FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Android Jetpack v mobilních aplikacích

Jakub Šlechta

PLZEŇ

2023

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Android Jetpack v mobilních aplikacích

Jakub Šlechta

Vedoucí práce
Ing. Ladislav Pešička

PLZEŇ 2023

Citace v seznamu literatury:
ŠLECHTA, Jakub. *Android Jetpack v mobilních aplikacích*. Plzeň, 2023. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Ladislav Pešička.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub ŠLECHTA**
Osobní číslo:	**A20B0243P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Android Jetpack v mobilních aplikacích**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prozkoumejte, jaké možnosti a výhody přináší Android Jetpack při vytváření mobilních aplikací na platformě Android.
2. Navrhněte vhodnou netriviální aplikaci, na které ukážete použití Android JetPacku.
3. Navrženou aplikaci realizujte, ověřte její funkcionalitu reprezentativní sadou testů na platformě Android a navrhněte další možná rozšíření aplikace.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Ing. Ladislav Pešička**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **3. října 2022**
Termín odevzdání bakalářské práce: **4. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2022

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Plzni dne 25. března 2023

. . . . . .
Jakub Šlechta

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Cílem této práce je prozkoumat jaké možnosti a výhody Android Jetpack přináší při vývoji na platformě Android. Práce se zabývá vývojem uživatelského rozhraní s využití nástroje Jetpack Compose a pochopením doporučovaných principů z Android Jetpacku pro vývoj softwaru. V praktické části se pak především zaměříme na architekturu projektu a její praktickou realizaci na vytvářené aplikaci. Na závěr práce ověříme funkcionalitu aplikace sadou testů a navrhneme možná rozšíření budoucí verze aplikace.

Abstract

Android Jetpack in mobile applications. The aim of this thesis is to investigate what possibilities and benefits Android Jetpack brings to development on the Android platform. The thesis deals with the development of user interface using Jetpack Compose tool and understanding the recommended principles from Android Jetpack for software development. In the practical part, we will mainly focus on the project architecture and its practical implementation on the application being developed. Finally, we will verify the functionality of the application with a set of tests and suggest possible extensions of the future versions of the application.

Klíčová slova

Android • Kotlin • uživatelská rozhraní • architektura • cloudové služby • testování

Poděkování

Na tomto místě bych rád vyjádřil své upřímné poděkování všem, kteří přispěli k úspěšnému dokončení mé bakalářské práce.

Především bych rád poděkoval svému vedoucímu práce, panu Ing. Ladislavu Pešičkovi, za jeho trpělivost, cenné rady a vstřícnost během celého procesu tvorby této práce.

Obsah

Úvod 1

V současnosti je trh mobilních aplikací velmi konkurenční a vyžaduje od vývojářů, aby byli schopni rychle a efektivně vyvíjet kvalitní aplikace. Android Jetpack je kolekce knihoven a nástrojů, které usnadňují vývoj aplikací pro platformu Android. Tyto knihovny a nástroje poskytují vývojářům širokou škálu funkcí, jako je například správa životního cyklu aktivit, práce s daty a úložištěm, zabezpečení a mnoho dalšího.

Cílem této bakalářské práce je představit Android Jetpack jako efektivní nástroj pro vývoj aplikací pro platformu Android a zvláště se pak zaměřit na Jetpack Compose a doporučovanou architekturu projektu podporovanou vývojáři Android Jetpacku. Práce bude rovněž analyzovat výhody a nevýhody použití poskytovaných technologií a porovnávat je s předešlými nástroji pro vývoj aplikací.

Současný stav vývoje je velmi rychlý, s mnoha novými funkcemi a vylepšeními přidávanými každým měsícem. Směr vývoje se zaměřuje na zlepšování uživatelského rozhraní, kompatibility s jinými knihovnami a nástroji, stejně jako na poskytování většího pohodlí a efektivity pro vývojáře. Google, který vyvíjí Android Jetpack, do tohoto nástroje investuje, aby stále splňoval moderní požadavky pro vývoj aplikací. V praktické části práce se proto zaměříme na nejnovější knihovny a zkusíme si ukázat moderní přístup vývoje na platformě Android. Vytvořená aplikace bude sloužit jako jednotný nástroj pro hledání a publikaci ztracených předmětů a měla by ukázat praktické využití Android Jetpacku.

Operační systém Android

2

V této kapitole se zmíníme o vývoji, principech a architektuře Android systému. Dále se zmíníme o životním cyklu aplikace a možnostech vývoje.

2.1 Vývoj systému

Operační systém Android byl poprvé představen veřejnosti v roce 2008 a od té doby se stal jedním z nejrozšířenějších operačních systémů na světě [1]. Tento operační systém byl vyvinut společností Android Inc., kterou později koupila společnost Google [2].

Vývoj operačního systému byl od počátku zaměřen na poskytnutí open-source platformy pro vývoj aplikací a na poskytování širokého spektra funkcí a služeb pro uživatele [1]. Android se od svého počátku snažil konkurovat již rozšířeným operačním systémům, jako je iOS od společnosti Apple, a nabídnout alternativu pro vývoj aplikací pro chytré telefony a tablety.

Dnes se Android stal nejen populárním operačním systémem pro chytré telefony a tablety, ale také se rozšířil do dalších oblastí, jako je například automobilový průmysl, kde se používá pro infotainment systémy v autech[1]. To ukazuje na širokou a rostoucí oblibu operačního systému Android a jeho důležitost v současném světě technologií.

2.2 Architektura

Architektura operačního systému Android se skládá z několika vrstev, které spolu spolupracují, aby zajistily stabilní chod operačního systému. Jádrem operačního systému Android je Linuxové jádro, které zajišťuje základní funkce pro správu operačního systému, jako je modul pro správu paměti, správu procesů, správu I/O či správu souborů [1].

[1] Infotainment systémy v autech poskytují služby jako asistenci řízení, informace o vozidle či aktuální zpravodajství pomocí aplikací v přístrojové desce auta.

Nad Linuxovým jádrem je vrstva aplikačního frameworku, která zahrnuje řadu knihoven a API (application programming interface), které umožňují vývojářům vytvářet aplikace pro Android. Tyto knihovny zahrnují funkce pro práci s grafikou, sítí, úložištěm a mnoho dalšího [1].

Android Jetpack je navržen tak, aby vývojářům umožnil jednodušší a intuitivnější vývoj aplikací pro Android. Tento nástroj využívá architekturu Androidu a poskytuje vývojářům jednoduchý a přehledný způsob, jak implementovat funkce a služby nabízené operačním systémem.

Závěrem lze říci, že Android Jetpack a architektura Androidu úzce souvisí. Architektura Androidu umožňuje vývoj aplikací pro Android, zatímco Android Jetpack pomáhá vývojářům tyto aplikace implementovat a optimalizovat.

2.3 Možnosti vývoje aplikací

Existují dva hlavní typy vývoje aplikací pro operační systém Android: nativní vývoj a multiplatformní vývoj.

Nativní vývoj aplikací pro Android se zaměřuje na vývoj aplikací, které jsou optimalizovány pro operační systém Android a využívají plně funkce a služby nabízené operačním systémem. Tyto aplikace se většinou vyvíjejí v programovacím jazyce Kotlin, C++ nebo Java a využívají nástrojů jako je Android Studio.

Multiplatformní vývoj se snaží vytvořit aplikace, která bude fungovat na více platformách, například na iOS, Androidu a dalších operačních systémech. Tyto aplikace se vyvíjejí například v programovacím jazyce Dart s nástrojem Flutter[2] nebo za použití frameworků jako je React Native, Xamarin či Ionic.

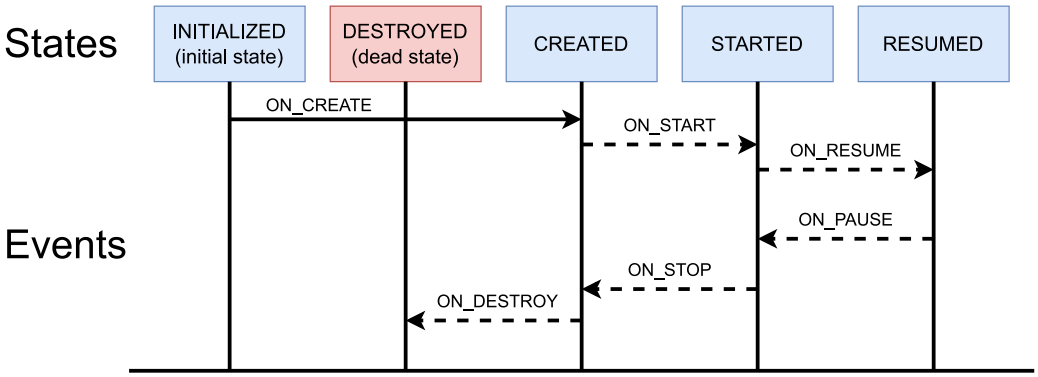
V této bakalářské práci se zaměříme na nativní vývoj aplikací pro operační systém Android s využitím programovacího jazyka Kotlin. Při vývoji aplikace využijeme vývojové prostředí Android Studio.

2.3.1 Životní cyklus

V práci se často odkazujeme na životní cykly aplikace (viz obr. 2.1). Životní cykly jsou spravovány operačním systémem nebo frameworkem spuštěným v procesu aplikace. Jsou základem fungování Androidu a naše aplikace je musí respektovat. Pokud tak neučiníme, může dojít k úniku paměti, pádům aplikace či nekonzistentním stavům [4].

Životní cyklus aplikace reaguje na vnější podněty jako vypnutí aplikace, otočení přístroje, tmu či málo paměti.

[2] Flutter je bezplatný open-source framework používaný pro tvorbu uživatelského rozhraní vydaný v květnu 2017 společností Google [3].



Obrázek 2.1: Stavy a události, které tvoří životní cyklus aktivity systému Android [4]

Naším cílem je správně reagovat na tyto stavy a udržet tak konzistentní stav aplikace. Příkladem může být přihlášený uživatel, který aplikaci skryje na pozadí a operační systém, který se po čase rozhodne aplikaci z důvodů omezení zdrojů ukončit. Pokud si poté uživatel obnoví aplikaci, musíme ověřit, zda je stále přihlášený, a má oprávnění vidět danou obrazovku. Podobných problémů může být několik a naším úkolem je postarat se o to, aby bylo vše ošetřeno, například použitím knihoven z Android Jetpacku, které jsou si vědomy životních cyklů a respektují je.

Android Jetpack 3

V této kapitole bude podrobně rozebrán Android Jetpack, k čemu slouží a co nového přináší. Zmíníme se o rozdílech mezi Support Library a Android Jetpack a proč bylo potřeba vydat novou generaci knihoven.

3.1 Support Library

Support Library byla předchůdcem Android Jetpacku a poskytovala funkce pro starší verze Androidu, které byly nezbytné pro vývoj aplikací. Na první pohled se může zdát, že Android Jetpack nepřináší nic nového, a že se jedná pouze o přejmenování již předešlých knihoven, ale není tomu tak.

Android Jetpack je sada nástrojů, knihoven a architektonických vzorů, které usnadňují vývoj aplikací pro Android [5]. V praxi to pak znamená, že je za nás řešeno několik problému z předešlých verzí Support Library. Mezi nejčastěji zmiňované problémy patří nutnost psát takzvaný boilerplate code, tedy kód, který se stále opakuje, a šel by zautomatizovat lepším návrhem. Už tato změna má velký vliv na vývoj, jelikož vytváří méně prostoru pro zavedení chyb.

Pokud bychom tedy například chtěli použít *constraint layout*, který slouží pro vytváření responzivních aplikací, v Support Library by se importoval z následujícího balíčku:

```
com.android.support.constraint:constraint-layout
```

a s použitím Android Jetpacku:

```
androidx.constraintlayout:constraintlayout
```

Android Jetpack zaštiťuje veškeré knihovny pod jednu doménu [5]:

```
androidx.*
```

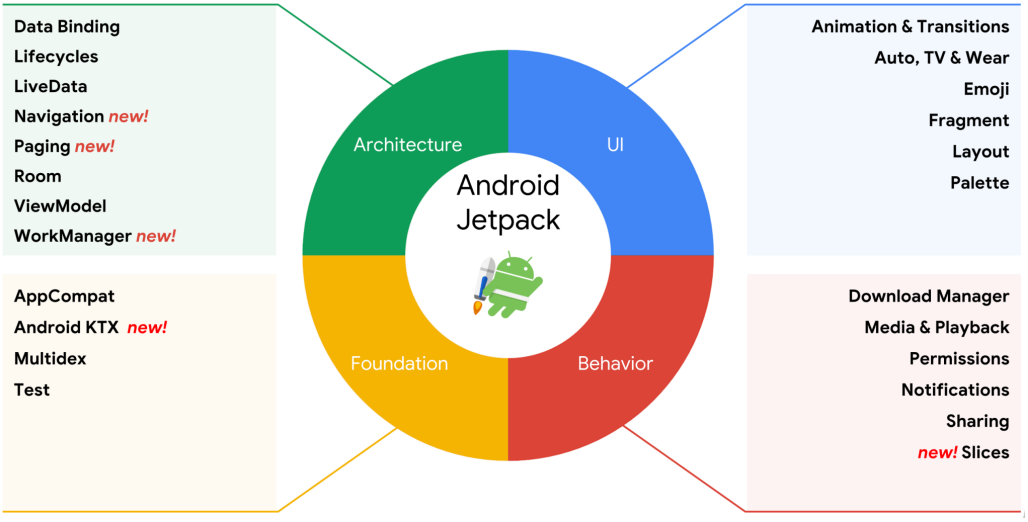
Jetpack na rozdíl od Support Library udržuje a aktualizuje balíčky `androidx` samostatně. Balíčky `androidx` používají striktní sémantické verze, počínaje verzí `1.0.0`. Způsob, jakým se jednotlivá čísla mění, vyjadřuje co bylo změněno od jedné

verze ke druhé a co dané změny představují. Inkrementace prvního čísla představuje potencionálně nekompatibilní změny, inkrementace druhého čísla představuje přidání zpětně kompatibilních funkcí a u posledního čísla se jedná o opravu chyb [6]. Knihovny tedy můžeme v projektu aktualizovat nezávisle [5].

Poslední verzí balíčku `android.support` byla revize `28.0.0` Support Library, která byla vydána jako stabilní v září 2018. Další verze v rámci Support Library již vydávány nebudou a měly by být považovány za zastaralé [7].

3.2 Komponenty

Android Jetpack se dále dělí do čtyř komponent: základ, architektura, chování a uživatelské rozhraní (viz obr. 3.1). Jejich funkce si nyní podrobně rozebereme.



Obrázek 3.1: Jednotlivé Android Jetpack komponenty [8]

3.2.1 Základ

Komponent se zaměřuje na základní funkce a služby, které jsou nutné pro vývoj aplikací. Tyto služby zahrnují například:

AppCompat Tato knihovna poskytuje *kompatibilitu* se staršími verzemi Androidu a umožňuje vývojářům používat nové funkce na starších zařízeních.

Android KTX Tato knihovna poskytuje různá rozšíření pro vývojáře, která využívají několik funkcí jazyka *Kotlin*, jako je například zápis lambda funkcí nebo možnost výchozích hodnot parametrů u funkcí a metod.

Multidex Aplikace pro Android obsahují spustitelné soubory bajtkódu ve formě souborů Dalvik executable (DEX), který je následně komprimován do jednoho souboru `.apk`. Jeden soubor `.dex` může obsahovat maximálně 65 536 metod (referencí) [9]. Tento limit se s použitím Android frameworků a různých knihoven v dnešních aplikacích snadno přesáhne. Tato knihovna umožňuje aplikacím tento limit překročit a používá se u starších API Androidu pro *zpětnou kompatibilitu*.

Test Tato knihovna poskytuje nástroje pro automatizované *testování* aplikací.

DataStore Tato knihovna poskytuje bezpečné a snadno použitelné API pro lokální ukládání a načítání uživatelských nastavení a dat (viz kapitola 8.4 s ukázkou použití).

Výše uvedené služby poskytují základní funkce pro vývoj Android aplikací, a to bez ohledu na konkrétní architekturu nebo designový vzor. To umožňuje vývojářům věnovat více času a energie vývoji funkčnosti aplikací a méně času konfiguraci základních služeb.

Důraz je zde kladen na vývoj v jazyce Kotlin, knihovny pro snadné testování a zpětnou kompatibilitu.

3.2.1.1 Zpětná kompatibilita

Zpětná kompatibilita v Androidu umožňuje aplikacím pracovat na různých verzích operačního systému Android, aniž by byly jejich funkce nebo vzhled negativně ovlivněny. To znamená, že aplikace, které byly napsány pro starší verze Androidu, by měly fungovat stejným způsobem i na novějších verzích systému bez nutnosti aktualizací nebo oprav.

Zpětná kompatibilita je důležitá, protože umožňuje uživatelům používat své aplikace i po aktualizaci na novější verzi Androidu, aniž by bylo nutné vytvářet a stahovat nové verze těchto aplikací, což je výhodné jak pro uživatele, tak pro vývojáře.

3.2.2 Architektura

Komponent architektury v Android Jetpacku se zaměřuje na návrh a vývoj aplikací. Tyto služby poskytují vývojářům řešení pro správnou architekturu projektů, které jsou *robustní*, odolné vůči chybám a snadno se *testují*. V kapitole 5 se zaměříme na doporučení a rady z Android Jetpacku ohledně architektury projektu. Služby této komponenty zahrnují:

Data Binding Umožňuje při vývoji deklarativně svázat jednotlivé prvky uživatelského rozhraní v našem rozložení s daty aplikace.

Lifecycles Tato knihovna umožňuje vývojářům snadno řídit životní cyklus aktivit a fragmentů.

LiveData Třída, která slouží pro uchovávání a získávaní dat s ohledem na životní cyklus aplikace. Její použití si ukážeme v kapitole 5.1.1.

Navigation Tato knihovna umožňuje vývojářům vytvářet a spravovat navigaci v aplikacích, tedy zjednodušeně řečeno přesuny mezi jednotlivými obrazovkami.

Paging Tato knihovna umožňuje vývojářům snadno a efektivně stránkovat velké množství dat. V praxi by to mohla být implementace tabulky, která by načítala data postupně podle vstupů od uživatele, například přejetím prstu od spodního okraje obrazovky nahoru.

Room Knihovna umožňuje vývojářům jednoduše pracovat s lokálním úložištěm poskytnutím abstraktní vrstvy nad SQLite databází. Oproti knihovně DataStore zmíněné v 3.2.1 slouží pro ukládání většího množství dat.

ViewModel Tato knihovna umožňuje vývojářům udržet stav aplikací při změnách konfigurace, jako je rotace zařízení. Je také důležitým prvkem při vytváření architektury projektů. Její využití si ukážeme v kapitole 5.1.1, kde ji použijeme pro implementaci různých návrhových vzorů a principů architektury projektu.

WorkManager Tato knihovna umožňuje vývojářům plánovat a spravovat úlohy, které se mají provést na pozadí, aniž by bylo nutné se starat o řízení životního cyklu.

Hilt Tato knihovna poskytuje jednoduchý způsob injektování závislostí do aplikací. Jednou z hlavních výhod je snadnější *testování* a mockování[1] objektů. Její důležitost je ukázána v kapitole 5.4 kde se pojednává o principu správy závislostí v projektu.

Výše uvedené služby poskytují jednoduché a efektivní řešení pro správnou architekturu aplikací, což je klíčové pro vývoj rozsáhlých produkčních aplikací. Tyto služby vývojářům umožňují vyhnout se potenciálním problémům s výkonem, chybami a laděním, a také jim umožňují věnovat se rozvoji funkčnosti aplikace místo řešení technických problémů.

Důraz je zde kladen na testovatelnost, tedy vytvoření aplikace, která se snadno testuje a architekturu projektu, tedy jak zařídit, aby velké projekty byly stále čitelné a udržitelné i s postupem času.

[1] Mockování je proces, kdy není volána konkrétní instance dané třídy, ale její náhražka za reálný objekt.

3.2.3 Chování

Komponent zaměřující se na očekávané chování aplikací. Tyto služby poskytují vývojářům jednoduché řešení pro očekávané chování aplikací, které jsou intuitivní pro uživatele. Tyto služby zahrnují:

Download Manager Systémová služba, která se stará o dlouhotrvající síťové požadavky. Stahování probíhá na pozadí a obsahuje mechanismy, které si poradí i s výpadkem konektivity, například obnova stahování po selhání nebo změně připojení a restartu systému.

Media & Playback Knihovny poskytující komplexní sadu nástrojů a knihoven pro implementaci přehrávání, nahrávání a zpracování médií v aplikaci.

Permissions Umožňuje vývojářům snadno získávat a kontrolovat oprávnění v aplikacích.

Notifications Poskytuje vývojářům možnost jednoduše vytvářet a spravovat upozornění v aplikacích.

Sharing Tato knihovna umožňuje vývojářům snadno sdílet data mezi aplikacemi.

Slices Slice je malá část uživatelského rozhraní aplikace, která zobrazuje relevantní informace nebo akce a může být zobrazena na různých místech, jako například v aplikaci Google Search nebo v Google Assistant. Slices mohou být použity k poskytnutí rychlého přístupu k funkcionalitě aplikace s tím, že aplikace nezabírá celou obrazovku zařízení, ale pouze určitou část.

Důraz je zde kladen na konzistenci a jednotnost chování aplikací v rámci celého ekosystému Androidu.

3.2.4 Uživatelské rozhraní

komponent se zaměřuje na vývoj uživatelského rozhraní. Poskytuje služby vývojářům pro snadnou a efektivní tvorbu uživatelského rozhraní, které je příjemné pro uživatele a umožňuje jim intuitivně ovládat aplikaci. Tyto služby zahrnují:

Animation & Transitions Obsahuje třídy a nástroje pro vytváření vlastních animací a přechodů obrazovek a poskytuje sadu předpřipravených animací a přechodů, které pomohou rychle implementovat běžné moderní chování aplikace.

Auto, TV & Wear Tato část poskytuje podporu pro vytváření aplikací pro Android TV, Wear OS a infotainment systémy v autech. Obsahuje sadu komponent uživatelského rozhraní, které jsou *optimalizovány* pro tyto platformy, jako například knihovnu *leanback* pro Android TV.

Emoji Obsahuje sadu nástrojů pro práci s emoji, jako například převádění znaků emoji na zdroje obrázků a poskytuje systém pro výběr a zadávání emoji v aplikacích.

Fragment Tato knihovna poskytuje podporu pro vytváření a správu fragmentů v aplikaci. Fragmenty si lze představit jako znovupoužitelnou komponentu uživatelského rozhraní, které lze přidávat do aktivity nebo jiného fragmentu. Aktivitu si lze představit jako jednu obrazovku aplikace.
Knihovna obsahuje třídy a nástroje pro správu životního cyklu fragmentů, přidávání a odebírání fragmentů a komunikaci mezi fragmenty.

Layout Podpora pro vytváření a správu rozvržení uživatelského rozhraní v aplikaci. Obsahuje sadu tříd, které umožní uspořádat prvky obrazovky různými způsoby, stejně jako nástroje pro vytváření *responzivních* rozvržení, která se přizpůsobují různým velikostem a orientacím obrazovky.

Palette Tato služba poskytuje podporu pro extrahování barev z obrázků a aplikování těchto barev na uživatelské rozhraní aplikace. Obsahuje sadu tříd a nástrojů pro analýzu obrázků a generování barevných palet. Programátoři tak mohou například vytvářet aplikace, které se adaptují na barvu tapety zařízení.

Material Design Tato knihovna poskytuje jednoduchý přístup k prvkům Material Designu, které zajišťují *konzistentní* vzhled aplikací. Jedná se o open-source designový standard od společnosti Google používaný v mnoha aplikacích. Její použití je možné vidět v ukázce 4.1.

Widgets Podporu pro vytváření a správu widgetů v aplikaci. Widgety jsou způsobem poskytování rychlého přístupu k funkčnosti a informacím v aplikaci na domovské obrazovce nebo na uzamčené obrazovce uživatele. Tento modul zahrnuje sadu předdefinovaných widgetů, stejně jako třídy a nástroje pro vytváření vlastních widgetů.

Výše uvedené služby umožňují vývojářům vyhnout se potenciálním chybám a snaží se ušetřit čas a úsilí, které by jinak museli vynaložit na vývoj těchto prvků samostatně.

Zároveň vývojáři nejsou limitováni a mohou použít jak již vytvořené komponenty, tak využít podpory pro snadnou implementaci nových funkcí.

V závěru lze říci, že tento komponent poskytuje vývojářům komplexní a efektivní řešení pro tvorbu uživatelského rozhraní. Tyto služby umožňují vývojářům snadno vytvářet uživatelsky přívětivé aplikace a udržet je aktuální a v souladu s nejnovějšími trendy.

Důraz je zde kladen na optimalizace, jednotný vzhled, responzivní design, flexibilní možnosti a snadnou integraci již ověřených služeb.

3.2.5 Shrnutí

Výše zmíněné knihovny a služby jednotlivých komponent nejsou zdaleka všechny, které Android Jetpack poskytuje, ale pouze vybraná množina často používaných technologií. Většina již byla vydaná v předešlé verzi Support Library a zde jsou pouze začleněny do jednoho logického celku.

Pro vývojáře jsou důležité častější aktualizace, spojení všeho podstatného pod jednu doménu `androidx`, dodržování osvědčených postupů a omezení nutnosti psát boilerplate kód. Další výhodou Android Jetpacku je plné využití jazykových vlastností Kotlinu, které tvorbu aplikací značně usnadňují.

Příchod Android Jetpacku pomohl vyřešit časté problémy jako správu životního cyklu aplikace, změny konfigurace či úniky paměti.

Jetpack Compose 4

Imperativní programování bylo dlouhou dobu oblíbené paradigma v oblasti vytváření uživatelského rozhraní. Příkladem může být vytváření rozhraní pomocí XML (extensible markup language) souborů v Androidu, kde navrhujeme jednotlivé prvky uživatelského rozhraní a v dalším souboru s jinou syntaxí řešíme načítání a spárování dat s uživatelským rozhraním. Oproti tomu deklarativní způsob vytváření uživatelského rozhraní vykresluje jednotlivé prvky na základě přijatých dat, tedy při definování uživatelského rozhraní pracujeme již z předanými daty a můžeme při vývoji plně využít jazykových konstrukcí daného jazyka.

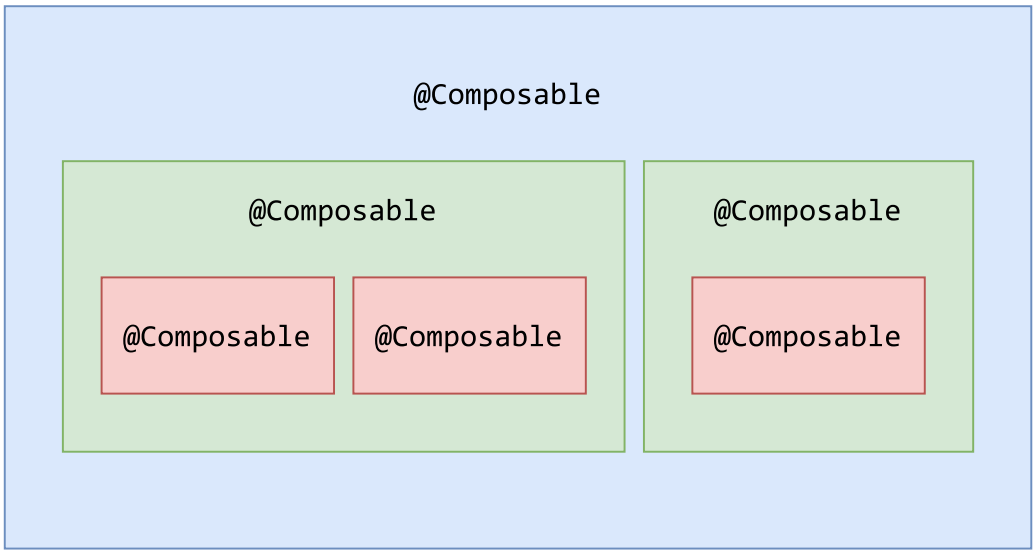
Deklarativní způsob programovaní přináší několik výhod:

- snižuje náklady na vývoj,
- snáze se čte a udržuje,
- pomáhá rozvíjet efektivitu a možnosti vývojářů zavedením abstraktnějšího přístupu,
- většinou postačí znát jeden jazyk, ve kterém se udržuje celý projekt.

Jetpack Compose je doporučená moderní sada nástrojů, představena v roce 2021, která slouží pro vytváření nativního uživatelského rozhraní [10]. Zařadili bychom ji tedy do komponenty uživatelského rozhraní v Android Jetpacku. Využívají ho aplikace jako je například Google Play, Pinterest, Twitter, Disney, Airbnb a Dropbox [10].

Jetpack Compose poskytuje intuitivní způsob, jak vytvářet grafická rozhraní pomocí deklarativního programování, které umožňuje vývojářům vytvářet aplikace s použitím Kotlinu bez potřeby používat XML soubory [11].

Dříve se jednotlivé části uživatelského rozhraní vytvářely dědičností, příkladem může být `EditText`, který dědí od `TextView`, který dědí od `View` [1]. Jetpack Compose však zvolil jiný přistup a uživatelské rozhraní sestavuje z takzvaných *composables* [11] (viz obr. 4.1).



Obrázek 4.1: Hierarchie sestavení uživatelského rozhraní v Jetpack Compose

Příkladem composable komponenty může být jednoduché tlačítko definované v ukázce kódu 4.1, kde je daná komponenta označena anotací `@Composable`.

4.1 Výhody

V další části textu si popíšeme výhody, které nám Jetpack Compose přináší. Především se jedná o změnu syntaxe a hotové implementace osvědčených postupů, které si dříve musel programátor sám implementovat a otestovat.

4.1.1 Intuitivnost

Jetpack Compose používá deklarativní funkční styl programování, který usnadňuje vytváření a udržování komplexních dynamických uživatelských rozhraní. Jednotlivé prvky obrazovky definujeme jako Kotlin funkce, které si drží svůj stav a při změně se samotný framework postará o překreslení.

Zanořováním jednotlivých composable funkcí vytváříme hierarchii uživatelského rozhraní a použitím Kotlinu můžeme snadno vytvářet komplexní rozložení,

[1] `View` v Android aplikaci představuje základní stavební kámen uživatelského rozhraní. Jedná se o prvek, který zobrazuje určitou část uživatelského rozhraní aplikace, jako například tlačítko, textové pole nebo obrázek.

která jsou čitelná a intuitivní. Například smyčkou `for` lze generovat jednotlivé composables, což nám umožňuje dynamické vytváření uživatelského rozhraní.

Zde je příklad, jak můžete definovat jednoduché tlačítko s využitím Jetpack Compose:

```
// MyButton.kt

package cz.zcu.students.compose_demos.components

import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable

@Composable
fun MyButton(onClickHandler: () -> Unit) {
    Button(onClick = onClickHandler) {
        Text("Click here")
    }
}
```

Zdrojový kód 4.1: Ukázka definice tlačítka s Jetpack Compose

V této ukázce kódu je funkce `MyButton` opatřena anotací `@Composable`, která říká, že ji má framework zařadit do hierarchie uživatelského rozhraní. Composable tlačítko převezme jako argument funkci `onClickHandler` a vrátí prvek tlačítka s podřízeným prvkem `Text` s hodnotou `Click here`. Při stisknutí tlačítka se zavolá funkce `onClickHandler` předaná jako argument a na standardní výstup[2] se vypíše `handle onClick`.

Poté by hlavní aktivita aplikace vypadala například takto:

```
package cz.zcu.students.compose_demos

// other imports...

import cz.zcu.students.compose_demos.components.MyButton

fun handler() {
    println("handle onClick")
}

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyButton(onClickHandler = { handler() })
        }
    }
}
```

Zdrojový kód 4.2: Kompletní příklad tlačítka a hlavní aktivity

[2]Pro zobrazení standardního výstupu je třeba využít speciálních nástrojů či prostředí jako je Android Studio.

Zároveň si můžeme z ukázky kódu 4.1 ověřit, že skutečně importujeme knihovny z balíčků `androidx.*`, a navíc je zde využit již zmiňovaný Material Design uvedený v kapitole 3.2.4.

Jetpack Compose zápis je lépe čitelný než zápis pomocí XML. Pro porovnání si ukážeme, jak bychom mohli dosáhnout podobného výsledku s použitím XML zápisu:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="handler"
        android:text="Click here"
        tools:layout_editor_absoluteX="0dp"
        tools:layout_editor_absoluteY="0dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Zdrojový kód 4.3: XML soubor s definicí hlavní aktivity a tlačítka

Ukázka kódu 4.1 zapsaná v Jetpack Compose je na první pohled lépe čitelná, než ukázka kódu 4.3 zapsaná v XML. Navíc zápis pomocí Compose umožňuje plně využívat jazykových konstrukcí Kotlinu společně s jednotnou strukturou projektu.

Pro úplnost ještě ukážeme zdrojový kód 4.4 hlavní aktivity a definovaného tlačítka v XML, který je velmi podobný ukázce kódu 4.2:

```
package cz.zcu.students.xml_demos

// other imports...

import android.view.View


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    fun handler(view: View) {
        println("handle onClick")
    }
}
```

Zdrojový kód 4.4: Kompletní příklad tlačítka a hlavní aktivity s využitím XML zápisu

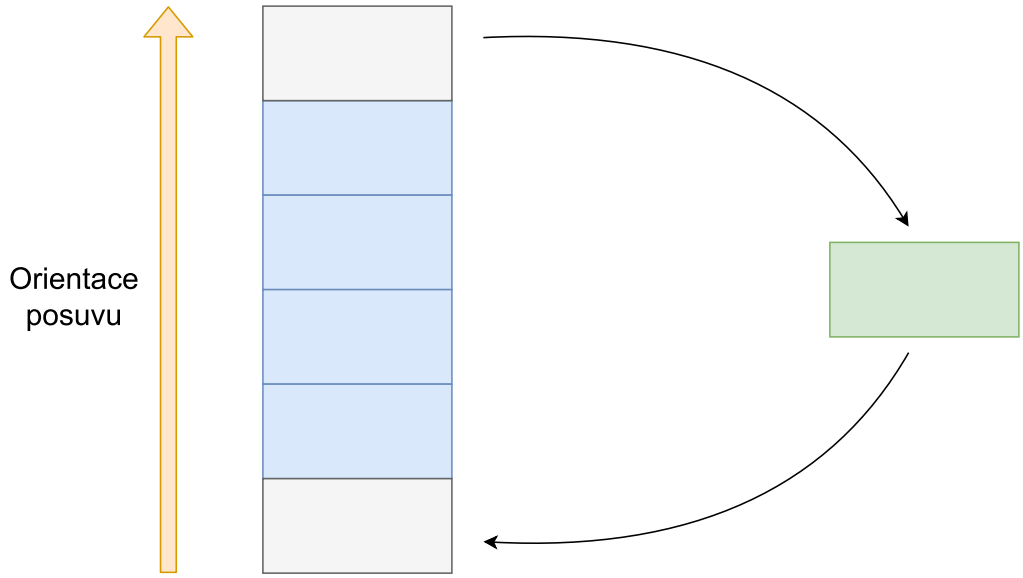
4.1.2 Optimalizace

Jetpack Compose používá řadu technik pro optimalizaci výkonu uživatelského rozhraní. Řeší také několik problémů, které si nyní rozebereme.

4.1.2.1 View recycling

Správa paměti je klíčovým aspektem vývoje aplikací. Vzhledem k tomu, že mobilní zařízení mají velmi omezenou paměť, je nutné ji v našich aplikacích používat opatrně. Jedním z osvědčených postupů, které s tím souvisí, je *view recycling*.

Bývá zvykem používat co nejméně paměti při vykreslovaní komponent. Pokud bychom například měli seznam s miliony položek, nemá smysl pro každou řádku vytvářet novou komponentu, jelikož by to zabralo příliš mnoho paměti. Místo toho vytvoříme pouze omezený počet komponent, které se právě vejdou na obrazovku a při dotazu od uživatele o načtení dalších řádek pouze nahradíme zakrytý obsah v právě vykresleném seznamu požadovaným obsahem (viz obr. 4.2). Tento proces se nazývá *view recycling* a Jetpack Compose ho automaticky řeší za nás.



Obrázek 4.2: Recyklace skryté řádky (komponenty) pro úsporu paměti

Takto, namísto mnoha komponent, může být naše obrazovka vytvořena pouze s několika z nich.

4.1.2.2 Lazy loading

Při stahování velkého množství dat z internetu se můžeme setkat se „zamrznutím“ uživatelské rozhraní aplikace. Tento problém lze vyřešit optimalizační technikou zvanou *lazy loading* nebo také *on-demand loading*.

Namísto načítání veškerých dat najednou, načteme pouze důležitá data, která se mají právě zobrazit a dokud uživatel nebude zbytek potřebovat, nebudeme jej stahovat.

Tento přístup má již několik výhod, jako snížení spotřeby času a paměti, čímž optimalizuje doručování obsahu a šetří využití content delivery network[3] (CDN) služeb. Dále zamezuje nepotřebného vykonávání kódu a v případě využívaní placeného úložiště snižuje náklady za provoz.

S využitím Jetpack Compose předpřipravených komponent a již zmiňované paging knihovny uvedené v kapitole 3.2.2 lze tuto optimalizační techniku snadno implementovat.

4.1.3 Snadné stylování

Aplikování různých stylů v Jetpack Compose je zmíněné v kapitole 3.2.4. Možnost definovat si palety barev a proměnit vzhled celé aplikace změnou jedné proměnné je mnohem snazší, než dosavadní přístup s XML soubory, kde je zvykem pro každý styl definovat nový soubor s jinou paletou barev. Navíc každá Jetpack Compose komponenta definovaná jako Kotlin funkce by měla přijímat parametr `modifier`, který umožňuje měnit vzhled komponenty řetězením posloupností transformačních funkcí. Tento postup je znatelně čitelnější než různá pojmenovaní v XML souborech.

4.1.4 Méně zdrojového kódů

Tím, že máme všechen kód definované komponenty v jednom Kotlin souboru, nemusíme přecházet mezi uživatelským rozhraním a logikou aplikace [12]. Navíc oproti XML přístupu nemusíme vytvářet tolik tříd, které by námi vytvořené rozvržení rozhraní spojovaly dohromady. V Jetpack Compose je vše provázáno za nás, například použitím responzivních komponentů jako je `Column`, `Row` či `Box`, které slouží jako kontejnery pro jednotlivé composables. Programátor se může soustředit na daný problém, přičemž má snazší hledání chyb a lepší možnosti pro otestování aplikace; kolega, který má promyslet a schválit danou změnu, má méně kódu na čtení a pochopení [12].

4.1.5 Architektura

V Compose je nemožné po prvotním vykreslení vynutit překreslení uživatelského rozhraní. Jediné co máme k dispozici je stav komponenty. Kdykoliv se tento stav

[3] CDN je síť distribuovaných serverů, které slouží ke zrychlení a optimalizaci doručování obsahu uživatelům. Hlavním účelem CDN je minimalizovat dobu načítání obsahu.

změní, Compose se postará o překreslení rozhraní. Jelikož komponenty pouze přijímají stav a vystavují události, lze s nimi snadno zařídit návrhový vzor zvaný *unidirectional data flow* [13]. Jak již bylo řečeno v kapitole 3.1, Android Jetpack je sada nástrojů, knihoven a *architektonických vzorů*, kde jeden z doporučených vzorů při vytváření architektury projektu je právě *unidirectional data flow*, o kterém se více pojednává v kapitole 5.3.

4.1.6 Shrnutí

Závěrem lze říci, že Jetpack Compose je vhodnou volbou i pro vývojáře, kteří s vývojem pro Android teprve začínají. Zatímco XML je stále dominantní ve starších projektech, Compose je moderní doporučenou sadou nástrojů pro vytváření nativního uživatelského rozhraní a určitě se ho vyplatí začít používat.

4.2 Nevýhody

V další části textu si popíšeme nevýhody, se kterými se Jetpack Compose momentálně potýká a které je třeba zvážit, pokud se rozhodnete zaměřit na tuto novou technologii.

4.2.1 Dokumentace

Jelikož Jetpack Compose je relativně nová technologie, pravděpodobně se ještě nějakou dobu setkáme s nedostatečnou dokumentací. Navíc při hledání problémů na internetu budeme muset více pátrat, jelikož vývoj Jetpack Compose je velmi rychlý a většině problému nebyla ještě věnována dostatečná pozornost.

4.2.2 Podpora nástrojů

Jednou z největších nevýhod je nedostatečná podpora nástrojů pro vývoj. Příkladem může být požadavek na náhled obrazovky při vytváření uživatelského rozhraní, kdy se změny projevují přímo při změně ve zdrojovém kódu, a to bez nutnosti stahovat celou aplikaci a spouštět emulátor s mobilním zařízením.

Stávající nástroje pro vytváření XML uživatelských rozhraní jsou důležitou součástí při vývoji aplikace, jelikož značně šetří čas programátora. Existují sice nástroje, které se snaží nahradit původní dynamický styl vytváření uživatelského rozhraní, ale bohužel se nedokážou vyrovnat předešlému způsobu, který má za sebou několik let vývoje.

4.2.3 Aktuální stav vývoje

Pokud bychom se chtěli věnovat Androidu profesionálně, mimo Android Jetpack se ještě stále vyplatí naučit pracovat s XML a Views, jelikož většina stávajících aplikací pracuje ještě stále s nimi. Je zřejmé, že Jetpack Compose získává na oblíbenosti, ale většině společnostem se stále ještě nevyplatí přepsat veškeré projekty do Compose a budou nadále používat XML zápis, dokud nebude příliš zastaralý.

4.2.4 Shrnutí

Většina nevýhod Jetpack Compose je důsledkem toho, že tato technologie je stále relativně nová. Můžeme však počítat s tím, že většina těchto problémů se časem vyřeší.

Architektura projektu 5

Architektura projektu v IT je soubor struktur, pravidel, postupů a technologií, které jsou použity při návrhu, implementaci a provozu softwarového systému [14]. Architektura definuje, jak budou jednotlivé komponenty systému navzájem interagovat a jak budou řešeny různé aspekty systému, jako je bezpečnost, výkon, škálovatelnost a udržitelnost.

Architektura projektu v IT je důležitá z následujících důvodů:

Přehlednost a organizace Zajišťuje, že projekt je navržen a organizován logicky a efektivně, což usnadňuje jeho vývoj a správu.

Komunikace mezi týmy Umožňuje lepší komunikaci mezi jednotlivými týmy, které na projektu pracují a zajistí, že všichni rozumí celkovému směru a cílům projektu.

Výkon a efektivita Optimalizuje výkon systému tím, že zajišťuje správné využití zdrojů, což vede k lepšímu výkonu a nižším nákladům na provoz.

Škálovatelnost Umožňuje systému růst a přizpůsobovat se změnám v požadavcích, aniž by bylo nutné provádět zásadní úpravy celého systému [14].

Bezpečnost Bezpečnostní rizika a hrozby jsou důležitým aspektem vývoje IT systémů. Dobře navržená architektura zohledňuje bezpečnostní požadavky a zajišťuje, že systém je odolný vůči různým útokům [15].

Modularita a snadná údržba Rozděluje systém na menší, modulární části, které lze snadno spravovat, aktualizovat a opravovat nezávisle na ostatních částech systému [14].

Snadnější integrace Usnadňuje integraci s dalšími systémy a technologiemi, což zvyšuje flexibilitu celého projektu [16].

Cílem architektury je minimalizovat lidské zdroje, které jsou potřeba k vytvoření a údržbě softwaru [14].

Význam architektury projektu v IT nelze přehlédnout, protože hraje důležitou roli ve všech fázích projektu, od návrhu až po provoz a údržbu. Investice do kvalitní architektury na začátku projektu může přinést významné dlouhodobé výhody, jako je snížení nákladů, zvýšení kvality a zajištění celkového úspěchu projektu.

Vývojáři Android Jetpacku jsou si této skutečnosti vědomi, a proto velkou část úsilí zaměřili právě na snadnou realizaci správné architektury projektu, a to jak pomocí rad v dokumentacích, tak v implementaci jednotlivých technologií, které využíváme v našich aplikacích.

Nyní si popíšeme, které základní principy Android Jetpack doporučuje dodržovat v každém projektu a jakým způsobem je nám pomáhá implementovat.

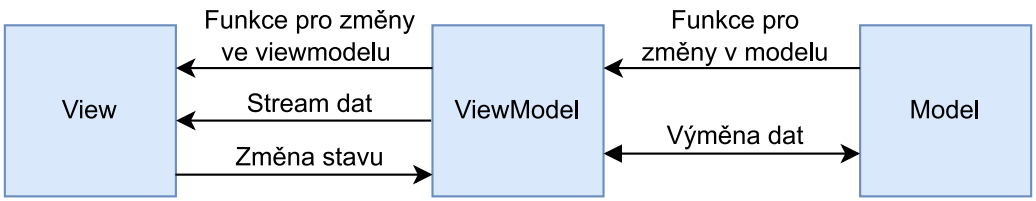
5.1 Separation of concerns

Separation of concerns (SOC) je designový princip, který spočívá v rozdělení programu nebo systému na části, takže každá část se zabývá jedním konkrétním aspektem nebo funkcí. Cílem je dosáhnout modularizace, což vede k jednoduššímu vývoji, údržbě a rozšiřitelnosti softwaru.

Tento koncept byl poprvé představen Edsgerem W. Dijkstrou v roce 1974 [17]. SOC podporuje zapouzdření, kde se každá část (modul, třída nebo funkce) soustředí na svůj vlastní úkol a komunikuje s ostatními částmi prostřednictvím definovaných rozhraní. Tímto způsobem se zvyšuje srozumitelnost a snižuje složitost systému.

Příkladem mohou být třídy založené na uživatelském rozhraní, které by měly obsahovat pouze logiku, která zpracovává interakce s uživatelským rozhraním. Pokud budou tyto třídy co nejjednodušší, můžeme se vyhnout mnoha problémům souvisejících s životním cyklem komponent a zlepšit testovatelnost těchto tříd [18].

V praxi se SOC často používá v rámci návrhových vzorů, architektonických stylů a programovacích paradigmat. Například, v Android vývoji je rozšířený koncept Model-View-Viemodel (MVVM), který rozděluje prezentační vrstvu na tři základní části: model (reprezentace dat a logiky), pohled (prezentace a uživatelské rozhraní) a viewmodel (zprostředkovatel mezi modelem a pohledem) [11] (viz obr. 5.1).



Obrázek 5.1: MVVM architektonický vzor

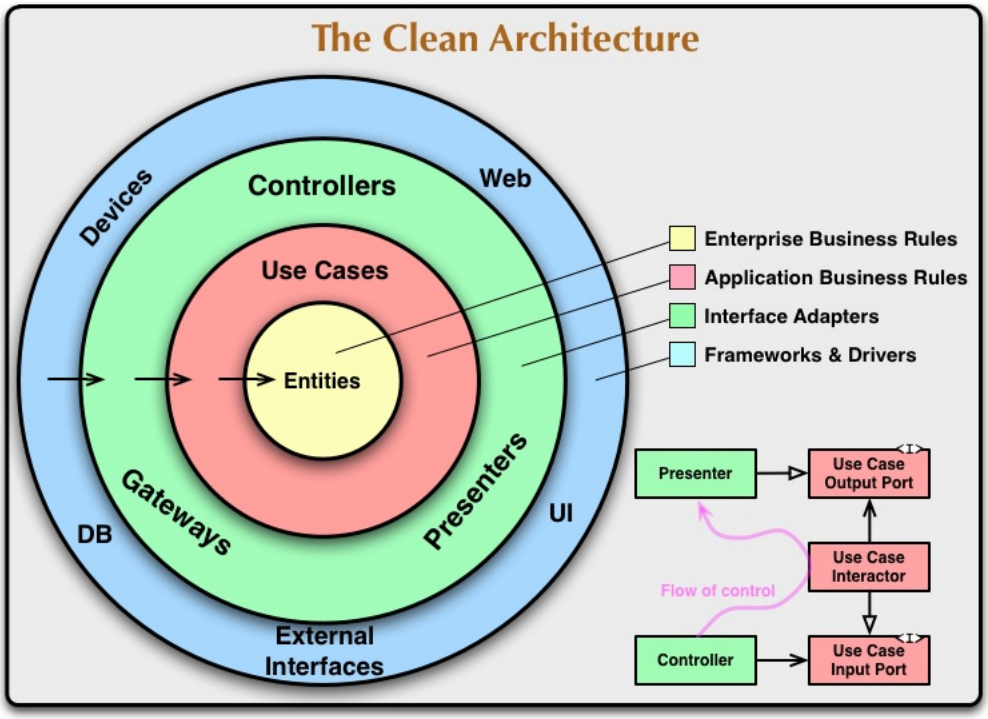
Tímto způsobem je dosaženo lepší organizace kódu a snazšího řešení problémů.

5.1.1 Clean architecture

Clean architecture je důležitou a často zmiňovanou architekturou systému, kterou v roce 2012 představil Robert C. Martin. V posledních několika letech se tato architektura stala velmi oblíbenou a začala se používat v mnoha projektech.

Použitím této architektury můžeme vytvořit systém, který [19]:

- je nezávislý na frameworku,
- má snadno testovatelnou business logiku,
- je nezávislý na uživatelském rozhraní,
- je nezávislý na databázi (obecněji na zdroji dat),
- je nezávislý na vnějším poskytovateli.

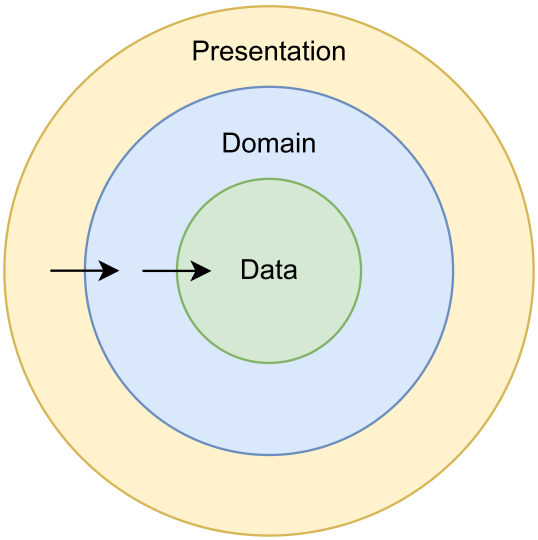


Obrázek 5.2: Diagram jednotlivých vrstev systému [19]

Obr. 5.2 je pokusem o vizualizaci všech těchto podmínek do jednoho smysluplného celku. Soustředné kruhy představují různé oblasti softwaru. Obecně platí, že čím dále jdeme z jádra ven, tím vyšší je úroveň softwaru. Hlavním pravidlem, díky němuž tato architektura funguje, je pravidlo závislosti. Toto pravidlo říká, že závislosti zdrojového kódu mohou směřovat pouze dovnitř. Nic ve vnitřní vrstvě nemůže vědět nic o vnější vrstvě [19].

5.1.1.1 Použití při vývoji Android aplikace

Při vývoji Android aplikace se spíše setkáme s doporučovanou architekturou jako je na obr. 5.3, kde platí stejné principy jako u definice od Robert C. Martina, ale obrázek lépe vystihuje reálnou strukturu projektu [11].

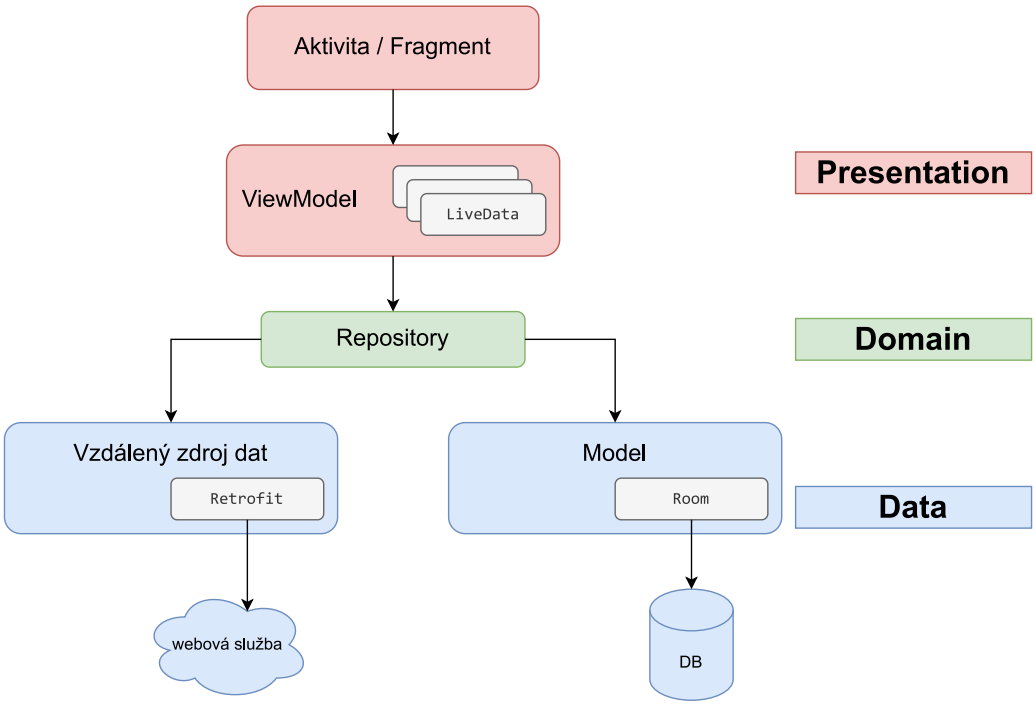


Obrázek 5.3: Diagram doporučených vrstev při vývoji Android aplikace

Jednotlivé vrstvy většinou představují adresáře v kořenovém adresáři aplikace. Obecně chceme tyto tři podadresáře seskupovat pod jeden logický adresář, který představuje nezávislou funkcionalitu aplikace. To může být například obrazovka s přihlášením, která může být využita několika aplikacemi a není tedy závislá pouze na jedné aplikaci. Tyto adresáře se poté nazývají features a umožňují mimo jiné znovu použití zdrojového kódu (viz kapitola 9.1.3 s praktickou ukázkou).

V *presentation* vrstvě se nachází prvky s uživatelským rozhraním, jako aktivity a fragmenty, v případě používaní XML zápisu nebo jednotlivé composables v případě použití Jetpack Compose. Dále se zde nachází některé třídy z Android Jetpack architektury, například `ViewModel` a `LiveData` uvedené v kapitole 3.2.2. *Presentation* vrstva je také místo, kde bychom použili již zmíněný architektonický vzor MVVM [11] (viz obr. 5.1). V *domain* vrstvě se nachází business logika aplikace, tedy nezávislá vrstva, která by měla jít sama o sobě otestovat a neměla by být závislá na žádné vnější knihovně. Nakonec *data* vrstva, která se stará o získávání dat, například ze vzdálené databáze.

Jak již bylo zmíněno v kapitole 3.2, Android Jetpack má komponentu zaměřující se přímo na architekturu. Na obr. 5.4 je obecný příklad, jak Android Jetpack pomáhá realizovat clean architecture, s využitím některých prvků z komponenty architektury.



Obrázek 5.4: Příklad clean architecture Android projektu s využitím Android Jetpack komponenty architektury [20]

`LiveData` je třída, kterou můžeme použít pro uchování a získávání dat, podle kterých se uživatelské rozhraní aktualizuje. Na rozdíl od „běžných“ proměnných, `LiveData` zohledňuje životní cyklus, což znamená, že respektuje životní cyklus ostatních komponent aplikace, jako jsou aktivity, fragmenty nebo služby. Repository obsahuje business logiku aplikace, tedy není zde žádná implementace, ale většinou pouze rozhraní popisující s jakými daty budeme pracovat. Poté v data vrstvě toto rozhraní implementujeme podle námi zvolených technologií, tedy zde knihovna Room, která poskytuje abstraktní vrstvu nad SQLite databází a Retrofit, který představuje snadnou a rychlou knihovnu pro načítání a nahrávání dat prostřednictvím webové služby založené na protokolu REST[1].

5.2 Single source of truth

Když je v aplikaci definován nový datový typ, měli byste mu přiřadit takzvaný single source of truth (SSOT). SSOT je vlastníkem těchto dat a pouze SSOT je může měnit. Aby toho mohl SSOT dosáhnout, musí poskytnout aplikační data pomocí neměnného datového typu[2] a pro jejich modifikaci vystavit funkce nebo přijímat

[1] REST (representational state transfer) je architektura založená na webových standardech a používá protokol HTTP pro výměnu dat mezi klientem a serverem.

[2] Neměnné datové typy jsou objekty, které po vytvoření nelze upravovat ani měnit.

události, které mohou volat jiné části aplikace [18].

Hlavním cílem SSOT je zajistit, aby všechny důležité informace a data byly udržovány na spolehlivém centrálním místě, ze kterého jsou pak přístupné pro všechny relevantní části aplikace.

Tento vzor přináší řadu výhod [18]:

- soustřeďuje všechny změny určitého typu dat na jednom místě,
- chrání data tak, aby s nimi nebylo chybně manipulováno,
- umožňuje lepší dohledatelnost změn v datech, tedy zamezuje případným chybám.

V praxi se SSOT často implementuje prostřednictvím centralizovaných databází, cloudových služeb nebo API, které zajišťují přístup k těmto datům pro různé aplikace a systémy. V případě Android Jetpacku se může jednat o `ViewModel` v prezentační vrstvě či `Room` knihovnu pro práci s databází.

Pokud bychom měli například databázi uživatelů, které chceme zobrazit na obrazovce, náš `ViewModel` by vystavil objekt pro získání dat, jako například neměnná `LiveData`. `LiveData` by obsahovaly načtená data uživatelů z vnějšího zdroje a společně s těmito daty by SSOT, tedy v našem případě `ViewModel`, poskytoval funkce pro změnu a modifikaci uživatelů.

5.3 Unidirectional data flow

Unidirectional data flow (UDF) návrhový vzor se často používá ve spojením se SSOT (viz kapitola 5.2), jelikož jejich implementace mají podobný princip. V UDF proudí stav pouze jedním směrem. Události, které modifikují data, proudí opačným směrem [18].

V systému Android se stavy nebo data obvykle přenášejí z vrstvy s vyšším rozsahem do vrstvy s nižším rozsahem. Události jsou obvykle vyvolány z nižších vrstev, dokud nedosáhnou SSOT, který funkci pro změnu stavu vystavil [11]. Například data aplikace obvykle proudí ze zdrojů dat do uživatelského rozhraní. Uživatelské události, například stisknutí tlačítka, proudí z uživatelského rozhraní do SSOT, kde jsou aplikační data modifikována a opět vystavena v neměnném datovém typu [18].

Tento vzor lépe zaručuje konzistenci dat, je méně náchylný k chybám, snadněji se ladí a přináší všechny výhody ze vzoru SSOT [18].

5.4 Správa závislostí

Dependency injection (DI), či také injektování závislostí, je běžný způsob správy závislostí v aplikacích a neměl by chybět v žádném větším projektu. Závislostí v apli-

kaci se chápe závislost třídy a na třídě B pro správné fungování některé funkcionality. DI umožňuje třídám definovat jejich závislosti bez jejich bezprostřední konstrukce. To znamená, že za vytvoření a dosazení požadované třídy závislé třídě je zodpovědný DI framework, kde programátorovi stačí nějakým způsobem frameworku říct, o kterou třídu se má postarat, například použitím anotace. Tímto způsobem si můžeme navíc objekty snadno uměle vytvořit a otestovat tak správné chování aplikace[3].

DI nám umožňuje v pozdější fázi vývoje snadno rozšířit stávající funkcionalitu, protože zavádí přesný postup jak řešit závislosti, a to bez duplicitního kódu nebo přidávání na složitosti. Tyto vzory navíc umožňují rychlé přepínání mezi testovacím a produkčním prostředím [18].

Při vývoji na platformě Android je od vývojářů Android Jetpacku doporučené používat již zmiňovanou knihovnu Hilt uvedenou v kapitole 3.2.2 pro zavádění závislostí. Hilt automaticky konstruuje objekty procházením stromu závislostí, poskytuje záruku dosazení závislostí při kompilaci a vytváří závislostní kontejnery pro framework třídy [18].

Použitím DI redukujeme boilerplate kód a vytváříme aplikace, která lze mnohem snáze otestovat [11].

5.5 Shrnutí

Android Jetpack se snaží vývojářům usnadnit vývoj, jak už poskytnutím kvalitní implementace knihoven, tak rady k vytvoření a údržbě projektu pomocí správné architektury a dekompozice známých problémů. Výše zmíněné principy jsou pouze doporučené zásady, které jsou všeobecně známé a podporované. Android Jetpack však poskytuje kvalitní dokumentaci a rady k řadě knihovnám, které jdou mnohem více do hloubky, jelikož už řeší konkrétní problémy.

[3]Tento proces nám umožňuje izolovat a testovat kód bez jakýchkoli zásahů závislostí a dalších proměnných, jako jsou problémy se sítí a kolísání provozu.

Funkce aplikace 6

Tento projekt má za cíl prezentovat Android Jeptack jako efektivní nástroj pro tvorbu aplikací na platformě Android. Pro vytvoření uživatelského rozhraní se používá Jetpack Compose a celý projekt je napsán v jazyce Kotlin, přičemž pro veškeré potřebné knihovny byly použity aktuální verze, aby aplikace demonstrovala aktuální možnosti vývoje.

Aplikace by měla uživatelům poskytnout jednotný nástroj pro hledaní a publikaci ztracených předmětů. Uživatelé si mohou prohlížet seznam publikovaných předmětů a reagovat na příspěvky od ostatních uživatelů. Ztracené předměty lze označit na mapách a ostatní uživatelé poté mohou vyhledat lokaci nalezeného předmětu, společně s ostatními informacemi.

U aplikace jsou zajímavé především následující použité technologie:

- protokol OAuth 2.0 pro autentizaci a autorizaci uživatelů,
- Firestore real-time NoSQL (not only structured query language) databáze hostovaná v cloudu,
- Firebase Cloud Storage pro ukládání aplikačních dat jako jsou například obrázky předmětů a profily uživatelů,
- Google Maps SDK pro integraci Google map do aplikace,
- Firebase Authentication pro definovaní pravidel mezi výše zmíněnými technologiemi a zajištění správného manipulovaní s uživatelskými daty.

V aplikaci jsou použité i další zajímavé technologie, jako například lokální alternativa databáze pro ukládání preferencí uživatele.

Některé představené služby jsou placené, a proto před publikací aplikace je důležité s tím počítat. Jedná se především o Firebase Cloud Storage, Firestore databázi, Firebase Authentication a Google Maps SDK.

6.1 Přidání a správa příspěvků

Tato funkce aplikace slouží pro uživatele, kteří nalezený předmět chtějí zveřejnit. Na obr. 1 je obrazovka, kterou každý uživatel uvidí při pokusu zveřejnit jejich první příspěvek. Obrazovka s formulářem pro přidání nového příspěvku (viz obr. 2) obsahuje textové pole pro nadpis a popis předmětu a obrázek předmětu, která jsou vždy nutná vyplnit a volitelnou možnost označit předmět na mapě (viz obr. 3). Po vyplnění formuláře (viz obr. 4) je uživatel přesměrován na obrazovku s přehledem jeho příspěvků (viz obr. 5), kde své příspěvky může spravovat (viz obr. 6).

6.2 Seznam ztracených předmětů

Tato funkce aplikace slouží pro uživatele, kteří hledají ztracený předmět a způsob jak ho získat zpět. Domovská obrazovka aplikace (viz obr. 7) zobrazuje seznam ztracených předmětů, seřazený od nejnovějších po nejstarší. Důležitou součástí této funkcionality je možnost hledat v potencionálně velkém seznam předmětů (viz obr. 8) kde každé hledané slovo rozšiřuje množinu zobrazených příspěvků. Pokud uživatel najde hledaný předmět, může si zobrazit dialog s informacemi, jak kontaktovat osobu, která předmět nalezla (viz obr. 9). Nedílnou součástí seznamu příspěvků je možnost zobrazit si detail ztraceného předmětu (viz obr. 10).

6.3 Mapy

Funkce aplikace poskytující interaktivní mapu v případě, že uživatel přibližně ví, kde hledaný předmět ztratil a přeji si podívat se, zda někdo předmět již v dané lokaci nenašel a neoznačil. V tomto případě se jedná pouze o jednu obrazovku s mapou a vyhledávacím polem (viz obr. 11).

6.4 Profil

Funkce pro správou přihlášeného uživatele, kde vidí základní přehled osobních informací (viz obr. 12) a kde si může například změnit profilový obrázek, přidat telefonní číslo (viz obr. 13) nebo se odhlásit.

6.5 Nastavení

Nastavení osobních preferencí uživatele jako je jazyk nebo motiv aplikace. Obrazovka má formu jednoduchého seznamu (viz obr. 14).

6.6 Navigace

Uživatel má k dispozici tři důležité navigační prvky. První je spodní panel s primární navigací mezi obrazovky, druhý je horní panel s aktuálním popisem obrazovky a třetí je boční navigační panel (viz obr. 15).

Injektování závislostí 7

V aplikaci používáme doporučovanou knihovnu Dagger Hilt pro injektování závislostí (DI). Ukážeme si, jak zaregistrovat třídu/rozhraní do závislostí aplikace, a jak danou třídu/rozhraní následně použít.

7.1 Příprava frameworku

První nutnou podmínkou je vytvoření souboru `LostAndFoundApp.kt` s definicí vstupního bodu pro knihovnu Hilt:

```
1 @HiltAndroidApp
2 class LostAndFoundApp : Application()
```

Zdrojový kód 7.1: Souboru `LostAndFoundApp.kt` s definicí vstupního bodu pro knihovnu Hilt

Soubor `LostAndFoundApp.kt` se zdrojovým kódem 7.1 si uložíme do adresáře popsaného v kapitole 9.1. Použitím této anotace si framework vytvoří speciální komponentu spjatou s životním cyklem celé aplikace, která zařídí injektování všech závislostí v naší aplikaci.

Poté v manifestu aplikace definujeme vstupní bod aplikace dle nově vytvořeného souboru (viz zdrojový kód 7.2):

```
1 <application
2   android:name=".LostAndFoundApp"
3   ...
4 >
5 ...
6 </application>
```

Zdrojový kód 7.2: Definování vstupního bodu aplikace v manifestu aplikace

Jako poslední je nutné použít anotaci `@AndroidEntryPoint` z knihovny Hilt na hlavní aktivitu naší aplikace (viz zdrojový kód 7.3). Tato anotace se používá na třídy, pro které chceme vygenerovat individuální Hilt komponentu, která umožňuje do dané třídy injektovat závislosti.

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
 //...
}
```

Zdrojový kód 7.3: Anotace hlavní aktivity pro injektování závislostí

Tyto definice je důležité definovat pro používaný framework a dále s nimi již pracovat nebudeme.

7.2 Modul s třídou

Nejběžnější způsob injektovaní závislostí je u běžných tříd, které stačí jednou vytvořit, a poté předat ostatním částem aplikace. Příkladem může být instance databáze (viz zdrojový kód 7.4):

```
@Module
@InstallIn(SingletonComponent::class)
object DbModule {
    @Provides
    @Singleton
    fun provideFirestoreDb(): FirebaseFirestore = Firebase.firestore
}
```

Zdrojový kód 7.4: Modul s injektovanou databází

Anotace `@InstallIn` se vstupním parametrem `SingletonComponent` zajistí vytvoření a uchování jedné instance s databází po celou dobu fungování aplikace. Návratová hodnota metody `provideFirestoreDb` určuje jaký objekt bude injektován, tedy v tomto případě budeme chtít všechny místa se závislostí na třídě `FirebaseFirestore` injektovat hodnotou `Firebase.firestore`.

7.3 Modul s rozhraním

V aplikaci také můžeme injektovat implementace různých rozhraní (viz zdrojový kód 7.5):

```
@Module
@InstallIn(SingletonComponent::class)
abstract class LostItemsRepositoryModule {
    @Binds
    @Singleton
    abstract fun bindLostItemRepository(
        lostItemRepositoryImpl: LostItemRepositoryImpl,
    ): LostItemRepository
}
```

Zdrojový kód 7.5: Modul s injektovaným rozhraním

Pokud poté v aplikaci použijeme rozhraní `LostItemRepository`, bude na dané místo injektovaná implementace daná třídou `LostItemRepositoryImpl`, která dědí rozhraní `LostItemRepository` a implementuje požadované metody.

7.4 Použití závislostí

Injektovat závislosti lze třemi způsoby:

Constructor injection Závislosti jsou injektovány prostřednictvím konstruktoru třídy. Tento přístup je preferovaný, protože jednotlivé závislosti jsou uvedeny v konstruktoru a nemohou být změněny po vytvoření objektu.

Property injection Závislosti jsou injektovány do některé proměnné definované v těle třídy nebo metody. Používá se hlavně v případě, že závislost je volitelná.

Method injection Závislosti jsou injektovány prostřednictvím volání specifických metod třídy. Tento přístup je méně běžný, ale může být vhodný v případech, kdy je nutné injektovat závislosti až v průběhu životního cyklu objektu nebo když je potřeba injektovat různé implementace stejného rozhraní.

V našem případě budeme používat pouze constructor injection (viz zdrojový kód 7.6):

```
@HiltViewModel
class LostItemViewModel @Inject constructor(
    private val dbRepo: LostItemRepository,
) : ViewModel() {
 //...
}
```

Zdrojový kód 7.6: `ViewModel` s injektovanými závislostmi

Anotace `@Inject` značí injektovaní závislostí pomocí konstruktoru, framework se poté postará o injektovaní potřebných závislostí. Anotace `@HiltViewModel` slouží pro injektování daného ViewModelu do uživatelského rozhraní (viz zdrojový kód 7.7):

```
@Composable
fun CreateLostItemForm(
    viewModel: LostItemViewModel = hiltViewModel(),
) {
 // ...
}
```

Zdrojový kód 7.7: Injektování třídy `ViewModel` do rozhraní definovaného v Jetpack Compose

V tomto případě injektujeme `ViewModel` volaním metody `hiltViewModel` poskytované z knihovny Hilt.

Ukládání dat 8

V této kapitole se zaměříme na to, jak v aplikaci řešíme uchovávání aplikačních a uživatelských dat. Dále se také zmíníme o způsobu ochrany uložených dat a jakým způsobem data zpracováváme v jednotlivých částech aplikace, tedy od datové vrstvy až po prezentační vrstvu s uživatelským rozhraním.

8.1 Autentizace a autorizace uživatelů

Důležitou součástí aplikace je zajištění ochrany dat. Aplikace musí mít ošetřené, aby nikdo jiný než vlastník příspěvku nemohl daná data upravovat a aby osobní informace jednotlivých uživatelů byly zabezpečené. K tomu byl použit protokol OAuth 2.0 poskytovaný službou Firebase Authentication, která umožňuje uživatelům přihlásit se více způsoby, například pomocí Google Gmail účtu nebo běžným mailem a heslem (viz obr. 16).

Hlavní výhodou využití již hotové služby je že se nemusíme starat o ukládání citlivých dat a implementaci logiky přístupu ke zdrojům jako je cloudová databáze s daty uživatelů.

Aplikace používá následující množinu pravidel pro práci s cloudovou databází[1]:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, update, delete: if request.auth != null
   && request.auth.uid == userId;
      allow create: if request.auth != null;
    }
  }
}
```

Proměnná `request.auth` obsahuje autentizační informace klienta, který žádá o data. Tímto způsobem máme v aplikaci propojené veškeré služby poskytované od Firebase.

[1] Syntaxe pravidel je dána použitou službou.

8.2 Firestore

Firebase Firestore, oficiálně nazývaný Cloud Firestore, je flexibilní a škálovatelná cloudová databáze ve formě NoSQL, která je součástí Google Firebase platformy. Je navržena tak, aby poskytovala snadnou synchronizaci dat mezi různými zařízeními.

Hlavním důvodem použití této služby byla možnost sledovat real-time změny dat. Při použití aplikace sice není třeba reagovat na real-time změny, ale bude přínosné z hlediska zaměření této práce si vyzkoušet, jak lze s použitím Android Jetpacku implementovat tuto funkcionalitu.

Jako první si v ukázce kódu 8.1 vytvoříme požadavek na propagaci real-time dat do naší aplikace[2]:

```
class LostItemApi @Inject constructor(
    db: FirebaseFirestore,
) {
    private val collectionRef = db.collection(
        LOST_ITEM_COLLECTION_KEY
    )

    private suspend fun fetchLostItemListFlow(
        query: Query
    ): Flow<LostItemListDto> {
        return withContext(Dispatchers.IO) {
            return@withContext query
                .snapshots()
                .map { snapshot ->
                    val lostItems = mutableListOf<LostItemDto>()
                    if (!snapshot.isEmpty) {
                        val documents = snapshot.documents
                        for (document in documents) {
                            document.toObject(
                                LostItemDto::class.java
                            )?.let { lostItems.add(it) }
                        }
                    }
                    LostItemListDto(lostItems)
                }
        }
    }
}
```

Zdrojový kód 8.1: Ukázka implementace propagování real-time dat z Firestore databáze

Proměnná `db` představuje referenci na injektovanou Firestore databázi. Metoda `fetchLostItemListFlow` se zavoláním metody `snapshots` zaregistruje k odběru změn v databázi podle předaného požadavku v parametru `query`. Tímto způsobem jsme vytvořili spojení mezi naší aplikací a cloudovou databází. Zavoláním

[2] Pro pochopení ukázky kódu se předpokládá znalost Kotlin korutin.

metody `fetchLostItemListFlow` získáme třídu `Flow` s DTO[3] (data transfer object) ztracených předmětů, kterou používá prezentační vrstva aplikace.

V prezentační vrstvě aplikace poté můžeme na real-time změny reagovat použitím třídy `MutableStateFlow` (viz zdrojový kód 8.2):

```
@HiltViewModel
class LostItemViewModel @Inject constructor(
    private val dbRepo: LostItemRepository,
    private val storageRepo: ImageStorageRepository,
    private val authRepo: AuthRepository,
) : ViewModel() {

    private val _lostItemListState =
        MutableStateFlow<Response<LostItemList>>(Success(null))
    val lostItemListState = _lostItemListState.asStateFlow()


    private fun fetchLostItems(
        repoCall: suspend () -> Response<Flow<LostItemList>>
    ) {
        viewModelScope.launch {
            _lostItemListState.update { Loading }
            when (val apiFlowResponse = repoCall()) {
                is Error -> _lostItemListState.update {
                    Error(apiFlowResponse.error)
                }
                is Success -> {
                    if (apiFlowResponse.data != null) {
                        try {
                            apiFlowResponse.data
                                .collect { lostItemList ->
                                    _lostItemListState.update {
                                        Success(lostItemList)
                                    }
                                }
                        } catch (e: Exception) {
                            _lostItemListState.update {
                                Success(null)
                            }
                            Log.e( /* ... */)
                        }
                    } else {
                        _lostItemListState.update { Success(null) }
                    }
                }
                Loading -> { /* ... */ }
            }
        }
    }
}
```

Zdrojový kód 8.2: Ukázka propagace real-time dat do prezentační vrstvy aplikace

[3] DTO je návrhový vzor používaný k přenosu dat mezi vrstvami aplikace, tedy v našem případě mezi datovou a business/prezentační vrstvou.

Privátní proměnná `_lostItemListState` slouží pro uchování dat a proměnná `lostItemListState` pro read-only přístup v jednotlivých Jetpack Compose komponentách. Tímto způsobem zajišťujeme SSOT a UDF. Metoda `repoCall` předaná jako parametr do `fetchLostItems` volá metody z ukázky kódu 8.1.

V Jetpack Compose komponentě poté přistupujeme k real-time datům tímto způsobem (viz zdrojový kód 8.3):

```
val lostItemListResponse: Response<LostItemList> = lostItemViewModel
 .lostItemListState
 .collectAsStateWithLifecycle()
 .value
```

Zdrojový kód 8.3: Použití třídy `MutableStateFlow` v Jetpack Compose komponentě

Tímto jsme z datové vrstvy získali real-time data do uživatelského rozhraní, vytvořeného v Jetpack Compose, které respektuje životní cyklus aplikace.

8.3 Firebase Storage

Firebase Storage umožňuje ve velkém měřítku ukládat a sdílet soubory, jako jsou obrázky, videa, audia a další binární data. V aplikaci je použit primárně na ukládání profilových obrázků a fotek ztracených předmětů.

Úložiště má podobu adresářové struktury. Jednotlivé složky tedy představují různá úložiště naší aplikace. Jako první je třeba definovat si v datové vrstvě API pro ukládání obrázků (viz zdrojový kód 8.4):

```
class ImageStorageApi(
    storage: FirebaseStorage,
) {
    private val imagesRef = storage.reference.child(
        IMAGES_KEY
    )

    suspend fun addImageToFirebaseStorage(
        imageUri: Uri,
        name: String,
    ): Uri {
        return withContext(Dispatchers.IO) {
            return@withContext imagesRef
                .child("$name.jpg")
                .putFile(imageUri)
                .await()
                .storage
                .downloadUrl
                .await()
        }
    }
}
```

Zdrojový kód 8.4: Ukázka API pro ukládání obrázků do Firebase Storage úložiště

Proměnná `IMAGES_KEY` představuje jméno adresáře s obrázky. Navracená `Uri` třída z metody `addImageToFirebaseStorage` obsahuje veřejný odkaz pro přístup k uloženému obrázku.

Nyní si definujeme třídu, která se bude volat v naší prezentační vrstvě (viz zdrojový kód 8.5):

```
@Singleton
class ImageStorageRepositoryImpl @Inject constructor(
    private val api: ImageStorageApi
) : ImageStorageRepository {

    override suspend fun addImageToStorage(
        imageUri: Uri,
        name: String,
    ): Response<Uri> {
        return try {
            val downloadUrl = api.addImageToFirebaseStorage(
                imageUri = imageUri,
                name = name
            )
            Success(downloadUrl)
        } catch (e: Exception) {
            Error(e)
        }
    }
}
```

Zdrojový kód 8.5: Implementace rozhraní pro prezentační vrstvu pracující s úložištěm

Jak bylo vidět v ukázce kódu 8.2, prezentační vrstvy pracují s repozitáři jako `ImageStorageRepository`, `AuthRepository` nebo `LostItemRepository`.

8.4 Proto DataStore

Proto DataStore je nové a preferované řešení, které má nahradit SharedPreferences pro ukládání lokálních dat. Může se například jednat o aktuální motiv či jazyk aplikace na daném zařízení. Tato implementace vyžaduje použití serializovatelných objektů a poskytuje typovou bezpečnost.

V našem případě budeme chtít uložit lokální nastavení aplikace (viz zdrojový kód 8.6):

```
@Serializable
data class AppSettings(
    val theme: ThemeOptions = ThemeOptions.SYSTEM,
    val language: LanguageOptions = LanguageOptions.ENGLISH
)
```

Zdrojový kód 8.6: Přepravka s nastavením aplikace

Dále si v ukázce kódu 8.7 vytvoříme třídu pro serializaci a deserializaci nastavení aplikace, která implementuje rozhraní `Serializer` z Proto DataStore:

```
object AppSettingsSerializer : Serializer<AppSettings> {
    override val defaultValue: AppSettings
        get() = AppSettings()

    override suspend fun readFrom(
        input: InputStream,
    ): AppSettings {
        return try {
            Json.decodeFromString(
                deserializer = AppSettings.serializer(),
                string = input.readBytes().decodeToString()
            )
        } catch (e: SerializationException) {
            defaultValue
        }
    }

    override suspend fun writeTo(
        settings: AppSettings, output: OutputStream,
    ) {
        output.write(
            Json.encodeToString(
                serializer = AppSettings.serializer(),
                value = settings,
            ).encodeToByteArray()
        )
    }
}
```

Zdrojový kód 8.7: Metody pro serializaci a deserializaci třídy s nastavením

Jako poslední si v ukázce kódu 8.8 zavedeme instanci Proto DataStore do závislostí v naší aplikaci:

```
@InstallIn(SingletonComponent::class)
@Module
object DataStoreModule {

    @Singleton
    @Provides
    fun provideProtoDataStore(
        @ApplicationContext appContext: Context
    ): DataStore<AppSettings> {
        return DataStoreFactory.create(
            serializer = AppSettingsSerializer,
            produceFile = {
                appContext.dataStoreFile(DATASTORE_FILE_NAME)
            },
            scope = CoroutineScope(Dispatchers.IO + SupervisorJob()),
        )
    }
}
```

Zdrojový kód 8.8: Zavedení instance Proto DataStore do injektovaných závislostí

K serializaci je použita knihovna `kotlinx-serialization-json` pracující s formátem JSON (JavaScript object notation), který následně převedeme na řetězec znaků pro finální serializaci. Tímto způsobem máme zajištěné zachování dat pro dané zařízení a ostatní části aplikace poté mohou jednoduše přistupovat k aktuálnímu nastavení.

Struktura a řešení aplikace

9

Kapitola popisuje jakým způsobem byl projekt strukturován, jak byly řešené některé určité problémy a měla by zodpovědět často kladené otázky spojené s vývojem na platformě Android.

9.1 Struktura projektu

Projekt dodržuje již zmiňovaný architektonický design zvaný clean architecture (viz kapitola 5.1.1). První si ukážeme zjednodušený pohled na strukturu projektu:

```
app.src.main.java.cz.zcu.students.lostandfound
├── common
│   ├── components
│   ├── constants
│   ├── extensions
│   └── util
├── di
├── features
│   └── profile
│       ├── data
│       ├── domain
│       └── presentation
├── navigation
│   ├── App.kt
│   └── Navgraph.kt
├── ui
│   └── theme
├── LostAndFoundApp.kt
└── MainActivity.kt
```

Primární adresáře *common*, *di*, *features*, *navigation* a *ui* si nyní rozebereme.

9.1.1 Adresář common

Jedná se o sdílený adresář, který obsahuje konstanty a části aplikace, které se vyskytují na více místech. Je zde například Jetpack Compose komponenta `ProgressBar`, která představuje znovupoužitelnou komponentu používanou pro indikaci načítaní dat ze sítě. Také se zde nachází různá rozšíření používaných knihoven, například rekurzivní rozšíření třídy `Context` pro hledání aktivity (viz zdrojový kód 9.1):

```
fun Context.findActivity(): ComponentActivity? = when (this) {
    is ComponentActivity -> this
    is ContextWrapper -> baseContext.findActivity()
    else -> null
}
```

Zdrojový kód 9.1: Rekurze hledající aktivitu třídy Context

Dále se zde nachází funkce pro práci s časem a různé obecné pomocné třídy jako třída `Response` pro určení aktuálního stavu požadavku ze sítě.

9.1.2 Adresář di

Adresář obsahuje jednotlivé definované závislostní moduly aplikace (viz ukázka kódu 7.4).

9.1.3 Adresář features

Adresář s primární funkcionalitou aplikace. Jednotlivé podadresáře představují nezávislou funkcionalitu aplikace:

```
features
├── about_app
├── lost_items
├── profile
└── settings
```

Tento přístup nám umožňuje použít dané části aplikace v úplně jiné aplikaci s minimálním zásahem do zdrojového kódu. Například `profile` obsahuje funkce pro práci s Firestore Authentication službou a obrazovku s profilem uživatele. Lze si tedy snadno představit, proč je tato část aplikace nezávislá na zbytku aplikace.

Každý podadresář dodržuje stejnou strukturu:

```
some_feature
├── data
├── domain
└── presentation
```

Jednotlivý význam těchto tří adresářů je popsán v kapitole 5.1.1.

9.1.4 Adresář navigation

Speciální adresář, který obsahuje veškeré navigační prvky aplikace a spojuje je do jedné třídy `App` volané z hlavní aktivity aplikace (viz ukázka kódu 9.2).

V adresáři se nachází definice obrazovek, mapovaní obrazovek na jejich názvy a především třída `NavHost`, která zajišťuje navigaci mezi jednotlivými obrazovkami aplikace. Jednotlivé obrazovky jsou definované jako url cesty a mají i stejný způsob předávání parametrů.

9.1.5 Adresář ui

Adresář obsahuje definice stylů uživatelského rozhraní, tedy světlý a tmavý motiv, velikosti rozestupů mezi jednotlivými prvky rozhraní či definice typografie.

Dále je zde definovaná třída `LostAndFoundTheme`, která zabalí již zmiňovanou třídu `App` (viz kapitola 9.1.4) do jednotného stylu:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        handleSplashScreen()
        setContent {
            LostAndFoundTheme {
                App()
            }
        }
    }
    // ...
}
```

Zdrojový kód 9.2: Kompletní definice hlavní aktivity

Tímto při změně v `LostAndFoundTheme`, například změna motivu ze světlého na tmavý, bude změna propagována do všech prvků rozhraní.

9.2 Řešení problémů

Tato kapitola bude pojednávat o různých problémech, které se při vývoji vyskytly a jakým způsobem byly řešeny.

9.2.1 Práce s real-time daty

Práce s real-time daty byla určitě jedna z nejkomplikovanější částí celé práce. Hlavní problém spočíval ve zprovoznění spojení mezi aplikací a Firestore databází. K implementaci bylo třeba přečíst velké množství dokumentace a projít několik online zdrojů. Jak lze vidět v kapitole 8.2, proces získání dat až po předaní Jetpack Compose komponentě je poměrně komplexní.

K implementaci bylo nutné nastudovat si třídu `Flow` ze standardní knihovny Kotlinu, která slouží pro asynchronní vysílaní dat. Jelikož se navíc jedná o přenos dat po síti, je nutné aby aplikace reagovala na všechny možné stavy, tedy stav, kdy se data načítají, stav kdy došlo k nějaké chybě nebo stav kdy někdo změnil nějaký popisek a danou změnu potřebujeme propagovat do uživatelského rozhraní. Pokud k tomu navíc přidáme možnost data filtrovat a zasazení všech těchto souvislostí do clean architecture designu, vznikne tak poměrně složitý úkol.

9.2.2 Konkurentní programování

Jelikož většinu dat načítáme z cloudových úložišť, je potřeba, aby na to uživatelské prostředí správně reagovalo a informovalo uživatele o aktuálním dění v aplikaci. Ke správnému fungování byla potřeba pochopit, jak funguje konkurentní programování v Kotlinu a vytvořit si pomocnou třídu `Response`.

9.2.2.1 Korutiny

První důležitou podmínkou je pochopení korutin v Kotlinu. Korutiny jsou nenáročná, vysokoúrovňová primitiva konkurentního programovaní v Kotlinu, které umožňují psát asynchronní kód. Jsou navrženy tak, aby byly efektivnější než tradiční vlákna. Korutiny umožňují psát kód, který vypadá jako sekvenční, ale ve skutečnosti se provádí asynchronně.

V ukázkách kódu se často vyskytuje volání metody `withContext`, která přijímá jako parametr konstantu ze třídy `Dispatchers`. To jakou konstantu zvolíme může výrazně ovlivnit chování naší aplikace. Volba konstanty určuje, které vlákno nebo vlákna budou provádět kód námi definované korutiny, přičemž je na nás rozhodnout, kam jaká operace patří. Pokud se například jedná o požadavek ze sítě, použili bychom konstantu `Dispatchers.IO`, naopak v případě filtrování dat na frontendu bychom použili konstantu `Dispatchers.Default`. Pokud bychom neřešili v jakém vlákně se korutiny volají, mohlo by například dojít k zamrznutí uživatelského rozhraní.

9.2.2.2 Pomocná třída Response

Třída `Response` uchovává aktuální stav požadavků ze sítě a je použita napříč mnoha viewmodely aplikace. Třída může nabývat tří stavů (viz ukázka kódu 9.3):

Loading Aplikace čeká na odpověď nebo právě načítá data ze zdroje.

Success Data jsou načtená a připravená pro čtení z proměnné `data`.

Error Při zpracování požadavku nastala chyba, kterou si lze přečíst z proměnné `error` a pokud se podařilo získat alespoň část načítaných dat, lze je získat

z proměnné `data`. Proměnná `data` lze případně použít pro předaní specifických dat při výskytu chyby.

Třída `Response`:

```
sealed class Response<out T> {
    object Loading: Response<Nothing>()

    data class Success<out T>(
        val data: T?
    ) : Response<T>()

    data class Error<out T>(
        val error: Exception,
        val data: T? = null
    ) : Response<T>()
}
```

Zdrojový kód 9.3: Pomocná třída pro reprezentaci aktuálního stavu požadavku ze sítě

Uživatelské rozhraní pak může na stavy patřičně reagovat, například při stavu `Loading` vykreslit prvek značící načítaní nebo v případě stavu `Error` zobrazit hlášku s informací o chybě.

9.3 Volba úrovně Android API

Jelikož by měla aplikace demonstrovat co nejmodernější přístup, byla zvolená poměrně vysoká verze minimální API úrovně, a to API verze 31, přičemž aplikace je primárně vyvíjena a testována na API verzi 33.

Tabulka 9.1: Distribuce verzí platformy Android a podpora zařízení [21]

Verze Androidu	Kód verze	Úroveň API	Podporovaná zařízení
5.0	Lollipop	21	99.3%
5.1	Lollipop	22	99.0%
6.0	Marshmallow	23	97.2%
7.0	Nougat	24	94.4%
7.1	Nougat	25	92.5%
8.0	Oreo	26	90.7%
8.1	Oreo	27	88.1%
9.0	Pie	28	81.2%
10.0	Q	29	68.0%

(tabulka pokračuje na další stránce)

Tabulka 9.1 *(pokračování z předchozí stránky)*

Verze Androidu	Kód verze	Úroveň API	Podporovaná zařízení
`12.0`	R	`30`	`48.5%`
`12.0`	S	`31`	`24.1%`
`13.0`	Tiramisu	`33`	`5.2%`

Tab. 9.1 byla vytvořena na základě udržované tabulky [21] od Android vývojářů ze dne 6. ledna 2023.

Podle údajů z tab. 9.1 by naše aplikace měla fungovat na zařízeních od Android verze `12.0`, tudíž kolem `24.1%` všech Android zařízení by mělo být schopno aplikaci používat bez problémů.

9.4 Distribuce

Aplikaci lze distribuovat prostřednictvím APK (Android package kit) souboru, který získáme dle návodu v instalační příručce v příloze.

9.5 Síťové připojení

Aplikace lze používat i bez připojení k internetu. Jedinou nutnou podmínku je, že uživatel nebyl při předešlém používání aplikace odhlášen. Uživatel může v offline režimu prohlížet a upravovat příspěvky, které byli načtené při posledním připojení aplikace k síti. V případě, že uživatel provede změny příspěvku v offline režimu, při dalším připojení k sítí se automaticky aktualizují, takže se nemusí bát, že by nějaká data ztratil.

Testování 10

Testování aplikace je rozplánováno do tří fází. První fáze testování spočívá ve statické analýze a inspekci kódů nástroji z Android Studia, a to především s použitím linteru[1] a profileru. Druhá fáze se zabývá implementací unit a integračních testů společně s vytvořením a vyzkoušením testovacích scénářů. Třetí fáze spočívá v nastavení služby pro sledování stavů a výpadků aplikace a procházení seznamu kvalit aplikace [22], kde projekt následně prošel různými změnami, aby splňovala určité standardy kladené na Android aplikace.

Použitím clean architecture designu a DI byla vytvořena aplikace, která by měla jít snadno otestovat unit a integračními testy. V této kapitole budou ukázány výhody, které tento přístup přináší.

Dále bude popsána druhá a třetí fáze testování, tedy některé příklady jak psát unit a integrační testy, jak zaznamenávat výpadky a jak aplikace obstála v celkovém hodnocení kvalit Android aplikace.

10.1 Unit testy

Unit testy by měly být jednoduché a rychlé, jelikož testují jednu funkcionalitu aplikace. Příkladem může být filtrování ztracených předmětů v uživatelském rozhraní. Pokud bychom se řídili clean architecture designem, tato funkcionalita by se měla nacházet v business logice aplikace, tedy v adresáři domain. V tomto případě bude cílem unit testu otestovat správnost vrácených výsledků při různých vstupech.

Pokud bychom chtěli otestovat, jak se zachová uživatelské rozhraní po odeslání požadavku na vyfiltrování ztracených předmětů, jednalo by se o integrační nebo end-to-end [2] test.

Pokud se tedy vrátíme k otestovaní filtrování ztracených předmětů, můžeme například otestovat třídu `SearchTermValidator`, která slouží pro validaci filtro-

[1] Linter je nástroj, který analyzuje zdrojový kód a hledá v něm potencionální chyby, které následně pomáhá opravit.

[2] End-to-end testy si můžeme představit jako komplexnější integrační testy, které simulují různé uživatelské akce.

vaných slov. Jako první si musíme připravit testovanou funkcionalitu (viz zdrojový kód 10.1):

```
class SearchTermValidatorTest {

    private lateinit var searchTermValidator: SearchTermValidator

    @Before
    fun setUp() {
        searchTermValidator = SearchTermValidator()
    }

    // ... unit tests
}
```

Zdrojový kód 10.1: Příprava testované funkcionality pro unit testy

Poté již stačí napsat jednotlivé testy, které kontrolují různé případy vstupů (viz zdrojový kód 10.2):

```
@Test
fun `Term with length smaller than MAX_TERM_LENGTH, is valid`() {
    val searchTerm = "a".repeat(MAX_TERM_LENGTH - 1)
    val isValid = searchTermValidator.validate(searchTerm)
    assertThat(isValid).isTrue()
}

@Test
fun `Term with new line, is invalid`() {
    val searchTerm = "a".repeat(MAX_TERM_LENGTH - 2) + "\n"
    val isValid = searchTermValidator.validate(searchTerm)
    assertThat(isValid).isFalse()
}

@Test
fun `Term with MAX_TERM_LENGTH length, is valid`() {
    val searchTerm = "a".repeat(MAX_TERM_LENGTH)
    val isValid = searchTermValidator.validate(searchTerm)
    assertThat(isValid).isTrue()
}

@Test
fun `Empty term, is invalid`() {
    val searchTerm = ""
    val isValid = searchTermValidator.validate(searchTerm)
    assertThat(isValid).isFalse()
}
```

Zdrojový kód 10.2: Jednotlivé unit testy pro třídu `SearchTermValidator`

Unit testy by se tedy měly především zaměřovat na testování naší business logiky aplikace a bývá jich z pravidla nejvíce.

10.2 Integrační testy

Integrační testy jednoduše testují funkcionalitu více tříd. Příkladem může být ověření funkčnosti `ViewModel` třídy s daty pro uživatelské rozhraní a komponenty v Jetpack Compose pracující s těmito daty. V tomto případě by se jednalo o takzvaný instrumented test, pro který je potřeba mít funkční emulátor či fyzické zařízení, tedy celý proces trvá o poznání déle, než běžné testy běžící pouze na JVM (Java virtual machine).

Dále si zkusíme ověřit správné chování uživatelského rozhraní s nastavením naší aplikace. Jako první si podobně jako u ukázky kódu 10.1 připravíme testovanou funkcionalitu, tedy v tomto případě Jetpack Compose komponentu s nastavením aplikace (viz zdrojový kód 10.3):

```
@HiltAndroidTest
class SettingsScreenTest {

    @get:Rule(order = 0)
    val hiltRule = HiltAndroidRule(this)

    @get:Rule(order = 1)
    val composeRule = createAndroidComposeRule<MainActivity>()

    @Before
    fun setUp() {
        hiltRule.inject()
        composeRule.activity.setContent {
            val navController = rememberNavController()
            LostAndFoundTheme {
                NavHost(
                    navController = navController,
                    startDestination = Screen.SettingsScreen.route,
                ) {
                    composable(
                        route = Screen.SettingsScreen.route,
                    ) {
                        SettingsScreen()
                    }
                }
            }
        }
    }
    // instrumented integration (or even unit) tests...
}
```

Zdrojový kód 10.3: Příprava testované funkcionality pro instrumented testy

Jelikož chceme, aby testy proběhl co nejrychleji, definujeme si jednoduchou navigaci `NavHost` pro vytvoření navigačního grafu, kterému předáme jedinou cestu `SettingsScreen` s obrazovkou nastavení. Proměnné `hiltRule` a `composeRule` slouží jako reference k jednotlivým knihovnám, tedy k Dagger Hilt modulům a Jetpack Compose komponentám. Důležité je zde injektování závislostí voláním `inject`

nad referencí `hiltRule`.

Nyní máme pro testy připravenou obrazovku s nastavením a máme injektované všechny závislosti, tedy v tomto případě Proto DataStore s nastavením aplikace. Instrumented integrační testy by potom mohly například ověřit otevření dialogu při požadavku na změnu motivu (viz zdrojový kód 10.4):

```
@Test
fun whenClickingOnThemeInSettings_dialogOpens() {
    val themeSettingsItem = composeRule.activity.getString(
        R.string.screen_settings_item_theme
    )
    val themeDialog = composeRule.activity.getString(
        R.string.screen_settings_theme_dialog_title
    )
    val theme = composeRule.onNodeWithText(themeSettingsItem)
    theme.assertExists()
    theme.performClick()
    composeRule.onNodeWithText(themeDialog).assertExists()
}
```

Zdrojový kód 10.4: Integrační test s ověřením otevření dialogu v nastavení aplikace

Ze zdrojů aktivity si požádáme o aktuální název položky s motivem a název dialogu s motivem, podle kterých budeme jednotlivé Jetpack Compose komponenty hledat v uživatelském rozhraní. Po získání reference `theme` můžeme zavoláním `perfromClick` dialog otevřít a metodou `assertExists` zkontrolovat jeho přítomnost.

10.3 Testovací scénáře

Testování proběhlo jak na virtuálních zařízení, tak na fyzických zařízení. Specifikace virtuálních zařízení jsou v tab. 10.1 a specifikace fyzických zařízení v tab. 10.2.

Tabulka 10.1: Specifikace použitých virtuálních zařízení

Název zařízení	Verze Androidu	RAM	Počet jader	Rozměry
Pixel 4	12 (API 31)	1536MB	2	1080 x 2280
Pixel 4	13 (API 33)	1536MB	2	1080 x 2280
Nexus 4	12 (API 31)	1536MB	2	768 x 1280

Tabulka 10.2: Specifikace použitých fyzických zařízení

Název zařízení	Verze Androidu	RAM	CPU
Galaxy Note10+	12 (API 31)	12GB	Exynos 9825

Pro všechna testovací zařízení uvedená v tab. 10.1 a tab. 10.2 proběhly níže popsané testovací scénáře dle očekávání.

V testovacích scénářích se předpokládá, že uživatel je již přihlášený a jazyk aplikace má nastavený na angličtinu.

10.3.1 Vytváření příspěvku

Testovací scénář pro ověření funkčnosti vytváření příspěvků ztracených předmětů:

1. Klikneme na ikonu s popiskem *My posts*.
2. Zkontrolujeme, že jsme byli přesměrování na obrazovku *My posts*.
3. Klikneme na tlačítko "+".
4. Zkontrolujeme, že jsme byli přesměrování na obrazovku *Add item*.
5. Klikneme na nadpis *Choose an image*.
6. Vybereme náhodný obrázek.
7. Klikneme na textové pole *Title*.
8. Vložíme nějaký text.
9. Klikneme na textové pole *Description*.
10. Vložíme nějaký text.
11. Klikneme na tlačítko *Mark item on map*.
12. Označíme nějaké místo na mapě.
13. Klikneme na tlačítko *Confirm*.
14. Zkontrolujeme, že na místo textu *Mark item on map* je text *Item marked*.
15. Klikneme na tlačítko *Create*.

16. Zkontrolujeme, že jsme byli přesměrování zpět na obrazovku *My posts*, a že vytvořený předmět je součástí seznamu.

Očekávaný výsledek scénáře je vytvoření nového příspěvku s přesměrováním na obrazovku se seznamem příspěvků publikovaných od právě přihlášeného uživatele, kde nově vytvořený příspěvek je součástí seznamu.

10.3.2 Označení předmětu na mapě

Testovací scénář pro ověření funkčnosti map se ztracenými předměty:

1. Klikneme na ikonu s popiskem *My posts*.
2. Vybereme nějaký příspěvek, kterému jsme při vytváření určili polohu.
3. Klikneme na možnosti příspěvku označenými ikonou s třemi tečkami.
4. Zkontrolujeme, že se otevřelo menu s nabídkou *Edit* a *Delete*.
5. Klikneme na možnost *Edit*.
6. Zkontrolujeme, že jsme byli přesměrování na obrazovku *Update item*.
7. Zkontrolujeme, že příspěvek má tlačítko s textem *Item marked*.
8. Klikneme na ikonu *Maps* v dolní liště aplikace.
9. Zkontrolujeme, že jsme byli přesměrování na obrazovku *Maps*.
10. Zkontrolujeme, že ztracený předmět je správně označen na mapě.
11. Klikneme na lokaci ztraceného předmětu.
12. Zkontrolujeme, že se zobrazí popisek s nadpisem a vysvětlením příspěvku.

Očekávaný výsledek scénáře je správně označený předmět na obecných mapách s označenými předměty.

10.3.3 Připojení k internetu

Testovací scénář pro ověření funkčnosti aplikace bez připojení k internetu:

1. Klikneme na ikonu s popiskem *Find item*.
2. Zkontrolujeme, že vidíme příspěvky načtené při minulém použití aplikace, kdy jsme měli přístup k internetu.

3. Klikneme na ikonu s popiskem *My posts*.
4. Vybereme náhodný příspěvek a klikneme na ikonu s třemi tečkami.
5. Zkontrolujeme, že se otevřelo menu s nabídkou *Edit* a *Delete*.
6. Klikneme na možnost *Edit*.
7. Změníme popisek u předmětu.
8. Klikneme na tlačítko *Confirm* a zkontrolujeme, že obrazovka indikuje načítaní.
9. Zavřeme aplikaci.
10. Povolíme na zařízení síťové připojení a připojíme ho k internetu.
11. Zkontrolujeme, že je příspěvek automaticky aktualizován.

Očekávaný výsledek scénáře je možnost editace v offline režimu a následné automatické aktualizace upraveného příspěvku při připojení k internetu.

10.3.4 Změna nastavení

Testovací scénář pro ověření funkčnosti změny nastavení aplikace:

1. Klikneme na menu aplikace.
2. Klikneme na tlačítko *Settings*.
3. Klikneme na položku seznamu s názvem *Theme*.
4. Zkontrolujeme, že se otevřelo dialogové okno s nadpisem *Choose theme*.
5. Vybereme libovolný motiv.
6. Potvrdíme volbu kliknutím na tlačítko *Ok*.
7. Zkontrolujeme, že motiv aplikace se změnil dle vybrané volby.
8. Opakujeme s ostatními položkami v seznamu.

Očekávaný výsledek scénáře je změna motivu aplikace a dalších nastavení dle volby uživatele.

10.4 Firebase Crashlytics

Firebase Crashlytics [23] je služba, která pomáhá sledovat a opravovat problémy se stabilitou. Hlavní výhodou této služby je poskytování statistik s pády aplikace, společně s vysvětlením jak k daným chybám došlo.

Pro projekt byla aktivována služba Firebase Crashlytics, společně se službou Google Analytics pro sledování aktivit uživatelů. V projektu jsou tyto služby velmi důležité, jelikož je použito několik placených služeb a je tedy dobré mít přehled o tom, jak se aplikace chová a kdy přesáhne určitý stanovený finanční limit. Služba také nabízí nastavení finančního limitu a bez svolení zákazníka nezačne účtovat danou službu. V našem případě jsme si zažádali o zkušební verzi, která by v případě překročení limitu pouze danou službu zakázala.

10.5 Hodnocení kvality aplikace

Jako poslední si ukážeme hodnocení naší aplikace (viz tab. 10.3) podle seznamu kvalit aplikace [22] definovaného od Android vývojářů:

Tabulka 10.3: Tabulka kvalit finální implementované aplikace

Testovaná kvalita	**ID splněných testů**
Design a estetika	VX-N1, VX-N2, VX-N3, VX-V1, VX-V2, VX-V3, VX-A1, VX-A2
Funkcionalita	FN-B1
Výkon a stabilita	PS-S1, PS-P1, PS-P2, PS-T1, PS-T2, PS-T4 (do dne 22. dubna 2023), PS-T5, PS-T6
Ochrana soukromí a zabezpečení	SC-P1, SC-P2, SC-P3, SC-P5, SC-DF1, SC-DF2, SC-DF3, SC-ID1, SC-N1, SC-E1
Google Play	—

Z tab. 10.3 můžeme vidět, že aplikace splňuje například test VX-N1 u testování kvality designu a estetiky aplikace. Tento test ověřuje, zda *aplikace správně zachovává a obnovuje stav uživatele nebo aplikace*. Dalším zajímavým testem může být například PS-P1 u výkonu a stability, který testuje, zda se *aplikace načítá rychle nebo poskytuje uživateli zpětnou vazbu (indikátor průběhu nebo podobné upozornění), pokud načítání aplikace trvá déle než dvě sekundy*.

10.6 Shrnutí

Aplikace byla otestována unit a integračními testy, scénáři použití a ohodnocena seznamem kvalit kladené na Android aplikace. Dále byly pro projekt aktivovány monitorovací služby poskytované od Firebase. Aplikace navíc splňuje doporučované zásady z Android Jetpacku jako SOC, SSOT, UDF a DI a je proto připravená na přidání dalších testů pro pokrytí celého projektu testy.

Možná rozšíření aplikace

11

V této kapitole budou popsány možná rozšíření, která by šla do aplikace přidat. Jedná se jak o rozšíření s novou funkcionalitou pro uživatele, tak rozšíření pro budoucí vývoj a udržitelnost projektu.

11.1 Filtrování dat

Jelikož Firestore nepodporuje komplexní filtrovací požadavky, data jsou filtrována na klientských zařízeních. Nepodařilo se však najít službu, která by byla zdarma a měla integraci s Cloud Firestore databází pro filtrování dat. Proto další rozšíření spočívá ve vyřešení tohoto problému a zrychlení tak procesu filtrování příspěvků pro uživatele.

11.2 Migrace

Databáze příspěvků je již připravena na rozšíření funkcionality aplikace, například přidáním atributů, jako `isFound` pro ztracený předmět, které aktuální aplikace ještě nevyužívá, ale mohla by je potřebovat v pozdější verzi. Firebase však neposkytuje příliš intuitivní nástroje, pokud by v budoucích verzích byla potřeba změnit strukturu databáze. Většinou musíme vytvořit vlastní migrační systém, proto dalším důležitým rozšířením je příprava takového systému.

11.3 Rozpoznávání předmětů

Dalším zajímavým rozšířením by bylo využití knihoven jako jsou TensorFlow Lite nebo Firebase Machine Learning pro detekci vyfocených předmětů. Aplikace by tak mohla detekovaný předmět označit pro případné filtrování příspěvků. Příkladem může být vyfocený *telefon*, kdy uživatel s příspěvkem zvolil slovo *iPhone*, poté vyhledáním slova *telefon* nenalezneme žádné výsledky, ale s použitím zmiňovaných

knihoven by odpadala starost uživatele vyfocený předmět kompletně popsat pro všechny možné filtry.

11.4 Vytvoření příspěvku v offline režimu

V případě, že uživatel nemá přístup k internetu, má i tak možnost příspěvky prohlížet a upravovat. Upravené příspěvky se po připojení k internetu automaticky aktualizují na server. Dalším rozšířením by mohla být možnost vytvořit si příspěvek v offline režimu, který by měl stejnou funkcionalitu, tedy po připojení k internetu se příspěvek vytvořený v offline režimu automaticky aktualizuje na server.

11.5 Skupiny uživatelů

Dalším vhodným rozšířením by byla implementace skupin uživatelů. Tyto skupiny by viděly pouze sdílené příspěvky v dané skupině. Mohlo by se jednat například o nějakou školní instituci se správci, uživateli a potencionálně dalšími rolemi.

Závěr 12

Cílem práce bylo prozkoumat vývoj Android aplikací s využitím Android Jetpack knihoven a následně vytvořit aplikaci, která ukáže použití Android Jetpacku. Práce rozebrala možnosti a výhody vývoje s použitím Android Jetpack knihoven a dále se zabývala vytvářením uživatelského rozhraní s využitím nástroje Jetpack Compose a architekturou projektu doporučovanou vývojáři Android Jetpacku.

Výsledek práce je aplikace pro mobilní telefony na platformě Android, která by měla uživatelům poskytnout jednotný nástroj pro hledaní a publikaci ztracených předmětů.

Na základě doporučení z Android Jetpacku se podařilo implementovat robustní architekturu projektu, která splňuje principy jako SOC, SSOT nebo UDF. Celkově by vytvořená aplikace měla používat aktuální technologie a měla by demonstrovat výhody, které Android Jetpack přináší. Dále by měla ukázat moderní deklarativní způsob vývoje uživatelského rozhraní, který v dnešní době pomalu nahrazuje dosavadní imperativní styl programovaní.

Následně byla aplikace ověřena unit a integračními testy, scénáři použití, seznamem kvalit, který je vhodné při vývoji Android aplikací dodržovat a jako poslední byly zprovozněny Firebase služby pro sledování stavu aplikace.

Poslední část práce se věnovala vhodným rozšířením aplikace a k čemu by mohla jednotlivá rozšíření sloužit.

Práce má za cíl čtenáře seznámit s možnostmi moderního vývoje pro Android s využitím jednotné sady knihoven z Android Jetpacku. Použití Android Jetpacku pomáhá řešit potencionální úniky paměti, správu životního cyklu, změny konfigurace nebo omezení nutnosti psát boilerplate kód. Android Jetpack tak představuje významnou pomoc při vývoji nativních aplikací pro Android.

Přehled zkratek

API	Application Programming Interface
APK	Android Package Kit
CDN	Content Delivery Network
DEX	Dalvik Executable
DI	Dependency Injection
DTO	Data Transfer Object
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MVVM	Model-View-Viemodel
NoSQL	Not only Structured Query Language
REST	Representational State Transfer
SOC	Separation of Concerns
SSOT	Single Source of Truth
UDF	Unidirectional Data Flow
XML	Extensible Markup Language

Bibliografie

1. MEIER, Reto; LAKE, Ian. *Professional Android*. Wrox, 2018. ISBN 978-1-11894952-8.
2. DIMARZIO, Jerome. *Beginning Android Programming with Android Studio (Wrox Beginning Guides)*. Wrox, 2016. ISBN 978-1-11870559-9.
3. THOMAS, Gaël. What is Flutter and Why You Should Learn it in 2020. *FreeCodeCamp*. 2021. Dostupné také z: `https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020`.
4. *Handling Lifecycles with Lifecycle-Aware Components*. 2023-03. Dostupné také z: `https://developer.android.com/topic/libraries/architecture/lifecycle`. [Online; accessed 20. Mar. 2023].
5. *AndroidX Overview*. 2022-01. Dostupné také z: `https://developer.android.com/jetpack/androidx`. [Online; accessed 9. Feb. 2023].
6. PRESTON-WERNER, Tom. *Semantic Versioning 2.0.0*. 2023-03. Dostupné také z: `https://semver.org/spec/v2.0.0.html`. [Online; accessed 6. Apr. 2023].
7. *Android Code Search - FAQ*. 2023-02. Dostupné také z: `https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:docs/faq.md`. [Online; accessed 24. Feb. 2023].
8. LARDINOIS, Frederic. Android gets a Jetpack. *TechCrunch*. 2018. Dostupné také z: `https://techcrunch.com/2018/05/08/android-gets-a-jetpack/?guccounter=1`.
9. *Enable multidex for apps with over 64K methods*. 2023-04. Dostupné také z: `https://developer.android.com/build/multidex`. [Online; accessed 6. Apr. 2023].
10. *Jetpack Compose UI App Development Toolkit - Android Developers*. 2023-02. Dostupné také z: `https://developer.android.com/jetpack/compose`. [Online; accessed 25. Feb. 2023].

11. GHITA, Catalin. *Kickstart Modern Android Development with Jetpack and Kotlin: Enhance your applications by integrating Jetpack and applying modern app architectural concepts*. Packt Publishing, 2022. ISBN 978-1-80181107-1.

12. *Why Compose*. 2023-03. Dostupné také z: `https://developer.android.com/jetpack/compose/why-adopt`. [Online; accessed 1. Mar. 2023].

13. *Architecting your Compose UI*. 2023-03. Dostupné také z: `https://developer.android.com/jetpack/compose/architecture`. [Online; accessed 1. Mar. 2023].

14. MARTIN, Robert. *Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)*. London, England, UK: Pearson, 2017. ISBN 978-0-13449416-6.

15. BABAR, Muhammad Ali; GORTON, Ian. Software Architecture Review: The State of Practice. *Computer*. 2009, roč. 42, č. 7, s. 26–32. Dostupné z DOI: `10.1109/MC.2009.233`.

16. *Benefits of software architecture you should know*. 2022-03. Dostupné také z: `https://apiumhub.com/tech-blog-barcelona/benefits-of-software-architecture`. [Online; accessed 16. Mar. 2023].

17. DIJKSTRA, Edsger W. *Selected Writings on Computing: A personal Perspective (Monographs in Computer Science)*. New York, NY, USA: Springer, 2011. ISBN 978-1-46125697-7.

18. *Guide to app architecture*. 2023-02. Dostupné také z: `https://developer.android.com/topic/architecture`. [Online; accessed 16. Mar. 2023].

19. *Clean Coder Blog*. 2023-01. Dostupné také z: `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`. [Online; accessed 16. Mar. 2023].

20. ADILOVIC, Abdurahman. A Complete Guide of Recommended Android Apps Architectures. *Scalable Path*. 2022. Dostupné také z: `https://www.scalablepath.com/android/android-apps-architecture`.

21. *Distribution dashboard*. 2023-01. Dostupné také z: `https://developer.android.com/about/dashboards`. [Online; accessed 21. Apr. 2023].

22. *Core app quality*. 2023-03. Dostupné také z: `https://developer.android.com/docs/quality-guidelines/core-app-quality`. [Online; accessed 21. Apr. 2023].

23. *Firebase Crashlytics*. 2023-05. Dostupné také z: `https://firebase.google.com/docs/crashlytics`. [Online; accessed 2. May 2023].

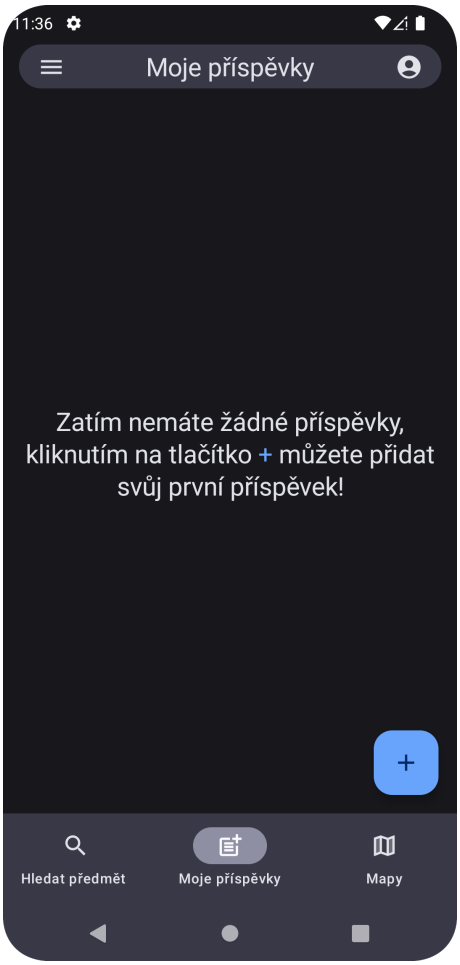
Přílohy

Uživatelská příručka

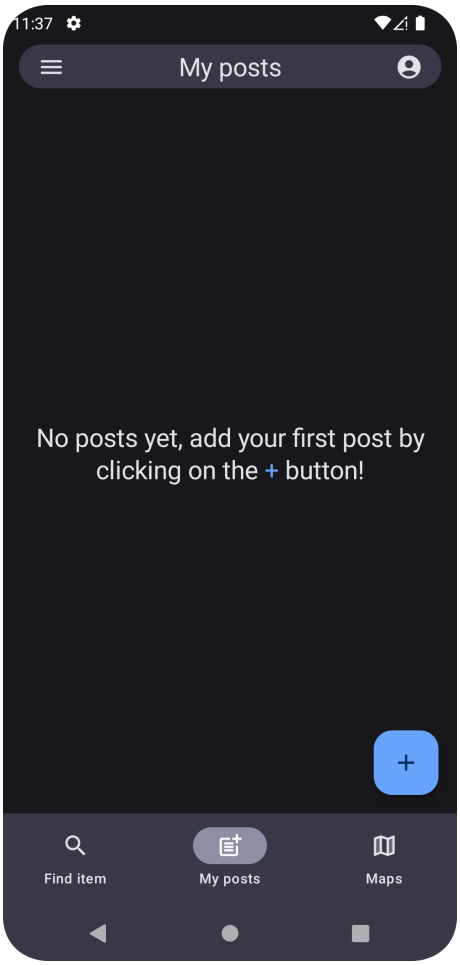
Aplikace poskytuje jednotný systém pro hledání a publikaci ztracených předmětů.

Přidání příspěvku

Tato funkcionalita je důležitá pro uživatele, kteří našli nějaký předmět a ten chtějí publikovat. První obrazovka, kterou uživatel uvidí je vidět na obr. 1. Po kliknutí na tlačítko pro přidání předmětu bude uživatel přesměrován na formulář pro publikaci příspěvků (viz obr. 2), kde nalezený předmět může označit na mapě (viz obr. 3) a musí vyplnit zbylé údaje jako nadpis a popis předmětu (viz obr. 4). Po přidání příspěvku bude přesměrován na seznam příspěvků, které vlastní (viz obr. 5).

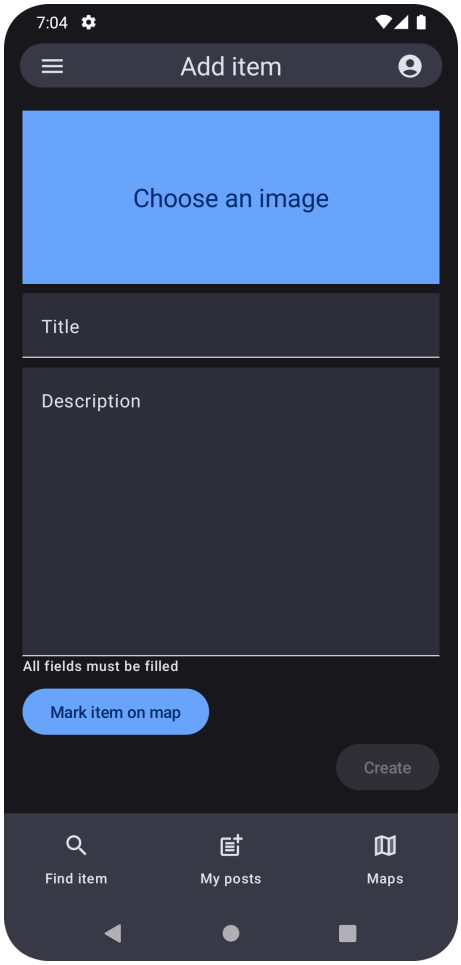


(a) Snímek obrazovky s uživatelem bez zveřejněných příspěvků, česky

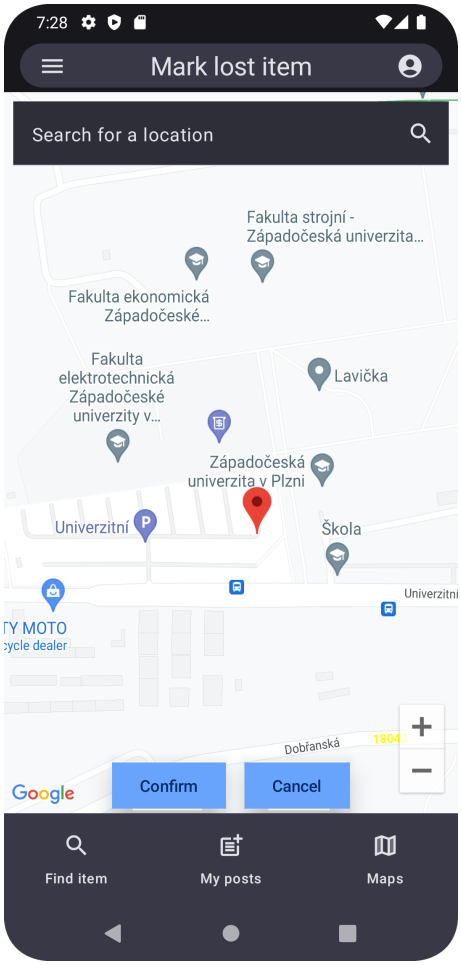


(b) Snímek obrazovky s uživatelem bez zveřejněných příspěvků, anglicky

Obrázek 1: Ukázkou multijazyčnosti a obrazovky uživatele bez příspěvků

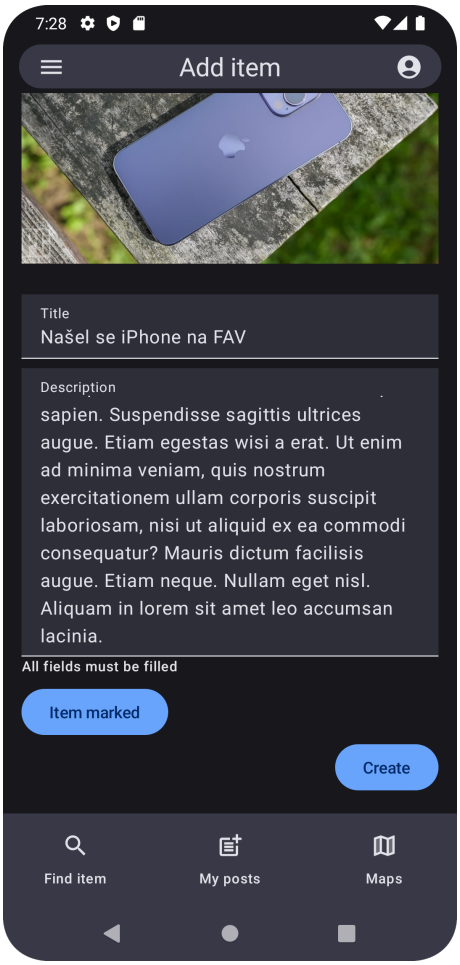


Obrázek 2: Snímek obrazovky s prázdným formulářem pro přidání příspěvku



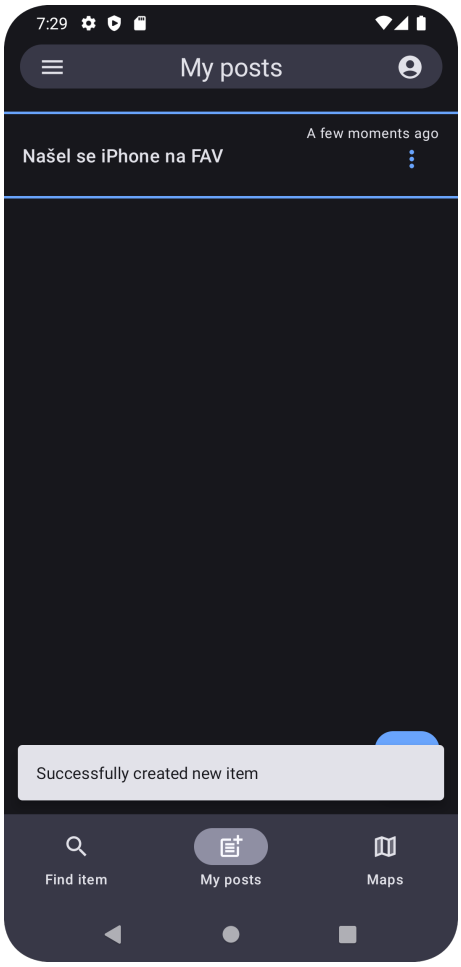
Obrázek 3: Snímek obrazovky s označenou lokací nalezeného předmětu na mapě

(a) Snímek obrazovky s vyplněnými povinnými údaji



(b) Snímek obrazovky s vyplněnými údaji a označeným předmětem na mapě

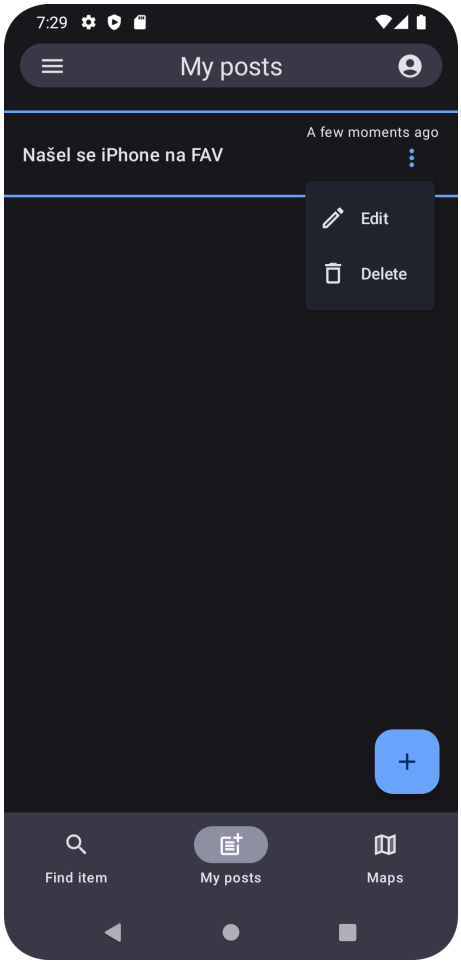
Obrázek 4: Vyplněný formulář pro přidání příspěvku se ztraceným předmětem



Obrázek 5: Snímek obrazovky po úspěšném zveřejnění příspěvku a přesměrovaní na obrazovku s přehledem

Úprava příspěvku

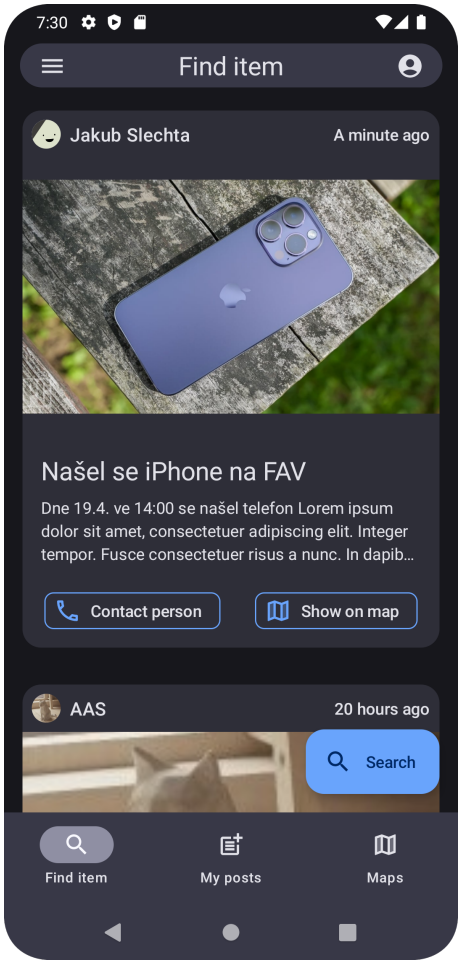
Aplikace také umožňuje zveřejněný příspěvek upravit nebo smazat (viz obr. 6).



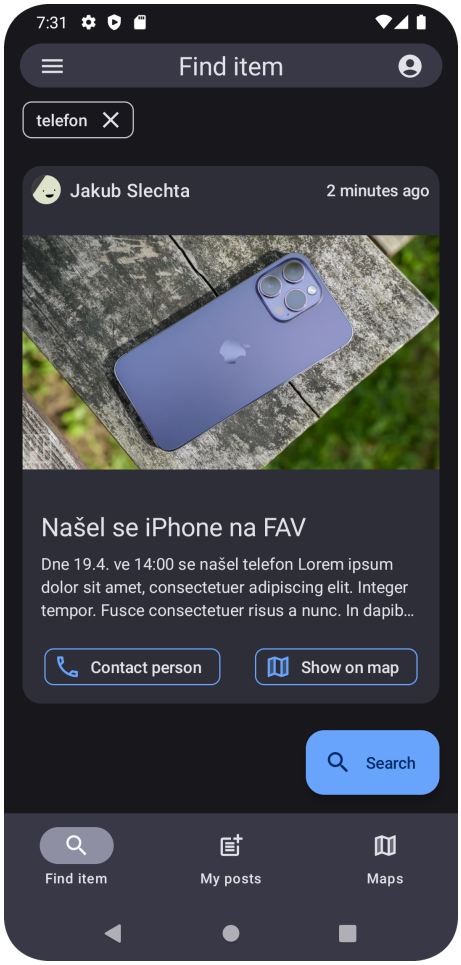
Obrázek 6: Snímek obrazovky s možností úpravy zveřejněného příspěvku

Seznam ztracených předmětů

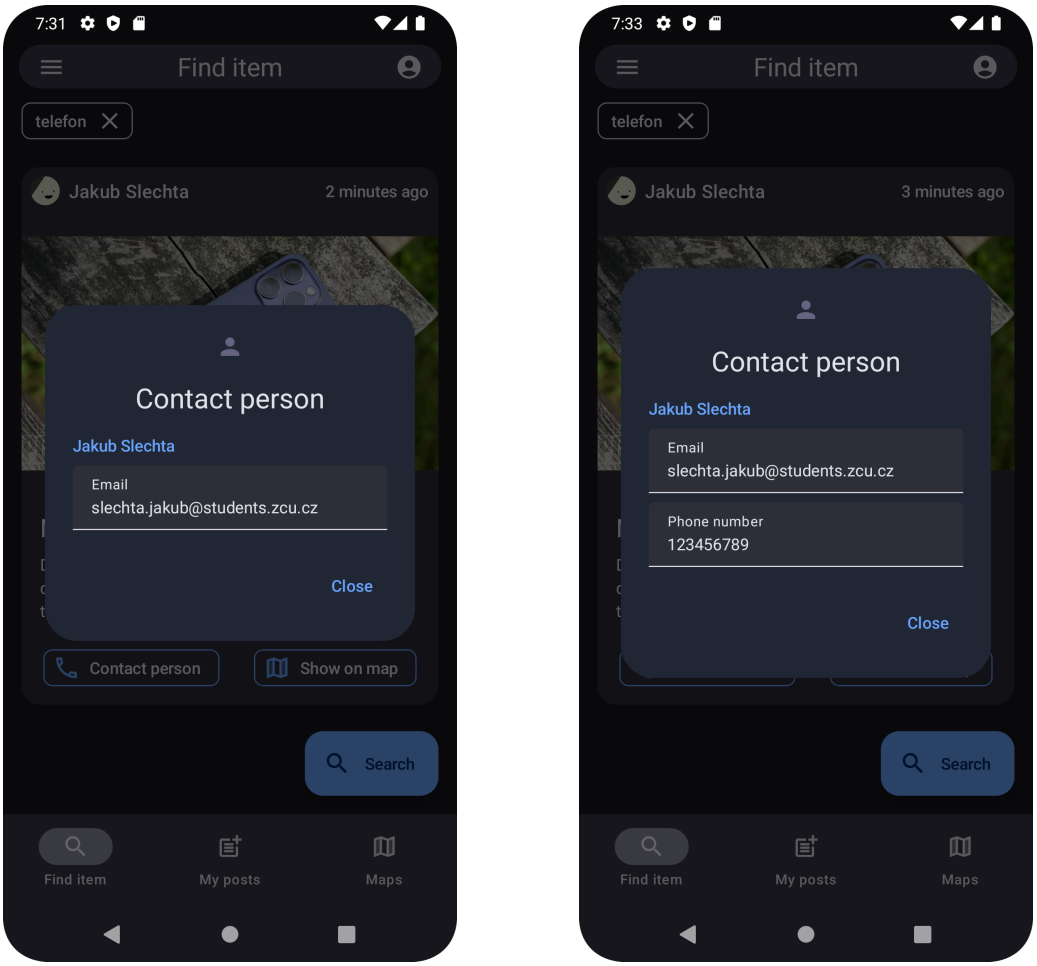
Seznam ztracených předmětů (viz obr. 7) obsahuje kompletní seznam příspěvků. Seznam je také možné filtrovat (viz obr. 8). Pokud by uživatele zajímal některý příspěvek, může si rozkliknutím položky seznamu zobrazit detail ztraceného předmětu (viz obr. 10) nebo v případě označeného předmětu si zobrazit na mapách místo nálezu s popiskem (viz obr. 11). Důležitá je zde také možnost získat kontakt na majitele příspěvku, který má buď dostupnou emailovou adresu nebo telefonní číslo (viz obr. 9).



Obrázek 7: Snímek domovské obrazovky se seznamem ztracených předmětů



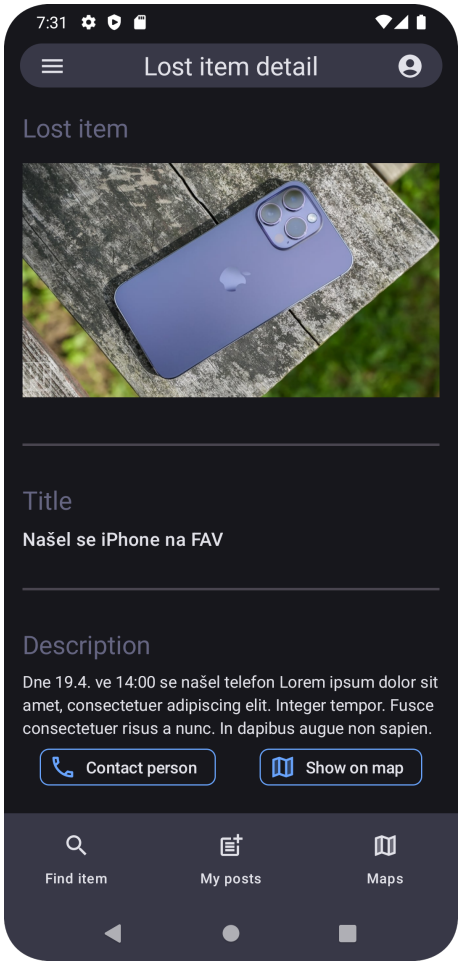
Obrázek 8: Snímek obrazovky seznamu ztracených předmětů s aplikovaným filtrem



(a) Snímek obrazovky s kontaktem na uživatele bez zveřejněného telefonního čísla

(b) Snímek obrazovky s kontaktem na uživatele se zveřejněným telefonním číslem

Obrázek 9: Různé možnosti jak kontaktovat majitele příspěvku s nalezeným předmětem

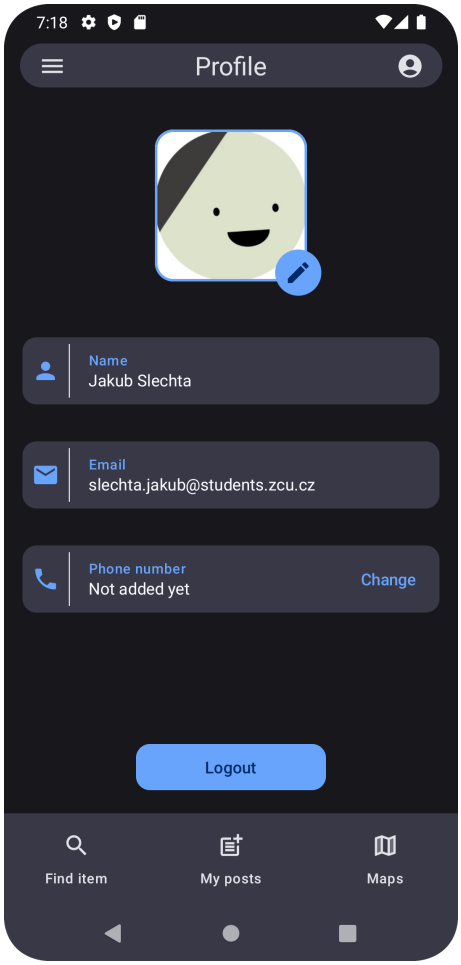


Obrázek 10: Snímek obrazovky s detailem ztraceného předmětu

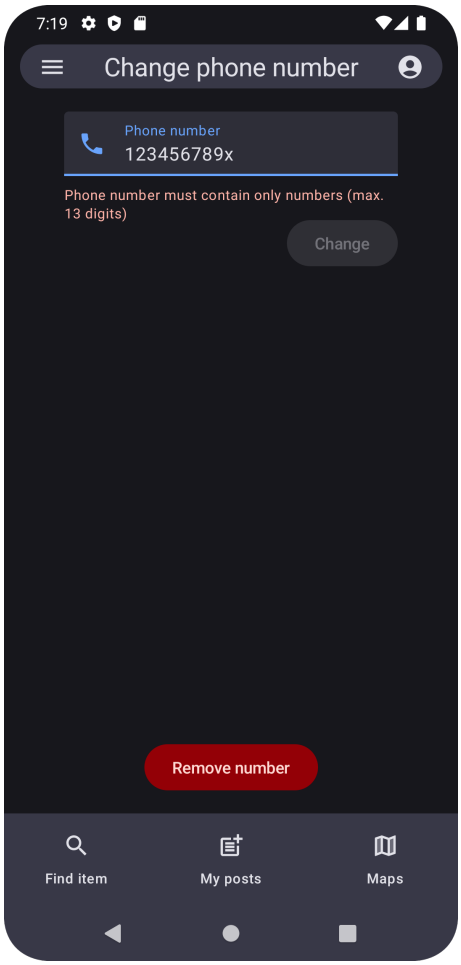
Obrázek 11: Snímek obrazovky s mapou zveřejněných příspěvků, které mají přiřazenou lokaci nalezeného předmětu

Profil uživatele

Profil uživatele obsahuje základní informace jako jméno, email a telefonní číslo uživatele (viz obr. 12). Dále si uživatel může zvolit, zda chce zveřejnit své telefonní číslo (viz obr. 13).



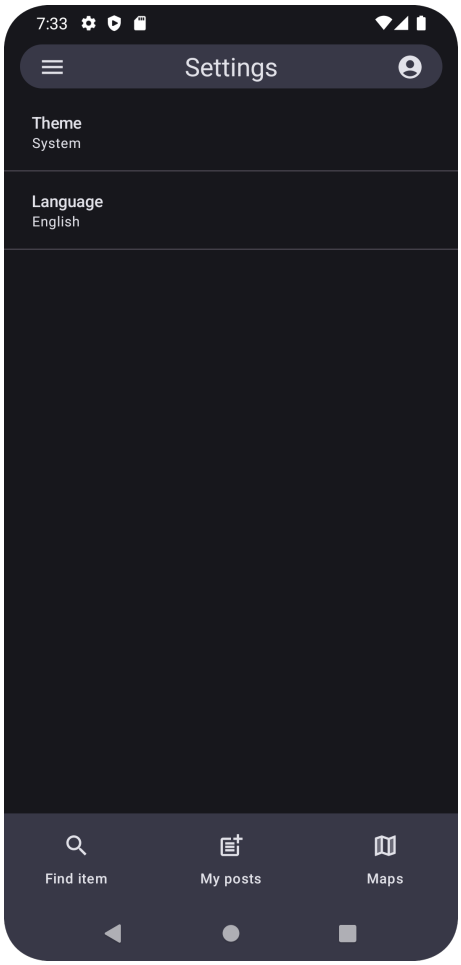
Obrázek 12: Snímek obrazovky s profilem přihlášeného uživatele



Obrázek 13: Snímek obrazovky formuláře pro změnu telefonního čísla s upozorněním o požadovaném formátu

Nastavení

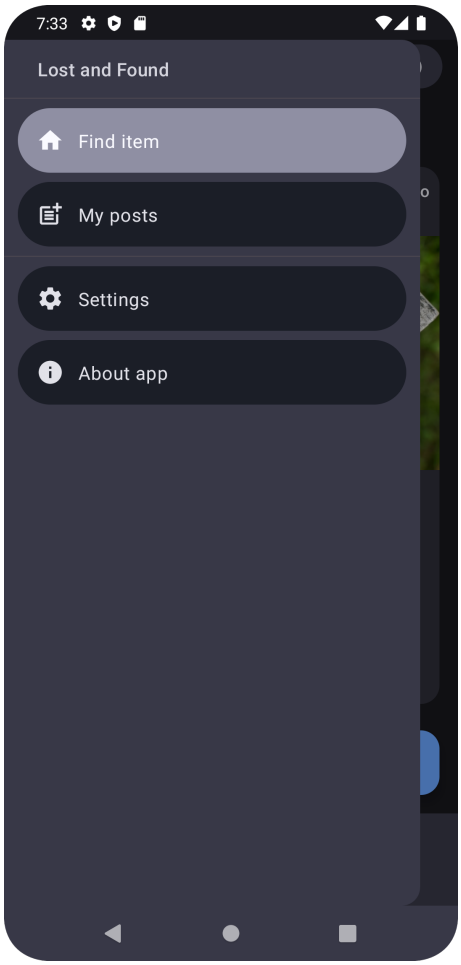
Nastavení aplikace obsahuje možnost výběru motivu a jazyku aplikace (viz obr. 14).



Obrázek 14: Snímek obrazovky s nastavením aplikace

Navigace

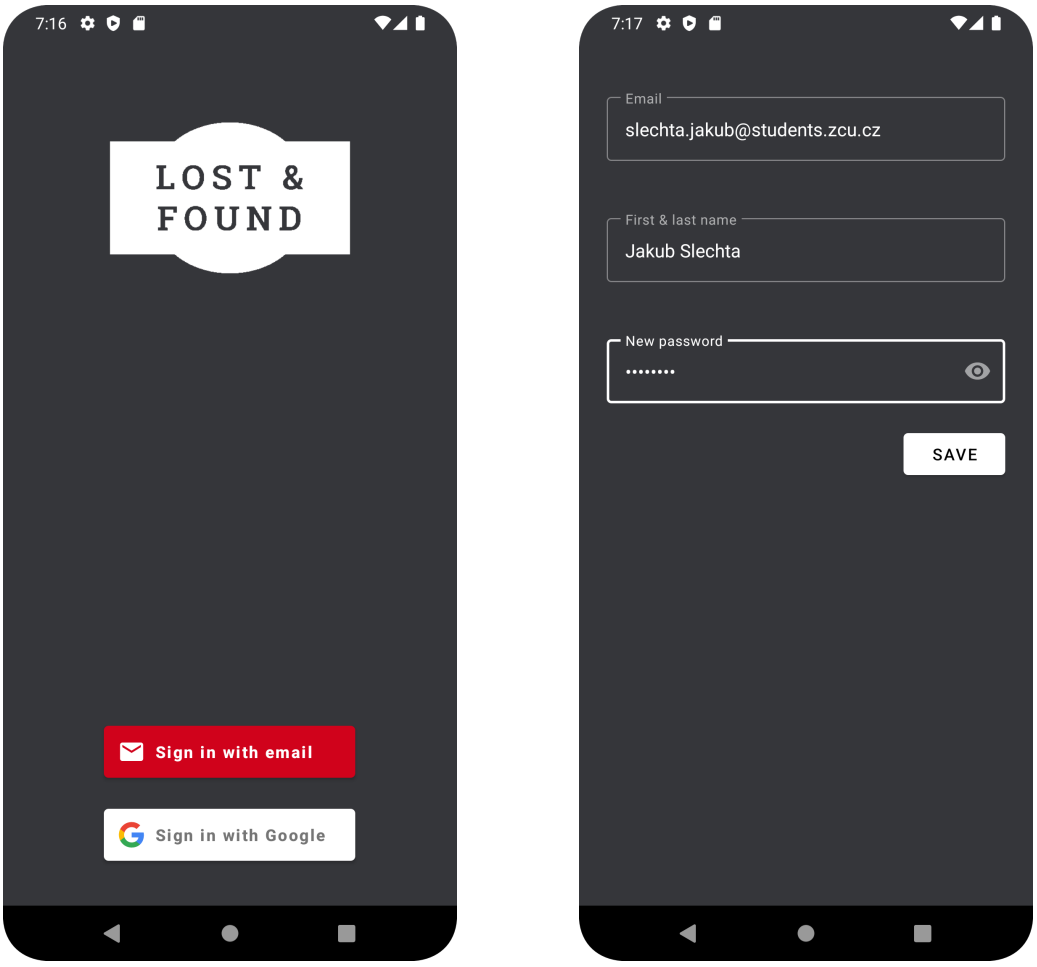
Pro snazší orientaci je dostupná obecná navigace (viz obr. 15).



Obrázek 15: Snímek obrazovky s navigačním panelem

Přihlášení a registrace

Aby šlo aplikaci používat, uživatel se musí nejprve registrovat (viz obr. 16). Po přihlášení zůstane uživatel přihlášený i při dalším spuštění aplikace.



(a) Snímek obrazovky pro nové a nepřihlášené uživatele

(b) Snímek obrazovky s formulářem pro registraci

Obrázek 16: Řešení možnosti registrace a přihlášení více způsoby

Instalační příručka

Pro nainstalování aplikace je nejprve potřeba nakopírovat dodaný APK soubor s názvem `lostandfound.apk` do cílového mobilního zařízení s minimální API úrovní 31. Poté otevřete na daném zařízení prohlížeč souborů a nakopírovaný soubor rozklikněte. Pokud na daném zařízení nebyla doposud povolena instalace aplikací, jež pocházejí z neznámého zdroje, zobrazí se dialogové okno, které vás přesměruje do nastavení, kde musíte tuto možnost povolit.

Obsah přiloženého ZIP souboru

ZIP soubor obsahuje:

- PDF verzi bakalářské práce,
- zdrojové soubory bakalářské práce,
- soubor `Readme.md`,
- zdrojové kódy aplikace,
- dokumentaci k aplikaci,
- APK soubor s aplikací.

Seznam obrázků

Seznam tabulek

Seznam výpisů