operací (viz sekce 4.1.1).

4.1.1 Reprezentace výsledků operací

Výsledky operací v doménové vrstvě jsou reprezentovány pomocí generického typu `DomainResult<T>`, implementovaného jako *sealed interface* s dvěma implementacemi: `Success<T>` a `Failure` (ukázka 4.1). Generický parametr `T` představuje typ úspěšného výsledku. Tento přístup je inspirován monadickým přístupem k reprezentaci výsledků operací, známým především z funkcionálního programování [56].

Implementace `Success<T>` obsahuje jediný atribut `data` typu `T`, který reprezentuje úspěšný výsledek operace. Naopak `Failure` obsahuje atribut `error` typu `DomainError`, který reprezentuje konkrétní doménově specifickou chybu, a volitelný atribut `cause` typu `Throwable`, který může nést původní výjimku z nižších vrstev (například pro logování chyb).

```
sealed interface DomainResult<out T> {

    data class Success<T>(val data: T) : DomainResult<T>

    data class Failure(
        val error: DomainError,
        val cause: Throwable? = null
    ) : DomainResult<Nothing>
}
```

Listing 4.1: Implementace typu DomainResult

Pro práci s výsledky operací jsou nad typem `DomainResult` definovány pomocné metody a funkce umožňující transformaci úspěšných výsledků, řetězení operací nebo reakci na úspěch či neúspěch pomocí předaných lambda funkcí (ukázka 4.2). Například funkce `mapSuccess` umožňuje transformovat úspěšný výsledek z typu `T` na jiný typ `R`, ale v případě neúspěchu zachovává původní chybu. Dále metoda `withSuccess` umožňuje provést zadanou akci pouze v případě úspěchu, zatímco `withFailure` umožňuje reagovat pouze na neúspěch.

```
inline fun <T, R> DomainResult<T>.mapSuccess(transform: (T) ->
    R): DomainResult<R> = when (this) {
    is Success -> Success(transform(data))
    is Failure -> this
}

suspend fun withSuccess(block: suspend (T) -> Unit):
   DomainResult<T> = apply {
    if (this is Success) block(this.data)
}
```

Listing 4.2: Pomocné metody a funkce pro práci s DomainResult

Chybové stavy jsou reprezentovány hierarchií typů implementujících rozhraní `DomainError`. Jednotlivé skupiny podtypů odpovídají konkrétním oblastem aplikace, například autentizaci uživatele, práci s obchody nebo práci s fotografiemi. Ve vrstvě uživatelského rozhraní jsou tyto chyby následně mapovány na lokalizované texty, které mohou

být zobrazeny uživateli, ať už jako součást komponent uživatelského rozhraní, nebo jako vyskakovací dialog.

4.1.2 Implementace use-case tříd

Use-case třídy představují základní prvek realizace doménové logiky. Každá use-case třída zapouzdřuje konkrétní operaci a řídí průběh a tok dat mezi repozitáři a dalšími závislostmi. Typicky zahrnují validaci vstupních dat, čtení nebo zápis do repozitářů a koordinaci dalších operací.

Use-case třídy jsou typicky implementovány jako třídy, které ve svém konstruktoru přijímají potřebné závislosti (repozitáře, další use-case třídy apod.) a obsahují jedinou veřejnou metodu `invoke`, která slouží jako vstupní bod pro vykonání operace. Koordinace jednotlivých kroků uvnitř use-case třídy je často řízena pomocí výsledků typu `DomainResult` a jeho pomocných funkcí.

Vytváření těchto tříd je řízeno pomocí mechanismu vkládání závislostí (viz sekce 3.6 minulé kapitoly).

Ukázka 4.3 demonstruje implementaci use-case třídy `CreateShopUC`, která zajišťuje vytvoření nového obchodu. Tato třída přijímá jako závislosti use-case třídu pro získání přihlášeného uživatele, repozitář pro práci s obchody a službu pro ukládání fotografií. Use-case nejprve získá přihlášeného uživatele, poté převede vstupní data na doménový model obchodu, uloží obchod do repozitáře, nahraje fotografie do úložiště a nakonec aktualizuje obchod o odkazy na nově uložené fotografie. Při selhání kterékoliv z těchto operací (návrat neúspěšného `DomainResult`) je metoda předčasně ukončena a aktuální chyba je okamžitě vrácena volajícímu pomocí funkce `getOrReturn`. Při úspěchu však `getOrReturn` vrací pouze vnitřní data objektu typu `DomainResult`, která jsou následně použita pro další operace.

```
class CreateShopUC @Inject constructor(
    private val getLoggedInUser: GetLoggedInUserUC,
    private val shopRepository: ShopRepository,
    private val photoStorage: PhotoStorage,
) {
    suspend operator fun invoke(input: ShopInput):
    DomainResult<Unit> {
        val user = getLoggedInUser.sync()
            .getOrReturn { return it }

        val shop: Shop = input
            .toShop(ShopId.newId(), user.id)
            .getOrReturn { return it }

        shopRepository.create(shop)
            .getOrReturn { return it }

        val shopWithPhotos = shop.updatePhotos(shop.id)
            .getOrReturn { return it }

        return shopRepository.update(shopWithPhotos)
    }
}
```

Listing 4.3: Ukázka implementace use-case třídy

4.2 Implementace datové vrstvy

Datová vrstva zajišťuje komunikaci aplikace s externími datovými zdroji. Mezi hlavní datové zdroje patří databáze Cloud Firestore, úložiště souborů Firebase Storage a autentizační služba Firebase Authentication.

Implementace datové vrstvy zahrnuje především mapování datových entit na doménové modely, zpracování chyb vznikajících při komunikaci s datovými zdroji a realizaci konkrétních operací nad těmito zdroji.

4.2.1 Mapování datových modelů a zpracování chyb

Datová vrstva pracuje s datovými entitami, jejichž struktura odpovídá způsobu uložení dat v konkrétním datovém zdroji (například dokumentům v databázi Cloud Firestore). Tyto entity jsou po načtení převáděny na doménové modely používané v doménové vrstvě aplikace. Při zápisu dat do datového zdroje jsou naopak doménové modely převáděny na odpovídající datové entity.

Při komunikaci s datovými zdroji mohou vznikat různé chyby, například síťové problémy, nedostatečná oprávnění nebo neexistující data. Tyto chyby jsou v datové vrstvě zachyceny a převedeny na konkrétní typy implementující rozhraní `DomainError`. Například při vytváření obchodu může dojít k chybě způsobené nedostatečnými oprávněními. Taková chyba je zachycena a převedena například na typ `ShopError.CreationFailed`. Tato hodnota je následně vrácena do doménové vrstvy jako atribut neúspěšného `DomainResult`. V případě úspěchu je vrácen úspěšný `DomainResult` obsahující relevantní data (například ID nově vytvořeného obchodu).

4.2.2 Geografické dotazy a stránkování

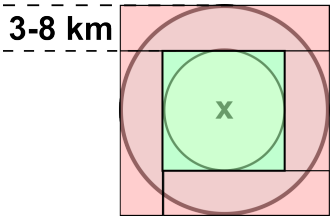
Aplikace umožňuje zobrazovat obchody v okolí aktuální polohy uživatele. V aplikaci jsou dvě možná zobrazení: zobrazení na mapě a zobrazení jako seznam. Pokud by bylo v databázi velké množství obchodů, bylo by neefektivní načítat všechny obchody zároveň, a proto bylo nutné implementovat stránkování výsledků. V mapovém i seznamovém zobrazení je přitom žádoucí, aby se obchody nacházející se blíže k uživateli načítaly a zobrazovaly dříve než vzdálenější obchody. Je tedy potřeba implementovat stránkování, které bude načítat obchody postupně podle přibližné vzdálenosti od uživatele.

Stránkování je přitom nutné realizovat na straně databáze, aby se minimalizovalo množství dat přenášených přes síť. Cloud Firestore však neposkytuje přímou podporu pro geografické dotazy v závislosti na vzdálenosti od určitého bodu [30].

Pro realizaci geografického vyhledávání je proto použita technika založená na tzv. *geohash* [30]. Geohash představuje způsob kódování geografické polohy do řetězce znaků, který zachovává prostorovou blízkost bodů. Body nacházející se blízko sebe mají podobný geo-

hashový prefix, což umožňuje provádět prostorové dotazy pomocí rozsahových dotazů nad tímto polem.

Pro rozdělení obchodů do stránek jsou definovány tzv. *prstence* (*DistanceRings*), které obsahují maximální a minimální vzdálenost od uživatele. Na základě těchto hodnot se generuje množina rozsahů geohashových hodnot pro daný prstenec (ukázka 4.1).



- **x** – aktuální poloha uživatele
- **zelený (vnitřní) obdélník** – rozsah prvního prstence (do 3 km)
- **červené (vnější) čtyři obdélníky** – rozsahy druhého prstence (3–8 km)

Obrázek 4.1: Ilustrační příklad rozsahů generovaných pro jednotlivé prstence

Rozsah geohashových hodnot je definován počáteční a koncovou geohashovou hodnotou. Tento rozsah tedy popisuje obdélník v geografickém prostoru. Hodnoty geohashů polohy jednotlivých obchodů mohou být s tímto rozsahem porovnány pomocí lexikografického porovnání řetězců (operace větší/menší), čímž lze určit, zda do daného rozsahu, a tím pádem i příslušného prstence, spadají.

Jelikož obdélníky definované množinou rozsahů geohashových hodnot z geometrického hlediska neodpovídají přesně prstencové oblasti definované vzdáleností od uživatele, není seřazení obchodů podle vzdálenosti od uživatele naprosto přesné, ale obsahuje mírnou odchylku (jak je popsáno v dokumentaci implementované funkce pro stránkované načítání obchodů). Například obchod, který spadá do prstence A (do 3 km), může být ve skutečnosti vzdálenější od uživatele než obchod, který spadá do prstence B (3–8 km), pokud se nachází v rohu obdélníku definovaného rozsahy geohashů pro prstenec A.

Tato částečná nepřesnost však není problém, protože řešení splňuje požadavek, aby se obchody nacházející se blíže k uživateli načítaly dříve než obchody výrazně vzdálenější. Zároveň toto řešení zajišťuje plně **deterministické rozdělení obchodů** do daných prstenců. Každý obchod tedy svou polohou patří do právě jednoho prstence a díky tomu při načítání dat nevznikají žádné duplicity ani vynechané záznamy, což je zásadní pro správné fungování stránkování.

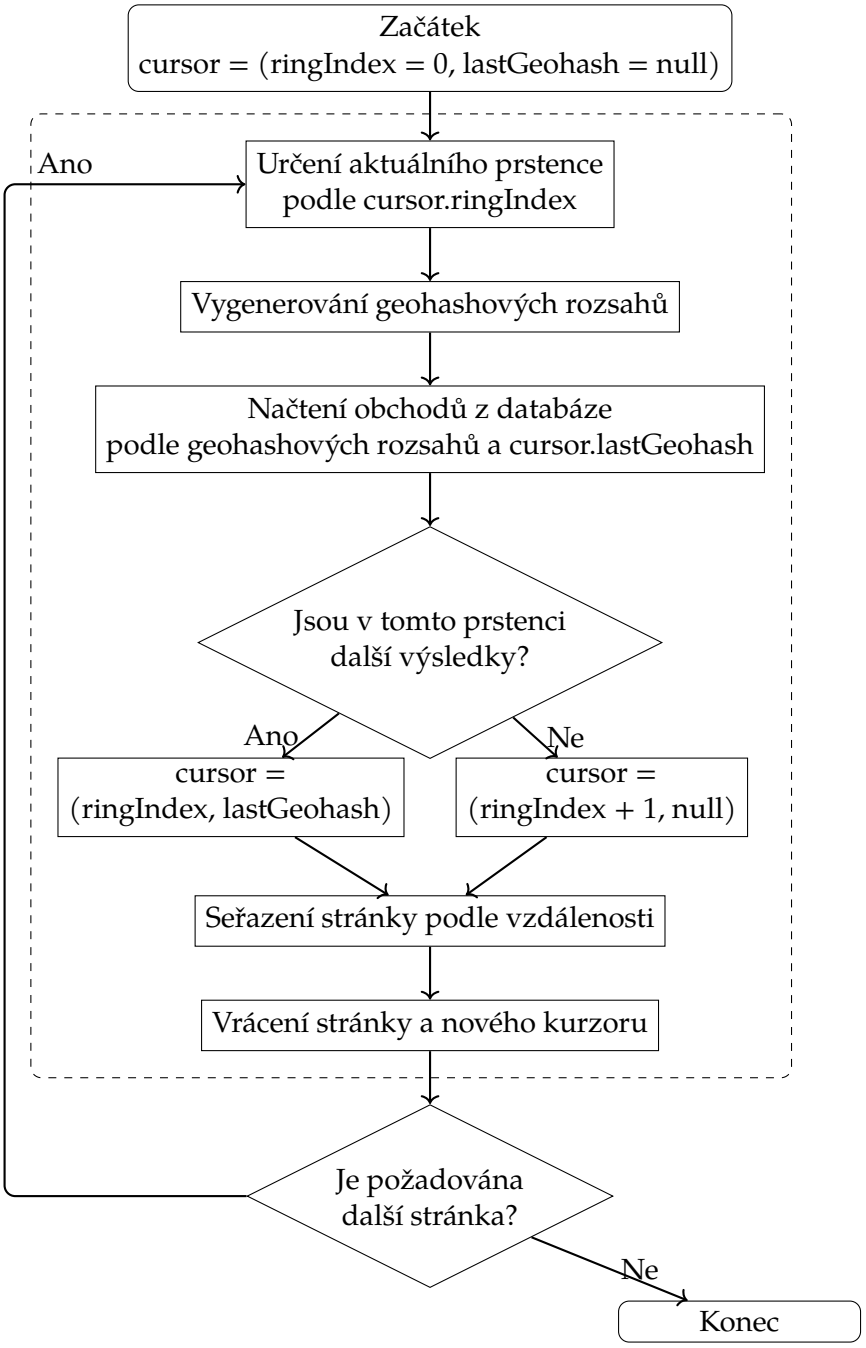
Po načtení stránky obchodů z databáze do aplikace jsou jednotlivé obchody v rámci této stránky lokálně seřazeny podle skutečné geografické vzdálenosti od uživatele. Efektivní řazení obchodů napříč jednotlivými stránkami však není možné, protože by vyžadovalo načtení všech obchodů najednou, což by vedlo ke značné neefektivitě a nadměrnému přenosu dat.

Při načítání jednotlivých stránek obchodů se využívá kurzor (*cursor-based paging*), který umožňuje načítat další stránku dat od místa, kde skončilo načítání předchozí stránky [42]. Stránka je z praktických důvodů omezena jak prstencem, tak maximálním počtem záznamů, aby se nenačítalo nekontrolovatelné množství dat v případě, že v oblasti definované prstencem existuje velké množství obchodů. Kurzor tak obsahuje jak informaci o tom, ze kterého prstence máme načítat, tak informaci o posledním načteném geohashi.

V případě, že načteme maximální počet obchodů z prstence A (kvůli omezení maximálního počtu záznamů) a zjistíme, že v prstenci A je ještě více obchodů k načtení, uložíme do kurzoru geohash posledního načteného obchodu a pokračujeme v načítání další stránky stále z prstence A, ale vynecháme obchody s hodnotou geohashe menší než je geohash uložený v kurzoru (již načtené obchody z aktuálního prstence). Teprve až načteme všechny obchody z prstence A, nastavíme v kurzoru prstenec B a hodnotu posledního načteného geohashe na `null` a tím načítáme z následujícího prstence B.

Celkový postup stránkování obchodů pomocí prstenců a kurzoru je znázorněn na obrázku 4.2.

Navržené řešení bylo ověřeno na přibližně 500 vygenerovaných testovacích záznamech obchodů. Z hlediska výkonu i uživatelského vnímání v rámci uživatelského testování se řešení ukázalo jako zcela dostatečné – načítání probíhalo plynule a přibližná přesnost řazení byla pro účely aplikace vyhovující.



Obrázek 4.2: Princip stránkování obchodů podle vzdálenosti pomocí geohashových prstenců a kurzorového stránkování

4.2.3 Kategorie obchodů v NoSQL databázi

Databáze Cloud Firestore je dokumentově orientovaná NoSQL databáze, která na rozdíl od relačních databází neumožňuje provádět složitější agregace nad daty uloženými ve více dokumentech [59]. V aplikaci je však potřeba získat četnost názvů kategorií napříč všemi obchody pro zobrazení nejpopulárnějších kategorií obchodů uživateli.

Pro tento účel je vytvořena samostatná kolekce `categorypopularity`, která obsahuje dokumenty reprezentující jednotlivé kategorie a jejich četnost. Jednotlivé dokumenty obsahují pouze název kategorie jako ID dokumentu a její četnost.

Zde využíváme službu Firebase Cloud Functions, která umožňuje spouštět serverový kód v reakci na události v databázi (popsáno v sekci 2.5.4). Při vytvoření nebo aktualizaci jakéhokoliv obchodu se na straně serveru spustí funkce, která porovná obchod před a po aktualizaci a získá přidané a odebrané kategorie. Následně aktualizuje odpovídajícím dokumentům v kolekci `categorypopularity` jejich četnost (zvýší o 1 pro přidané kategorie, sníží o 1 pro odebrané kategorie), případně vytvoří nový dokument pro novou kategorii.

Funkce je spouštěna pouze na straně serveru, aby se zabránilo neautorizovaným změnám a souběhům při aktualizaci četností kategorií. Samotná operace zvýšení nebo snížení četnosti je atomická, aby byla zajištěna konzistence dat.

Četnosti se vztahují pouze k názvům kategorií, protože kategorie nejsou globální entitou, ale pouze atributem obchodu. Před aktualizací četnosti se název kategorie normalizuje – například se odstraní mezery na začátku a konci řetězce nebo se počáteční písmeno převede na velké a zbytek písmen na malá.

Tyto četnosti kategorií jsou využívány pro nabízení nejpopulárnějších kategorií obchodů při vytváření nebo úpravě obchodu a při nastavování filtrování pro zobrazení obchodů jak v mapovém, tak v seznamovém zobrazení.

4.3 Implementace uživatelského rozhraní

Uživatelské rozhraní zajišťuje interakci uživatele s aplikací. Je implementováno pomocí frameworku Jetpack Compose, který umožňuje

deklarativní definici uživatelského rozhraní a reaktivní aktualizaci zobrazení na základě změn stavu.

Tato kapitola se zaměřuje především na strukturu navigace v aplikaci a na správu stavu a vedlejších efektů v rámci uživatelského rozhraní. Tyto prvky jsou klíčové zejména pro aplikace s větším počtem obrazovek a složitějšími uživatelskými interakcemi.

4.3.1 Navigační graf aplikace

Navigace v aplikaci je implementována pomocí navigační knihovny Jetpack Compose Navigation [41]. Pro zajištění bezpečnosti a přehlednosti je každá destinace v navigačním grafu reprezentována jako konkrétní datová třída, která implementuje společné rozhraní `Route` pro zajištění typové bezpečnosti.

Navigační graf je strukturován hierarchicky, přičemž obsahuje čtyři hlavní vnořené podgrafy: *auth*, *profile*, *home* a *myshops*.

Každý z těchto podgrafů obsahuje destinace, které jsou relevantní pro danou oblast aplikace. Podgraf *auth* obsahuje destinace pro přihlašování a registraci a je přístupný pouze pro nepřihlášené uživatele. Podgraf *profile* obsahuje destinace pro zobrazení a úpravu profilu a je přístupný pouze pro přihlášené uživatele. Podgraf *home* obsahuje jedinou destinaci pro zobrazení mapy a seznamu obchodů a podgraf *myshops* obsahuje destinaci pro zobrazení seznamu obchodů vytvořených přihlášeným uživatelem. Mezi těmito podgrafy je v uživatelském rozhraní možné přecházet pomocí spodní navigace, která byla již popsána v sekci 2.4.2. Při přepínání mezi podgrafy je zachován jejich navigační stav, což umožňuje uživateli plynule pokračovat v práci bez ztráty kontextu.

Mimo tyto podgrafy obsahuje navigační graf destinace, které nespadají pod žádný z těchto podgrafů, například destinaci pro zobrazení detailu obchodu. Tyto destinace jsou globální a mohou být navigovány z různých částí aplikace.

4.3.2 Správa stavu a vedlejší efekty

Ve frameworku Jetpack Compose je uživatelské rozhraní definováno jako funkce, která na základě vstupního stavu deklaruje výslednou podobu UI [52]. Tyto funkce jsou zpravidla navrženy jako čisté a

neobsahují vedlejší efekty. Vedlejší efekty, jako jsou navigace nebo zobrazení dialogů, jsou řešeny mimo samotnou definici UI.

Stav uživatelského rozhraní je reprezentován jako neměnná datová třída, která obsahuje všechny informace potřebné pro vykreslení dané komponenty. Tento stav je předáván jako vstupní parametr do UI funkce a podoba komponenty uživatelského rozhraní je deklarativně odvozena z tohoto stavu. Aktualizace uživatelského rozhraní se provádí automaticky na základě změny tohoto stavu, což vyvolá rekompozici dané komponenty s novými daty.

ViewModel spravuje stav zobrazení, zpracovává uživatelské události a komunikuje s doménovou vrstvou prostřednictvím use-case tříd. Stav zobrazení je ve ViewModelu vystaven pomocí reaktivního datového toku `StateFlow`, který uchovává aktuální hodnotu stavu a umožňuje pozorování jeho změn. Hodnoty tohoto toku dat jsou pozorovány v UI, které se při každé změně stavu automaticky aktualizuje.

Interakce uživatele s uživatelským rozhraním (například kliknutí na tlačítka nebo zadávání textu) jsou reprezentovány jako události (`Event`). Ty jsou pomocí volání funkcí předávány z UI do ViewModelu, který je zpracovává a na základě nich aktualizuje stav zobrazení. Aktualizace stavu je provedena pomocí vytvoření nové instance stavu vycházející pouze z předchozího stavu a přijaté události. Tento přístup je inspirován vzorem *Reducer pattern* [46].

V reakci na zpracování události mohou také vznikat vedlejší efekty, které jsou od samotného stavu odděleny a reprezentovány samostatným datovým tokem. Tím je zajištěno, že stav zůstává čistou reprezentací UI. Vedlejší efekty jsou zpracovávány jednorázově ve vrstvě propojující ViewModel a samotné UI komponenty. Mezi vedlejší efekty patří například navigace nebo zobrazení zpráv uživateli.

Ukázka A.1 demonstruje implementaci ViewModelu pro vytvoření obchodu a implementaci správy stavu, vedlejších efektů a zpracování událostí v tomto ViewModelu.

4.4 Bezpečnostní model a řízení přístupu

Bezpečnost aplikace je zajištěna kombinací autentizačních mechanismů a pravidel řízení přístupu definovaných na úrovni serveru.

4.4.1 Autentizace uživatelů

Autentizace uživatelů je realizována pomocí služby Firebase Authentication, která poskytuje různé metody přihlášení (například e-mail a Google účet). Po úspěšném přihlášení získává aplikace unikátní identifikátor uživatele (`uid`), který představuje identitu uživatele v rámci autentizační služby. Tento identifikátor je uložen v dokumentu uživatele v databázi jako samostatný atribut z důvodu propojení autentizační identity s doménovým identifikátorem uživatele. Doménový identifikátor uživatele je reprezentován nezávislým atributem (`id`).

Samotná autentizace řeší pouze ověření identity uživatele. Oprávnění k přístupu k datům jsou následně řízena pomocí bezpečnostních pravidel databáze popsaných v následující sekci.

4.4.2 Firebase Security Rules

Řízení přístupu k datům je realizováno pomocí bezpečnostních pravidel databáze Cloud Firestore, která jsou vyhodnocována na straně serveru při každém požadavku [28]. Tím je zajištěno, že omezení přístupu nelze obejít úpravou klientské aplikace nebo zachycením a úpravou síťové komunikace.

Pro každou kolekci v databázi jsou nastavena zvlášť pravidla pro vytváření, čtení, aktualizaci a mazání dokumentů. Operace čtení je pro většinu kolekcí povolena bez omezení, protože se nejedná o citlivá data a v aplikaci si je mohou zobrazit i nepřihlášení uživatelé.

Pro **operaci vytvoření** jsou pro většinu kolekcí nastavena pravidla, která kontrolují, zda je uživatel autentizován (přihlášen) a zda jsou data ve správném formátu. Navíc pro dokumenty, které obsahují odkaz na uživatele jako svého vlastníka, například formou atributu `ownerId`, je kontrolováno, zda tento atribut odpovídá identifikátoru přihlášeného uživatele. Tím je zajištěno, že uživatel nemůže vytvořit dokument, který by patřil jinému uživateli. Pro kolekci `review` je navíc přidáno pravidlo, které zakazuje uživatelům vytvářet recenze pro své vlastní obchody, aby se zabránilo zneužívání systému recenzí.

V požadavku zaslaném klientskou aplikací se nachází pouze autentizační identifikátor uživatele (`uid`) (popsáno v předchozí sekci 4.4.1), který je oddělen od doménového identifikátoru uživatele (`id`). Pro **porovnání vlastnictví dokumentu**, které je nejčastěji reprezentováno

například atributem `ownerId`, je tedy potřeba provést mapování mezi těmito dvěma identifikátory. V rámci bezpečnostních pravidel je proto implementována funkce, která na základě doménového identifikátoru uživatele získá odpovídající autentizační identifikátor (`uid`) z kolekce uživatelů v databázi a porovná jej s identifikátorem přihlášeného uživatele v požadavku. Tímto způsobem se ověří, zda je vlastníkem dokumentu skutečně přihlášený uživatel, aniž by bylo nutné ukládat v dokumentu autentizační identifikátor, který je závislý na implementaci autentizační služby. Díky tomu je možné v případě potřeby změnit autentizační službu bez nutnosti upravovat strukturu dat v databázi.

Operace aktualizace dokumentů je pro většinu kolekcí povolena pouze pro vlastníka dokumentu, přičemž je kontrolováno, že vlastník dokumentu zůstává nezměněn. Stejně jako u pravidel pro vytváření dokumentů je kontrolováno, že aktualizovaná data jsou ve správném formátu a uživatel je autentizován. Pro kolekci `review` není operace aktualizace implementována, protože úprava recenze není v aplikaci povolena.

Operace mazání dokumentů je opět pro většinu kolekcí povolena pouze pro vlastníka dokumentu, přičemž je kontrolováno, zda je uživatel autentizován.

Speciálním případem jsou pravidla pro kolekci `categorypopularity`, kde jsou všechny operace kromě čtení zakázány, protože tyto dokumenty jsou vytvářeny na straně serveru pomocí Firebase Cloud Functions, nikoliv požadavky z klientské aplikace.

Kolekce recenzí je v databázi strukturována jako podkolekce uvnitř každého dokumentu obchodu. Jednotlivé dokumenty recenzí jsou následně v této podkolekci identifikovány pomocí ID uživatele, který recenzi vytvořil. Tímto je zajištěno pravidlo, že každý uživatel může vytvořit pouze jednu recenzi pro každý obchod. Při pokusu o vytvoření další recenze pro stejný obchod bude požadavek zamítnut, protože dokument s daným ID již existuje.

Bezpečnostní pravidla databáze jsou demonstrována v ukázce A.2.

Pro ukládání souborů, jako jsou fotografie, je využívána služba Firebase Storage, která má vlastní bezpečnostní pravidla fungující na stejném principu jako pravidla pro databázi. U každé kolekce se pro operace zápisu kontroluje autentizace uživatele a vlastnictví entity, ke které se soubor vztahuje (například obchod, ke kterému fotografie patří).

4.5 Shrnutí implementační části

Výsledná implementace je rozdělena do osmi funkčních modulů (deseti včetně sdílených modulů *core* a *common*), které odpovídají jednotlivým částem domény a dodržují principy architektury navržené v předchozí kapitole.

Aplikace obsahuje přibližně 15 000 řádků zdrojového kódu ve více než 300 souborech. Během implementace se ukázalo, že navržená architektura umožnila lepší organizaci kódu a efektivní implementaci jednotlivých funkcionalit i přes narůstající velikost projektu.

Napříč datovou a doménovou vrstvou se při implementaci velmi osvědčila reprezentace výsledků operací a chyb pomocí typu `DomainResult`. To přispělo k vyšší robustnosti aplikace a výrazně omezilo výskyt neočekávaných chyb způsobených zapomenutím zpracování chybových stavů.

Ve vrstvě uživatelského rozhraní se ukázalo, že oddělení stavu zobrazení a vedlejších efektů ve ViewModelu umožnilo čistší a přehlednější implementaci UI. Explicitní reprezentace událostí, například vstupů od uživatele, a inspirace vzorem *Reducer pattern* v některých ViewModelech přispěly k lepší přehlednosti kódu a jasnějšímu oddělení jednotlivých odpovědností.

5 Testování aplikace

Testování aplikace bylo zaměřeno na ověření správnosti implementace, stability aplikace a odhalení chyb vznikajících při běžném i méně obvyklém použití.

V průběhu vývoje byly využity různé přístupy k testování, od automatizovaných jednotkových testů přes manuální testování až po sledování chyb v reálném provozu pomocí nástrojů třetích stran.

5.1 Jednotkové testy

Pro ověření správnosti aplikační logiky byly využity jednotkové testy zaměřené především na doménovou vrstvu, zejména use-case třídy.

Testy byly navrženy tak, aby ověřovaly chování jednotlivých use-case tříd nezávisle na konkrétní implementaci datových zdrojů. Díky oddělení doménové a datové vrstvy bylo možné testovat use-case třídy pomocí mockovaných repozitářů, což umožnilo simulovat různé scénáře, včetně chybových stavů.

Bylo implementováno více než 80 jednotkových testů, které pokrývají různé scénáře použití a chybové stavy. Největší část testů je zaměřena na funkční celky `core`, `common` a `shop`, které obsahují klíčové funkce aplikace.

Testy pomohly odhalit například chyby v mapování doménových chyb, kdy use-case třídy po některých úpravách vracely nepřesné nebo příliš obecné `DomainError` chyby (například obecnou chybu pro práci s obchody namísto chyby chybějící autentizace uživatele).

5.2 Firebase crashlytics

Pro sledování chyb v reálném provozu aplikace je využívána služba Firebase Crashlytics, která umožňuje automatické zaznamenávání pádů aplikace.

Crashlytics poskytuje detailní informace o vzniklých chybách, jako jsou informace o zařízení, verze aplikace nebo konkrétní zdroj chyby ve zdrojovém kódu, což umožňuje rychle identifikovat příčinu problému a následně jej opravit.

Tento nástroj je určen především pro použití v produkčním prostředí, kde umožňuje sledovat chyby vznikající u koncových uživatelů.

5.3 Manuální testování

Manuální testování bylo zaměřeno na ověření funkčnosti aplikace z pohledu běžného uživatele a na odhalení chyb, které se nemusí projevit při jednotkovém testování.

Testování probíhalo formou procházení jednotlivých scénářů použití aplikace, například registrace a přihlášení uživatele, vytváření a úprava obchodů, prohlížení obchodů na mapě a práce s recenzemi.

Během tohoto testování bylo odhaleno několik chyb a nedostatků v chování aplikace. Například při přechodu zpět po úspěšném přihlášení docházelo k návratu na přihlašovací obrazovku, přestože je již uživatel přihlášen. Tento problém byl způsoben nevhodnou strukturou navigačního grafu a byl opraven rozdělením části navigace do dvou podgrafů: autentizačního a profilového (tato finální podoba je již popsána v kapitole implementace v sekci 4.3.1). Jakmile je tedy uživatel po přihlášení přesměrován do profilového podgrafu, nemůže se již systémovou navigací zpět vrátit do autentizačního podgrafu.

Dalším nalezeným problémem bylo neočekávané resetování mapy do výchozího stavu. K tomu docházelo po přechodu do recenzí obchodu ze spodního dialogu v mapovém zobrazení obchodů a následném návratu zpět. Tento problém byl vyřešen uložením aktuálně vybraného obchodu a jeho automatickým znovuzvolením při návratu zpět z obrazovky recenzí daného obchodu.

Na základě manuálního testování byly také přidány potvrzovací dialogy pro destruktivní operace, jako je například smazání nebo úprava obchodu. Tyto akce bylo možné původně provést jedním kliknutím na tlačítko, což zvyšovalo pravděpodobnost nechtěné akce. Po úpravě se před provedením takové akce zobrazí potvrzovací dialog, který umožňuje potvrdit nebo zrušit danou akci.

Manuální testování tak přispělo nejen k odhalení chyb, ale i ke zlepšení použitelnosti aplikace z pohledu koncového uživatele.

5.4 Uživatelské testování

Uživatelské testování bylo zaměřeno na ověření použitelnosti uživatelského rozhraní a identifikaci nedostatků, které se projeví při používání aplikace uživateli, kteří nejsou obeznámeni s jejím vnitřním fungováním.

Testování probíhalo formou plnění předem definovaných scénářů, které simulují běžné činnosti uživatele v aplikaci. Testování se účastnilo celkem 8 uživatelů různé technické zdatnosti, od běžných uživatelů mobilních aplikací až po technicky zaměřené uživatele.

Mezi tyto scénáře patřilo vyhledání obchodů podle vzdálenosti, kategorie a hodnocení za pomoci filtrování, vytvoření nového obchodu včetně přidání produktů a fotografií, úprava dostupnosti produktů, úprava uživatelského profilu nebo vytvoření recenze. Podrobné zadání těchto scénářů je uvedeno v příloze B.

Všichni uživatelé prošli scénáři úspěšně, lišila se pouze rychlost provedení jednotlivých kroků. V reakci na tento průběh byly přidány dodatečné popisky pro některá vstupní pole, oznámení pro právě provedené úspěšné i neúspěšné akce (formou upozornění typu *toast message* [53]), zvýrazněna tlačítka pro důležité operace a adekvátně upraveno chování některých prvků uživatelského rozhraní.

Jedním z návrhů od uživatelů bylo například přidání navigační lišty pro jednotlivé kroky procesu vytváření a úpravy obchodu. To by urychlilo menší běžné úpravy obchodu, jako je například změna dostupnosti produktu, protože by uživatelé nemuseli procházet všechny kroky a hledat, ve kterém kroku se dané nastavení nachází.

Dále bylo navrženo například přidání ukazatele aktuálně aktivních filtrů, aby bylo pro uživatele jasnější, zda se ukazují všechny obchody nebo pouze ty odpovídající nastaveným filtrům.

Tyto i další změny budou postupně implementovány v dalších aktualizacích aplikace pro zlepšení použitelnosti a uživatelského zážitku.

Uživatelské testování tak přispělo především k odhalení nedostatků v použitelnosti pro běžné uživatele a jejich následné odstranění.

6 Vydání a nasazení aplikace

Před vydáním aplikace bylo nutné provést její přípravu pro produkční prostředí. Ta zahrnovala zejména nahrazení testovacích API klíčů produkčními a vytvoření produkčního sestavení aplikace, které zahrnuje optimalizovaný kód a omezení ladicích nástrojů. Součástí procesu sestavení je také podepsání aplikace pomocí privátního release klíče, což je nezbytné pro její distribuci.

Aplikace byla publikována prostřednictvím platformy Google Play, která umožňuje její distribuci koncovým uživatelům. Publikace zahrnovala vytvoření záznamu aplikace na Google Play, který obsahuje informace o aplikaci, jako je její název, popis, kategorie, omezení regionů vydání, věkové omezení nebo ukázkové obrázky, a samozřejmě samotnou podepsanou aplikaci ve formátu AAB (Android App Bundle) [1]. Součástí publikace je také dostupná stránka se zásadami ochrany osobních údajů, která reflektuje požadavky GDPR.[1]

Po vydání aplikace je důležité zajistit její další sledování a údržbu. K tomu slouží například nástroje pro monitoring chyb, jako je Firebase Crashlytics (viz sekce 5.2), které umožňují identifikovat problémy vznikající v reálném provozu.

Aplikace je nyní pro uživatele plně dostupná v obchodě Google Play [2] a je stále možné ji pravidelně aktualizovat a sledovat její výkon a chování.

Konkrétní hardwarové a softwarové požadavky aplikace jsou uvedeny v příloze C. Zdrojový kód aplikace je možné sestavit podle návodu uvedeného v příloze D.

---

1. `https://developerthomas.cz/privacy_policy/my_farmer.html`
2. `https://play.google.com/store/apps/details?id=com.tondracek.myfarmer`

Závěr

Cílem této bakalářské práce bylo navrhnout a implementovat mobilní aplikaci pro podporu lokálního prodeje produktů menších producentů. Tento cíl byl splněn vytvořením aplikace, která umožňuje vyhledávání obchodů v okolí uživatele, jejich filtrování podle různých kritérií, vytváření vlastních obchodů a práci s uživatelskými profily a recenzemi.

V rámci práce byla nejprve provedena analýza existujících řešení a identifikace jejich nedostatků v oblasti podpory drobných producentů. Na základě této analýzy byl vytvořen návrh aplikace využívající standardní techniky softwarového inženýrství.

Následně byla navržena architektura aplikace, kde byl kladen důraz zejména na oddělení jednotlivých vrstev aplikace, přehlednost kódu a jeho dlouhodobou udržovatelnost. Velmi přínosným prvkem se ukázalo zavedení vlastního typu `DomainResult` pro reprezentaci výsledků operací, inspirovaného monadickým přístupem, který umožnil konzistentní a robustní řešení chyb. Celkově se zvolený návrh osvědčil při implementaci i postupném rozšiřování aplikace.

Součástí práce byla také implementace datové a doménové vrstvy, uživatelského rozhraní a bezpečnostního modelu. Řešeny byly také specifické problémy, jako je například geografické vyhledávání, správa stavu uživatelského rozhraní a práce s vedlejšími efekty nebo řízení přístupu k datům. Aplikace byla následně otestována pomocí jednotkových testů, manuálního testování i uživatelského testování, na jejichž základě došlo k dalším úpravám a zlepšení použitelnosti.

Výsledkem práce je funkční aplikace, která je připravena pro používání reálnými uživateli a je dostupná v obchodě Google Play. Práce tak splnila stanovené cíle a přinesla praktický výstup ve formě reálně použitelné mobilní aplikace.

Dalším logickým krokem, který již přesahuje rámec této práce, je propagace aplikace mezi komunitami odpovídajícími cílovým uživatelům, k čemuž je nyní aplikace připravena.

Bibliografie

[1] *About Android App Bundles*. Google. 26. ún. 2026. URL: https://developer.android.com/guide/app-bundle (cit. 13. 04. 2026).

[2] *About OpenStreetMap*. OpenStreetMap. URL: https://www.openstreetmap.org/about (cit. 21. 01. 2026).

[3] Hugo Allain. „Improving Productivity and Reducing Costs of Mobile App Development Using BaaS“. Master's thesis. Aalto University, 2020. URL: https://aaltodoc.aalto.fi/bitstreams/2d357f5b-4e2c-4452-ab41-bb944d478226/download (cit. 30. 04. 2026).

[4] *Amazon Cognito Documentation*. Amazon Web Services. URL: https://docs.aws.amazon.com/cognito/ (cit. 29. 04. 2026).

[5] *Android Developers*. Google. URL: https://developer.android.com (cit. 19. 04. 2026).

[6] *App architecture overview*. Google. 24. lis. 2025. URL: https://developer.android.com/topic/architecture#recommended-app-arch (cit. 03. 02. 2026).

[7] Luca Ardito et al. „Effectiveness of Kotlin vs. Java in Android App Development Tasks“. In: *Information and Software Technology* 127 (2020), s. 106374. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2020.106374. URL: https://doi.org/10.1016/j.infsof.2020.106374 (cit. 07. 05. 2026).

[8] *Architecture for Compose*. Google. 16. led. 2026. URL: https://developer.android.com/develop/ui/compose/architecture (cit. 30. 01. 2026).

[9] *Auth0 Documentation*. Auth0 Inc. URL: https://auth0.com/docs (cit. 29. 04. 2026).

[10] *Build better apps faster with Jetpack Compose*. Google. URL: https://developer.android.com/compose (cit. 22. 04. 2026).

[11] *Cloud Firestore Data model*. Google. 15. led. 2026. URL: https://firebase.google.com/docs/firestore/data-model (cit. 21. 01. 2026).

[12] *Cloud Firestore Documentation*. Google. 15. led. 2026. URL: https://firebase.google.com/docs/firestore (cit. 21. 01. 2026).

[13] *Cloud Functions for Firebase Documentation*. Google. 15. led. 2026. URL: https://firebase.google.com/docs/functions (cit. 21. 01. 2026).

[14] *Color roles — Material Design 3*. Google. URL: https://m3.material.io/styles/color/roles (cit. 31. 12. 2025).

[15] *Common modularization patterns*. Google. 5. břez. 2026. URL: https://developer.android.com/topic/modularization/patterns (cit. 21. 04. 2026).

[16] *Cross-platform and native app development: How do you choose?* JetBrains. 28. čvc. 2025. URL: https://kotlinlang.org/docs/multiplatform/native-and-cross-platform.html (cit. 20. 01. 2026).

[17] *Dependency injection in Android*. Google. 10. ún. 2025. URL: https://developer.android.com/training/dependency-injection (cit. 04. 02. 2026).

[18] *Dependency injection in Android*. Google. 10. ún. 2025. URL: https://developer.android.com/training/dependency-injection#hilt (cit. 03. 02. 2026).

[19] *Dependency injection with Hilt*. Google. 30. led. 2026. URL: https://developer.android.com/training/dependency-injection/hilt-android (cit. 03. 02. 2026).

[20] *Design systems in Jetpack Compose*. Google. 19. led. 2026. URL: https://developer.android.com/develop/ui/compose/designsystems (cit. 19. 01. 2026).

[21] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004. ISBN: 0-321-12521-5.

[22] Y. Evjen. *Introducing Material Theme Builder*. 27. říj. 2021. URL: https://m3.material.io/blog/material-theme-builder (cit. 09. 01. 2026).

[23] *Facebook*. Meta Platforms, Inc. 2026. URL: https://www.facebook.com (cit. 14. 04. 2026).

[24] *Facebook Marketplace*. Meta Platforms, Inc. 2026. URL: https://www.facebook.com/marketplace (cit. 14. 04. 2026).

[25] *Farmáři z regionu*. 2026. URL: https://experience.arcgis.com/experience/1bb702bc365647a6a12952ca36b2e276 (cit. 14. 04. 2026).

[26] *Firebase Android SDK*. Firebase Open Source. 20. led. 2026. URL: https://firebaseopensource.com/projects/firebase/firebase-android-sdk/ (cit. 20. 01. 2026).

[27] *Firebase Documentation*. Google. URL: https://firebase.google.com/docs (cit. 20. 04. 2026).

[28] *Firebase Security Rules*. Google. 10. dub. 2026. URL: https://firebase.google.com/docs/rules (cit. 20. 04. 2026).

[29] Mario Fuksa, Sandro Speth a Steffen Becker. „MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern“. In: *arXiv preprint arXiv:2504.18191* (2025). DOI: 10.48550/arXiv.2504.18191. URL: https://arxiv.org/abs/2504.18191 (cit. 02. 05. 2026).

[30] *Geo queries*. Google. 13. břez. 2026. URL: https://firebase.google.com/docs/firestore/solutions/geoqueries (cit. 13. 03. 2026).

[31] *Google Maps*. Google LLC. 2026. URL: https://maps.google.com (cit. 14. 04. 2026).

[32] Ruben Horn et al. „Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts“. In: *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2023, s. 44–54. DOI: 10.1109/mobilsoft59058.2023.00013. URL: https://doi.org/10.1109/mobilsoft59058.2023.00013 (cit. 07. 05. 2026).

[33] *Kodex dodavatelů na Scuku*. Scuk.cz. URL: https://www.blog.scuk.cz/kodex (cit. 17. 12. 2025).

[34] *Mapbox Maps SDK for Android*. Mapbox. URL: https://docs.mapbox.com/android/maps/ (cit. 01. 05. 2026).

[35] *Maps Compose for Android*. Google. 9. ún. 2026. URL: https://developers.google.com/maps/documentation/android-sdk/maps-compose (cit. 12. 02. 2026).

[36] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice-Hall, 2018. ISBN: 978-0-13-449416-6.

[37] *Material Design 3*. Google. URL: https://m3.material.io/get-started (cit. 09. 01. 2026).

[38] *Material Design 3 in Jetpack Compose*. Google. 19. led. 2026. URL: https://developer.android.com/develop/ui/compose/designsystems/material3 (cit. 20. 01. 2026).

[39] *Mobile Operating System Market Share Czech Republic*. StatCounter Global Stats. URL: https://gs.statcounter.com/os-market-share/mobile/czech-republic (cit. 27. 04. 2026).

[40] *Model-View-ViewModel (MVVM)*. Microsoft. 10. zář. 2024. URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (cit. 30. 01. 2026).

[41] *Navigation with Compose*. Google. 11. břez. 2026. URL: https://developer.android.com/develop/ui/compose/navigation (cit. 17. 03. 2026).

[42] *Paginate data with query cursors*. Google. 21. dub. 2026. URL: https://firebase.google.com/docs/firestore/query-data/query-cursors (cit. 22. 04. 2026).

[43] *Prodej přes Scuk*. Scuk.cz. URL: https://napoveda.scuk.cz/prodej-pres-scuk (cit. 17. 12. 2025).

[44] *Představujeme novou mapovou aplikaci: Farmáři z regionu!* Brněnská metropolitní oblast. 20. květ. 2025. URL: https://metropolitni.brno.cz/predstavujeme-novou-mapovou-aplikaci-farmari-z-regionu/ (cit. 17. 12. 2025).

[45] *Recommendations for Android architecture*. Google. 10. ún. 2025. URL: https://developer.android.com/topic/architecture/recommendations (cit. 30. 01. 2026).

[46] *Redux Fundamentals, Part 3: State, Actions, and Reducers*. Redux. 18. ún. 2025. URL: https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers (cit. 19. 03. 2026).

[47] *Scuk.cz*. 2026. URL: https://www.scuk.cz/ (cit. 14. 04. 2026).

[48] *Side effects in Jetpack Compose*. Google. 26. led. 2026. URL: https://developer.android.com/develop/ui/compose/side-effects (cit. 30. 01. 2026).

[49] Ian Sommerville. *Software Engineering*. 9. vyd. Boston: Pearson, 2011. ISBN: 978-0-13-703515-1.

[50] *Spring Framework*. VMware. URL: https://spring.io/projects/spring-framework (cit. 14. 04. 2026).

[51] *The difference between web apps, native apps, and hybrid apps*. Amazon Web Services. URL: https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/ (cit. 18. 02. 2026).

[52] *Thinking in Compose*. Google. 18. bře. 2026. URL: https://developer.android.com/develop/ui/compose/mental-model (cit. 19. 03. 2026).

[53] *Toasts overview*. Google. 26. ún. 2026. URL: https://developer.android.com/guide/topics/ui/notifiers/toasts (cit. 04. 04. 2026).

[54] *ViewModel Overview*. Google. 3. zář. 2025. URL: https://developer.android.com/topic/libraries/architecture/viewmodel (cit. 11. 02. 2026).

[55] Ayush Vijaywargi a Uchinta Kumar Boddapati. „Architectural Patterns in Android Development: Comparing MVP, MVVM, and MVI“. In: *International Journal for Research in Applied Science and Engineering Technology* 12.4 (dub. 2024), s. 4611–4616. ISSN: 2321-9653. DOI: 10.22214/ijraset.2024.60762. URL: https://doi.org/10.22214/ijraset.2024.60762 (cit. 07. 05. 2026).

[56] Philip Wadler. „Monads for Functional Programming“. In: *Advanced Functional Programming*. Sv. 925. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, s. 24–52. ISBN: 9783540594512. DOI: 10.1007/3-540-59451-5_2. URL: https://doi.org/10.1007/3-540-59451-5_2 (cit. 07. 05. 2026).

[57] *What is Backend as a Service (BaaS)?* Cloudflare. URL: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/ (cit. 09. 01. 2026).

[58] Kathryn Whitenton. *Minimize Cognitive Load to Maximize Usability*. Nielsen Norman Group. 22. pros. 2013. URL: https://www.nngroup.com/articles/minimize-cognitive-load/ (cit. 19. 01. 2026).

[59] *Write-time aggregations*. Google. 17. dub. 2026. URL: https://firebase.google.com/docs/firestore/solutions/aggregation (cit. 20. 04. 2026).

[60] *Zájem o lokální potraviny v Česku roste: Více než dvě třetiny lidí jsou ochotny si za ně připlatit*. Asociace malých a středních podniků a živnostníků ČR. URL: https://www.asociaceampi.cz/blog

/zajem-o-lokalni-potraviny-v-cesku-roste-vice-nez-dve-tretiny-lidi-jsou-ochotny-si-za-ne-priplatit/ (cit. 19. 02. 2026).

[61] *Zájem o značku Regionální potravina lámal rekordy i letos*. Státní zemědělský intervenční fond. 19. srp. 2025. URL: https://szif.gov.cz/cs/CmDocument?rid=%2Fapa_anon%2Fcs%2Fzpravy%2Fzpravy_o_fondu%2Ftiskove_zpravy%2F1755679700913.pdf (cit. 19. 02. 2026).

A Vybrané ukázky implementace

A.1 Implementace uživatelského rozhraní

```
class CreateShopViewModel @Inject constructor(
    private val createShop: CreateShopUC,
) : BaseViewModel<CreateShopEffect>() {

    private val _step = MutableStateFlow(ShopFlowStep.Initial)
    private val _input = MutableStateFlow(ShopInput.Empty)
    private val _isSubmitting = MutableStateFlow(false)

    val state: StateFlow<ShopFlowState> = combine(
        _step,
        _input,
        _isSubmitting
    ) { step, input, isSubmitting ->
        when {
            isSubmitting -> ShopFlowState.Loading
            else -> ShopFlowState.Editing(step, input)
        }
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5_000),
        initialValue = ShopFlowState.Loading,
    )

    private fun submitShop() = viewModelScope.launch {
        _isSubmitting.update { true }
        val shopInput: ShopInput = _input.value

        when (val result = createShop(shopInput)) {
            is DomainResult.Success -> {
                emitEffect(CreateShopEffect.
   ShowCreatedSuccessfully)
                emitEffect(CreateShopEffect.ExitShopCreation)
            }

```

```
34              is DomainResult.Failure ->
35                  emitEffect(CreateShopEffect.ShowError(result.
    error))
36          }
37          _isSubmitting.update { false }
38      }
39
40      private fun requestCreateFlowExit() = viewModelScope.
    launch {
41          emitEffect(CreateShopEffect.
    RequestExitShopCreationConfirmation)
42      }
43
44      fun onShopFormEvent(event: ShopFormEvent) =
45          _input.update { ShopFormEvent.applyEvent(it, event) }
46
47      fun onShopFlowEvent(event: ShopFlowEvent) = when (event) {
48          ShopFlowEvent.GoToNextStep -> _step.update { it.next()
    }
49          ShopFlowEvent.GoToPreviousStep -> _step.update { it.
    previous() }
50          ShopFlowEvent.Submit -> submitShop()
51          ShopFlowEvent.ExitShopFlow -> requestCreateFlowExit()
52      }
53  }
```

Listing A.1: Ukázka implementace ViewModelu pro vytvoření obchodu

A.2 Implementace bezpečnostních pravidel

```
1  rules_version = '2';
2  service cloud.firestore {
3    match /databases/{database}/documents {
4
5      //
6      // Helper function:
7      // Maps domain User ID (UUID) -> Firebase Auth UID
8      //
```

```
9     function getUserAuthId(userId) {
10      let userDoc = get(/databases/$(database)/documents/user/
    $(userId));
11      return userDoc.data.firebaseId;
12    }
13
14    match /user/{userId} {
15
16      // Anyone can read user profiles
17      allow read: if true;
18
19      allow create: if request.auth != null
20        && request.auth.uid == request.resource.data.
    firebaseId
21        && isValidUser(request.resource.data);
22
23      // Only this user (via firebaseId) may modify their
    profile
24      allow update: if request.auth != null
25        && request.auth.uid == resource.data.firebaseId
26        && isValidUser(request.resource.data);
27
28      // No client-side delete
29      allow delete: if false;
30
31      function isValidUser(data) {
32        return data.id == userId
33          && data.firebaseId is string
34          && data.name is string
35          && (data.profilePicture == null || data.
    profilePicture is string)
36          && data.contactInfo is map;
37      }
38    }
39
40    match /shop/{shopId} {
41
42      // Public read access
43      allow read: if true;
44
```

```
45      // Anyone logged in can create a shop for themselves
46      allow create: if request.auth != null
47        && request.resource.data.ownerId is string
48        && request.auth.uid == getUserAuthId(request.resource.
    data.ownerId)
49        && isValidShop(request.resource.data);
50
51      // Only the owner may update their shop
52      allow update: if request.auth != null
53        && request.auth.uid == getUserAuthId(resource.data.
    ownerId)
54        && request.auth.uid == getUserAuthId(request.resource.
    data.ownerId)
55        && isValidShop(request.resource.data)
56        && request.resource.data.ownerId == resource.data.
    ownerId;
57
58
59      // Only the owner may delete their shop
60      allow delete: if request.auth != null
61        && request.auth.uid == getUserAuthId(resource.data.
    ownerId);
62
63      function isValidShop(data) {
64        return data.id is string
65          && data.ownerId is string
66          && data.name is string
67          && data.description is string
68          && data.categories is list
69          && data.images is list
70          && data.menu is map
71          && data.location is map
72          && data.openingHours is map;
73      }
74    }
75
76    match /shop/{shopId}/review/{userId} {
77      // public read
78      allow read: if true;
79
```

```
80      // vytvoření recenze
81      allow create: if request.auth != null
82        && request.auth.uid == getUserAuthId(userId)
83        && isValidReview(request.resource.data)
84        && !isOwnShop(shopId, request.auth.uid);
85
86      // update not implemented
87      allow update: if false;
88
89      // delete jen autor
90      allow delete: if request.auth != null
91        && request.auth.uid == getUserAuthId(userId);
92
93      function isValidReview(data) {
94        return data.userId == userId
95          && data.shopId == shopId
96          && data.id is string
97          && data.rating is int
98          && data.rating >= 0
99          && data.rating <= 5
100         && (data.comment == null || data.comment is string)
101         && (
102           !('createdAt' in data) ||
103           data.createdAt is timestamp
104         );
105     }
106
107     function isOwnShop(shopId, authUid) {
108       let shop = get(/databases/$(database)/documents/shop/
    $(shopId));
109       let ownerAuthUid = getUserAuthId(shop.data.ownerId);
110       return ownerAuthUid == authUid;
111     }
112   }
113
114   // deprecated
115   // collectionGroup("review")
116   match /{path=**}/review/{reviewId} {
117     allow read: if true;
118   }
```

```
119
120     match /category_popularity/{catId} {
121
122       allow read: if true;
123       // Only cloud functions handle write
124       allow write: if false;
125     }
126   }
127 }
```

Listing A.2: Ukázka pravidel pro Firestore

B Scénáře uživatelského testování

B.1 Zadání testování

B.1.1 Scénář 1: Vyhledání obchodu pomocí filtrování

Zadání: Najděte obchod v okolí do 5 km, který prodává zeleninu a má hodnocení alespoň 3 hvězdičky.

Upřesnění: Využijte filtry pro kategorii, vzdálenost a hodnocení.

B.1.2 Scénář 2: Vytvoření obchodu

Zadání: Zaregistrujte se a vytvořte obchod zaměřený na prodej domácích vajec a brambor.

Upřesnění: Přidejte alespoň 2 fotografie (1 fotografie místa prodeje, 1 fotografie produktu), u produktů vyplňte alespoň název a cenu za určité množství a nastavte otevírací dobu (např. formou textové informace pro kontaktování).

B.1.3 Scénář 3: Úprava dostupnosti produktů

Zadání: Ve vytvořeném obchodě nastavte, že produkt „brambory“ není aktuálně dostupný.

Upřesnění: Změňte stav produktu na „není skladem“.

B.1.4 Scénář 4: Úprava uživatelského profilu

Zadání: Upravte svůj uživatelský profil.

Upřesnění: Přidejte profilovou fotografii, vyplňte jméno a přidejte alespoň jeden preferovaný kontakt.

B.1.5 Scénář 5: Vytvoření recenze

Zadání: Vytvořte recenzi u libovolného obchodu.

Upřesnění: Přidejte hodnocení i text recenze.

C Hardwarové a softwarové požadavky aplikace

Tato příloha shrnuje minimální požadavky na zařízení pro běh mobilní aplikace *My Farmer*.

C.1 Softwarové požadavky

- Minimální verze systému: Android 8.1 (API 27)
- Cílová verze systému: Android 16 (API 36)
- Požadována přítomnost služeb Google Play (Google Play Services)

C.2 Hardwarové požadavky

- Podpora OpenGL ES 3.0 nebo vyšší
- Připojení k internetu (Wi-Fi nebo mobilní data), které je nezbytné pro většinu funkcí aplikace

C.3 Podporované architektury

Aplikace podporuje následující procesorové architektury:

- arm64-v8a
- armeabi-v7a
- x86
- x86_64

C.4 Volitelné funkce zařízení

Některé funkce aplikace mohou využívat volitelné hardwarové prvky zařízení:

- Lokalizační služby (GPS)

Tyto funkce nejsou nezbytné pro základní běh aplikace.

C.5 Omezení kompatibility

Aplikace není určena pro zařízení:

- bez podpory OpenGL ES 3.0
- bez služeb Google Play

D Sestavení a spuštění aplikace

Pro sestavení a spuštění aplikace ze zdrojových kódů je nutné mít k dispozici následující:

- Vývojové prostředí Android Studio (verze 2025.3.4 nebo novější)
- Připojení k internetu pro stažení projektových závislostí a potřebných komponent SDK

Po stažení souboru `code.zip` z archivu bakalářské práce a jeho rozbalení otevřete projekt v prostředí Android Studio. Následně dojde ke stažení potřebných závislostí a konfiguraci projektu.

Aplikaci bude následně možné sestavit a poté spustit:

- na fyzickém zařízení s Androidem 8.1 (API 27) nebo novějším, s povoleným režimem pro vývojáře a laděním přes USB
- nebo pomocí emulátoru v prostředí Android Studio

Pro správnou funkčnost aplikace je vyžadováno:

- připojení k internetu
- splnění minimálních systémových požadavků cílového zařízení uvedených v příloze C