Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Perzistentní ukládání dat na platformě Android

Bakalářská práce

2022

Oleg Golovkov

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Oleg Golovkov**
Osobní číslo:	**I20256**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Perzistentní ukládání dat na platformě Android**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem bakalářské práce je tvorba mobilní aplikace umožňující perzistentní ukládání dat na platformě Android.
V teoretické části budou popsány použité technologie spojené s řešeným tématem bakalářské práce, důvod jejich výběru a popis zvolené architektury. Výstupem praktické části bude mobilní aplikace, která bude využívat abstraktní vrstvu v podobě knihovny Room s jejímž využití bude docházet k ukádání dat v mobilní aplikaci. Součástí práce bude detailní popis tvorby praktické části včetně analýzy zadání, návrhu systému, popis implementace a způsob ověření výsledků. V příloze práce bude uveden postup pro nasazení praktického výstupu včetně uvedení nutných systémových a provozních prostředků pro správný běh aplikace.

Rozsah pracovní zprávy: **40**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

BANDEKAR, Namrata, Darryl BAYLISS, Fuad KAMAL, Kevin MOORE a Tom BLANKENSHIP, [2021]. Android apprentice: beginning android developement with Kotlin. Fourth edition. McGaheysville: Razeware. ISBN 978-1950325399.
PHILLIPS, Bill, Chris STEWART, Kristin MARSICANO a Brian GARDNER, [2019]. Android Programming: The big nerd ranch guide. 4th edition. Atlanta: Big nerd ranch. ISBN 978-0135245125.
NUDELMAN, Greg, [2013]. Android design patterns: interaction design solutions for developers. Indianapolis: Wiley. ISBN 978-1118394151.
LEIVA, Antonio, [2016]. Kotlin for Android developers: learn Kotlin the easy way while developing an Android app. 7th edition. Scotts Valley: CreateSpace. ISBN 978-1530075614.
BAILEY, Jennifer, Dean DJERMANOVIĆ, Aldo Olivares DOMINGUEZ, Fuad KAMAL, Subhrajyoti SEN a Harun WANGEREKA, [2021]. Saving Data on Android: learn Jetpack DataStore, Room, Firebase & SQLite with Kotlin. Second edition. McGaheysville: Razeware. ISBN 978-1950325436.
FAZIO, Michael, [2021]. Kotlin and Android development featuring Jetpack: build better, safer Android apps. Raleigh: Pragmatic Bookshelf. ISBN 978-1680508154.

Vedoucí bakalářské práce: **Ing. Monika Borkovcová, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **17. prosince 2021**
Termín odevzdání bakalářské práce: **13. května 2022**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2022

Prohlašuji:

Práci s názvem Perzistentní ukládání dat na platformě Android jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 10. 5. 2022

Oleg Golovkov

PODĚKOVÁNÍ

Rád bych poděkoval své rodině za finanční a morální podporu po celou dobu mého studia. Další obrovské poděkování patří kolegovi z práce Ing. Pavlovi Schreinerovi, za cenné rady při řešení praktické části. V neposlední řadě patří velký dík vedoucí bakalářské práce Ing. Monice Borkovcové Ph.D. bez Vás by nebylo možné tuto práci začít psát a úspěšně ji dokončit v stanovenom termínu.

ANOTACE

Tato práce se zabývá problematikou vytvoření mobilní aplikace pro platformu Android a trvalého ukládání informací do lokální databáze. Práce popisuje možné způsoby ukládání dat, hlavní typy, vzory vývojových architektur a dostupné frameworky pro vývoj s lokální databází. Součástí práce je i analýza a implementace aplikace pro práci s daty poskytovanými SpaceX a následné uložení do lokální databáze. Pro ověření výsledků je uveden postup testování.

KLÍČOVÁ SLOVA

Android, Room, Kotlin, databáze, architektura mobilní aplikace

TITLE

Persistent data storage in Android

ANNOTATION

This thesis deals with the problem of creating a mobile application for the Android platform and permanently storing information in a local database. The thesis describes possible ways of data storage, main types and patterns of development architectures and available frameworks for development with local database. The thesis also includes the analysis and implementation of an application for working with data provided by SpaceX and then storing it in a local database. A testing procedure is presented to validate the results.

KEYWORDS

Android, Room, Kotlin, database, mobile application architecture

OBSAH

SEZNAM OBRÁZKŮ ... 9
SEZNAM TABULEK ... 10
SEZNAM ZDROJOVÝ KÓDŮ ... 11
SEZNAM ZKRATEK ... 12
ÚVOD ... 13
1 UKLÁDÁNÍ DAT A SOUBORŮ ... 15
1.1 SharedPreferences – sdílena data ... 15
1.2 Vnitřní úložiště ... 16
1.3 Externí úložiště ... 16
1.4 Ukládání dat do cache paměti ... 17
1.5 SQLite databáze ... 17
1.6 Základní operace v databázi (CRUD) ... 18
2 ARCHITEKTURY APLIKACÍ ... 19
2.1 MVC (Model – View – Controller) ... 19
2.2 MVP (Model – View – Presenter) ... 20
2.3 MVVM (Model – View – ViewModel) ... 23
2.4 VIPER (View – Interactor – Presenter - Router) ... 25
2.5 MVI (Model – View – Intent) ... 26
3 OBJEKTOVĚ RELAČNÍ MAPOVÁNÍ ... 28
3.1 OrmLite ... 29
3.2 SugarORM ... 29
3.3 GreenDAO ... 29
3.4 Active Android ... 29
3.5 Room perzistentní knihovna ... 29
3.5.1 Room architektura ... 30
3.5.2 Definice entity ... 30
3.5.3 Druhy vztahů ... 31
3.5.4 Šablona DAO ... 31

4 NÁVRH 32
4.1 Analýza 32
4.1.1 Funkční požadavky 33
4.1.2 Nefunkční požadavky 33
4.2 Návrh databáze 34
5 IMPLEMENTACE 35
5.1 Grafické uživatelské rozhraní 35
5.1.1 Navigace v aplikaci 36
5.2 Připojení k REST API a zpracování výsledků 36
5.3 Perzistentní ukládání s využitím knihovny Room 37
5.3.1 Databázový model 37
5.3.2 Data access object 38
5.3.3 Databázové vztahy 39
5.3.4 Inicializace databáze 40
5.4 Bezpečnost a ochrana dat 41
6 TESTOVÁNÍ 43
6.1 End – to – end testy 43
6.2 Integrační testy 45
6.3 Unit testy 46
ZÁVĚR 48
POUŽITÁ LITERATURA 49
PŘÍLOHY 51

SEZNAM OBRÁZKŮ

Obrázek 1:.Realistická aplikace modelu MVC pro Android 20
Obrázek 2:.Presenter a Model nesmí obsahovat třídy specifické pro platformu Android 21
Obrázek 3:.Model MVP 22
Obrázek 4:.Vytváření rozhraní pro Presenter a View 23
Obrázek 5:.Model MVVM 24
Obrázek 6:.Model VIPER 25
Obrázek 7:.Model MVI 27
Obrázek 8:.Filmová třída s vlastnostmi 28
Obrázek 9:.Filmova tabulka 28
Obrázek 10:.Schéma architektury knihovny Room 30
Obrázek 11:.Relační databázový model 34
Obrázek 12:.Grafický návrh aplikace SpaceX 35
Obrázek 13:.Grafické znázornění navigace v aplikace SpaceX 36
Obrázek 14:.Reprezentace dat přijatých ze SpaceX API 43
Obrázek 15:.Aktivace režimu Letadlo 44
Obrázek 16:.Reprezentace dat, když je zapnutý režim Letadlo 44
Obrázek 17:.Výsledky spuštění třídy LaunchWithShipDaoTest.kt 46

SEZNAM TABULEK

Tabulka 1:.Funkční požadavky 33
Tabulka 2:.Nefunkční požadavky 33

SEZNAM ZDROJOVÝ KÓDŮ

Zdrojový kód 1:.RetrofitClient.kt 37
Zdrojový kód 2:.LaunchLocal.kt 38
Zdrojový kód 3:.LaunchWithShipDao.kt pro přístup k databázi 39
Zdrojový kód 4:.LaunchesWithShips.kt 39
Zdrojový kód 5:.LaunchShipCrossRef.kt propojovací tabulka 39
Zdrojový kód 6:.LaunchWithShipDao.kt pro přístup k tabulce se vztahy many to many 40
Zdrojový kód 7:.AppDatabase.kt 41

SEZNAM ZKRATEK

API	Application Programming Interface
CRUD	Create Read Update Delete
DAO	Data Access Object
DBMS	Database Management Systems
GB	Gigabyte
JSON	JavaScript Object Notation
MVC	Model – View – Controller
MVI	Model – View – Intent
MVP	Model – View – Presenter
MVVM	View – Interactor – Presenter - Router
MySQL	My Structured Query Language
ORM	Object Relational Mapping
REST	Representational State Transfer
SD	Secure Digital
SDK	Software Development Kit
SQL	Structured Query Language
USB	Universal Serial Bus
VIPER	View – Interactor – Presenter - Router
XML	Extensible Markup Language

ÚVOD

Mobilní telefon má dnes téměř každý, přičemž dominantním řešením je operační systém Android. Ukládání jakýchkoli dat je nezbytné v každé aplikaci. Různé typy dat vyžadují různé úložiště. Některá data vyžadují krátkodobé uložení, zatímco jiná potřebují bezpečné dlouhodobé uložení. Motivací pro tuto práci byly následující informace.

Rozvoj mobilních technologií se zlepšuje, a proto klesají ceny komponent, jako jsou paměti a procesory. Jedním z důležitých požadavků při nákupu nového mobilního telefonu je velikost paměti pro trvalé uložení dat. V roce 2021 má průměrný Android smartphone 95,7 GB paměti, telefony se systémem iOS se v průměru prodávají se 140,9 GB úložiště. Z toho vyplývá, že průměr mezi dvěma nejoblíbenějšími platformami je více než 100 GB. [15]

Celosvětové rozložení doby sledování YouTube podle typu zařízení ukazuje, že více než sedmdesát procent uživatelů platformy používá ke konzumaci obsahu mobilní zařízení. [16] Oblíbené služby pro sledování filmů a seriálů Netflix, HBO. K poslechu hudby využití platforem Spotify a Apple Music. Disponují funkcí ukládání vysoce kvalitního obsahu do trvalé paměti zařízení pro offline přístup, což vyžaduje velké množství paměti. Ukládání obsahu do trvalé paměti navíc nezanedbávají na první pohled zdánlivě nepotřebné aplikace jako Instagram a TikTok. Příklad pásku krátkých videí je dočasně uložen v telefonu na 10–15 videí dopředu. Důvodem je, že v případě špatného pokrytí sítě uživatel nemusí pokaždé čekat na obnovení připojení.

Cílem práce je tvorba mobilní aplikace umožňující perzistentní ukládání dat na platformě Android. Tato práce se tak soustředí na zvážení všech možných způsobů interakce s pamětí mobilního zařízení na platformě Android. Hlavní pozornost je věnována knihovně *Room*, která byla oficiálně představena na konferenci Google I/O 2017 jako doporučená pro použití ve vývojových ORM řešeních Andorid.

V prvních třech kapitolách teoretické části budou zvažovány dostupné způsoby práce s pamětí a jejich zásadní rozdíly. Druhá kapitola bude vysvětlovat, co jsou mobilní vývojové architektury, proč jsou potřebné a jejich hlavní typy a rozdíly. Třetí kapitola bude popisovat objektově-relační mapování, hlavní historicky důležité frameworky, knihovny a podrobný popis a potřebné komponenty pro práci s moderní knihovnou *Room*.

Praktická část práce bude rozdělena do tří částí, skládá se z návrhu, implementace a testování. V první části budou definovány funkční a nefuknční požadavky, druhá část se pak bude

soustředit na konkrétní implementaci s příklady kódu a popisem jeho významu. Poslední kapitola pak bude verifikační, jelikož prezentuje způsoby testování, které slouží pro ověření zadaných cílů a požadavků.

1 UKLÁDÁNÍ DAT A SOUBORŮ

Tato kapitola shrnuje hlavní způsoby ukládání dat na platformě Android. Každá metoda má své specifické vlastnosti, které se odvíjejí od doby uložení a typu dat.

1.1 SharedPreferences – sdílena data

Soubory jsou jednou z nejprimitivnějších a nejpohodlnějších metod ukládání nestrukturovaných dat na platformě Android. Existují však pohodlnější a logicky uspořádané metody pro ukládání malého množství informací nazývané SharedPreferences. [1]

SharedPreferences nebo kratší název prefs je rozhraní API, které je součástí sady Android SDK a poskytuje možnost zapisovat a číst soubory XML. Rozhraní API je užitečné zejména tehdy, když potřebujete uložit běžná programová data pro stylování aplikace, postup uživatele, filtry dat atd. [1]

Soubory jako prefs, které vytvoříte, upravíte nebo odstraníte, jsou uloženy v adresáři *data/data/{application packe}*. Umístění tohoto souboru lze získat voláním funkce programu z knihovny *Environment.getDataDirectory()*. Stejně jako soubory jsou data specifická pro aplikaci a budou ztracena, pokud jsou data aplikace ručně vymazána z nastavení zařízení nebo pokud je aplikace zcela odstraněna ze zařízení. [1]

Pro ukládání velkého množství dat není použití SharedPreferences rentabilní, protože byly vytvořeny a nacházejí lepší využití pro ukládání malého množství dat, jako jsou uživatelsky zvolené styly, skóre nebo úroveň v případě hry. Data jsou tedy strukturována do párů klíč-hodnota, kde každá hodnota má samostatný klíč-identifikátor, který označuje její umístění v souboru. Předáním klíče je možné tak získat poslední uloženou hodnotu. V případě, že pro klíč neexistuje žádná hodnota, lze definovat výchozí hodnotu, která bude vrácena. Protože účelem SharedPreferences je ukládat jednoduchá data, jedinými podporovanými typy pro ukládání dat jsou Int, Float, String a Set<String>. [1]

V závislosti na tom, jak je zapotřebí používat SharedPreferences, existují tři hlavní způsoby přístupu nebo vytvoření souboru SharedPreferences:

1. Pro užití více souborů SharedPreferences s různými názvy, je k dispozici metoda *getSharedPreferences()*, kterou lze volat z libovolného kontextu v aplikaci a umožňuje nastavit název souboru.

2. Pokud je zapotřebí vytvořit různé soubory pro přizpůsobení každé obrazovky (aktivity) odděleně od ostatních, je možné využít metodu *getPreferences()*. Ta se volá z obrazovky, na kterou bude soubor vytvořen, bez možnosti změny názvu souboru, protože ve výchozím nastavení používá název třídy této obrazovky.
3. Pro přístup najednou k celé aplikaci, což je užitečné při ukládání dat, která jsou nezbytná pro celou aplikaci, nebo funkce, které je třeba nakonfigurovat pokaždé, když uživatel spustí aplikaci slouží metoda *getDefaultSharedPreferences()*, která se používá ke konfiguraci a použití předvoleb pro celou aplikaci a vrací výchozí soubor nastavení pro celou aplikaci. [1]

1.2 Vnitřní úložiště

Interní úložiště je ve výchozím nastavení soukromé a může ho používat pouze konkrétní aplikace. Systém proto tedy vytvoří adresář vnitřního úložiště, a to pro každou aplikaci zvlášť a pojmenuje jej názvem balíčku aplikace. Interní úložiště má omezený prostor pro data aplikací.

Stejně jako v SharedPreferences dojde ke ztrátě dat uložených v interním úložišti při úplném odstranění aplikace ze zařízení. Interní úložiště je skvělé pro ukládání dat, která jsou potřeba pouze pro použití cílené aplikace. [1] [2]

1.3 Externí úložiště

Externí úložiště je vhodné pro uživatelská data, která mají být zpřístupněna uživateli nebo jiným aplikacím. Soubory, které jsou uloženy na externím úložišti, se nesmažou, i když je aplikace ze zařízení zcela odstraněna. Externí úložiště se skládá ze standardních veřejných adresářů. Soubory uložené na externím úložišti jsou ve výchozím nastavení čitelné pro každého a lze je upravit povolením velkokapacitního úložiště přenosem souborů do počítače přes USB.

Je také třeba vzít v úvahu, že externí úložiště není vždy k dispozici, někdy může být prezentováno ve formě vyměnitelné SD karty. Před pokusem o přístup k souboru v externím úložišti je zapotřebí zkontrolovat dostupnost adresářů externího úložiště a souborů v něm. Soubory je možné také ukládat na externí úložiště, kde je systém odstraní, když uživatel aplikaci odinstaluje. [1] [2] Nové modely telefonů v posledních letech nemají vestavěný slot na SD kartu, jelikož se zvyšuje kapacita úložiště samostatného zařízení.

1.4 Ukládání dat do cache paměti

Pokud je třeba data dočasně uložit, jsou k tomu využity interní soubory mezipaměti. Každá aplikace má vyhrazený a soukromý adresář mezipaměti pro ukládání dočasných souborů. Je důležité si uvědomit, že systém Android může tyto soubory smazat, když zařízení již nemá prostor na vnitřním úložišti, takže není bezpečné do tohoto prostoru ukládat nic jiného než dočasné soubory. Neexistuje také žádná záruka, že Android tyto soubory odstraní, takže tento adresář je nutné pravidelně udržovat v podobě vyčištění mezipaměti zařízení, protože to ovlivňuje celkový výkon zařízení.

Dobrým případem použití dočasných souborů je nahrávání obrázků na server, v situaci, kdy uživatel nemá zájem, aby byl obrázek uložen v zařízení. V tomto případě dočasný soubor funguje spolehlivě, kdy obrázek je uložen právě do tohoto dočasného souboru, poté dojde k nahrání na server a po kontrole úspěšného nahrání se soubor smaže jako nepotřebný. [1] [3]

1.5 SQLite databáze

Použití souborů a SharedPreferences jsou dva skvělé způsoby, jak aplikace může ukládat malá množství dat. Někdy však aplikace potřebuje ukládat velké množství dat, a to strukturovanějším způsobem, což obvykle vyžaduje použití databáze. Ve výchozím nastavení se systém správy databáze (DBMS), který platforma Android používá, nazývá SQLite. SQLite je knihovna, která poskytuje DBMS založenou na jazyce SQL. Mezi charakteristické vlastnosti SQLite patří:

- Použití dynamických datových typů pro tabulky. To znamená, že lze uložit hodnotu do libovolného sloupce, bez ohledu na datový typ.
- Pomocí jediného databázového připojení lze přistupovat k více databázovým souborům současně.
- Možnost vytváření in-memory databází, se kterými se pracuje velmi rychle. [1]

Android poskytuje rozhraní API potřebná k vytváření a interakci s databázemi SQLite v balíčku *android.database.sqlite*.

I když jsou tato rozhraní API výkonná a známá, jsou na nízké úrovni a jejich použití vyžaduje určitý čas a úsilí. V současné době se místo toho doporučuje používat knihovnu *Room*

persistence, která poskytne abstrakční vrstvu pro přístup k datům v SQLite databázích aplikace.

Jednou z nevýhod použití SQLite API je absence ověřování nezpracovaných SQL dotazů v době kompilace, a pokud se změní struktura databáze, je třeba dotčené dotazy aktualizovat ručně.

Další nevýhodou je, že je nutné napsat spoustu opakujícího se standardního kódu pro připojení a transformaci SQL dotazů a datových objektů. [1] [4]

1.6 Základní operace v databázi (CRUD)

CRUD je zkratka pro Create, Retrieve, Update a Delete, tedy vytvoř, zobraz, změň a odstraň. Toto jsou základní operace, které lze provádět s databází pro ukládání, modifikaci a používání dat.

Create tato operace přidá nový záznam do databáze. V SQL nebo Structured Query Language je toho dosaženo pomocí příkazu *INSERT INTO*.

Retrieve operace slouží pro čtení dat a dotazuje se do databáze tím, že provádí vyhledávání mezi záznamy, které odpovídají určitým kritériím a může vracet hodnotu null. V SQL se využívá k tomutu účelu příkaz *SELECT FROM*.

Operace Update provede změny existujícího záznamu nebo sady záznamů, které odpovídají určitým kritériím. SQL používá příkaz *UPDATE*.

Operace Delete se používá k odstranění záznamů, které splňují určitá kritéria. V SQL se k odstranění záznamu používá příkaz *DELETE FROM*.

Ke specifikaci kritérií v SQL se používá klauzule *WHERE* spolu s výše uvedenými příkazy. Pro odstranění opakujícího se standardního kódu poskytuje Android užitečnou třídu nazvanou *SQLiteOpenHelper*, která zjednoduší integraci databáze. Umožňuje také podrobné studium SQL dotazů v jeho nejčistší podobě. [1] [5]

2 ARCHITEKTURY APLIKACÍ

Při vývoji skutečné mobilní aplikace, která má dynamický charakter a bude rozšiřovat své možnosti podle potřeb uživatele, není možné psát hlavní logiku v activity nebo fragmentech. Aby bylo možné strukturovat kód projektu a dát mu modulární design (samostatné části kódu), používají se architektonické šablony k oddělení zájmů. [6]

Moderní velké projekty využívají několik architektonických šablon najednou, v závislosti na jejich požadavcích. Proto je důležité znát všechny základní funkce každé moderní šablony.

2.1 MVC (Model – View – Controller)

Šablon Model View Controller je vzor, který odděluje součásti softwarového systému na základě odpovědností. Tato šablona má tři součásti s různými odpovědnostmi definovanými takto:

- Model je datová vrstva, která zahrnuje datové objekty, databázové třídy a další obchodní logiku související s ukládáním dat, načítáním dat, aktualizací dat apod.
- Pohled je vrstva vizualizace dat využívající model uživatelského rozhraní. Jinými slovy, toto je to, co se uživateli zobrazí.
- Řadič je mozkem systému, a to tak, že zapouzdřuje logiku systému a spravuje model i pohled. Uživatel komunikuje se systémem prostřednictvím tohoto ovladače. [7]

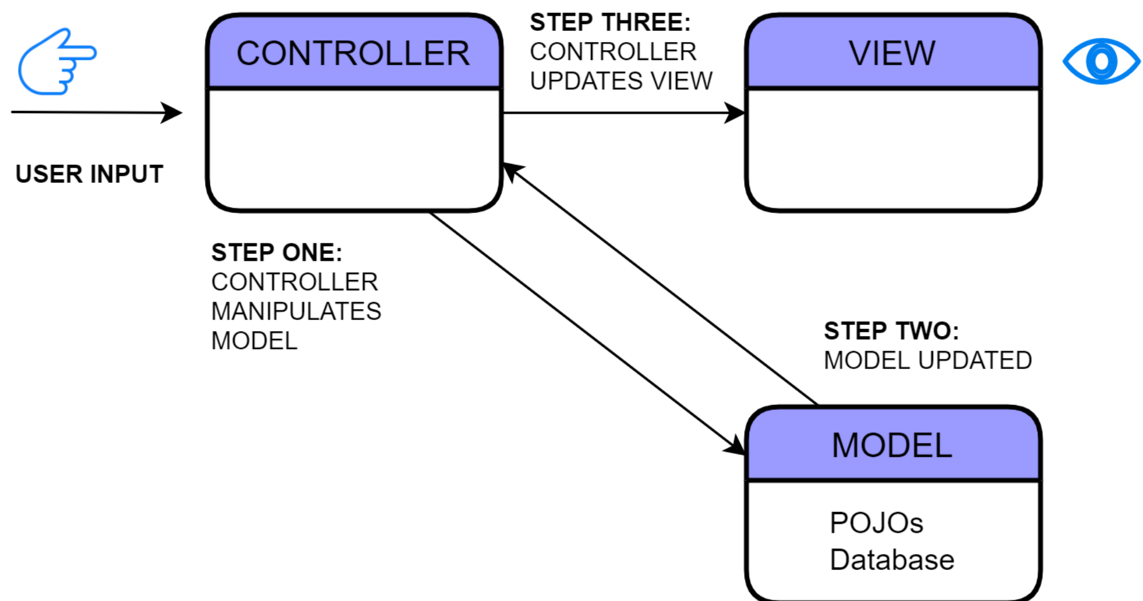
Když byl MVC původně konceptualizován, byl aplikován na desktopové a webové aplikace. [7]

V MVC je hlavním vstupním bodem do aplikace řadič, takže má smysl přidělit roli ovladače aktivitě Android, kde může přijímat uživatelský vstup, jako je kliknutí na tlačítko, a reagovat následujícím způsobem. Model se skládá z datových objektů, což jsou v Androidu obvyklé datové třídy Kotlin, a také třídy pro místní a vzdálené zpracování dat. Zobrazení se skládá ze souborů xml v aplikaci pro Android. [7]

S touto příliš zjednodušenou adaptací MVC na Android je však problém. Když řadič aktualizuje pohled, pohled se nemůže aktualizovat, pokud se jedná pouze o statické rozvržení .xml. Místo toho by jakákoli akce měla téměř vždy obsahovat nějaký druh prezentační logiky, jako je zobrazení nebo skrytí pohledů, zobrazení ukazatele průběhu nebo aktualizace textu na obrazovce v reakci na vstup uživatele. Také ne všechna rozvržení jsou inflated pomocí .xml,

jako když *Aktivita* démonicky načítá rozvržení. Pokud má akce obsahovat odkazy na pohledy a logiku, kterou je třeba změnit, a také veškerou logiku jejích odpovědností správce, pak akce v této šabloně účinně slouží jako řadič i pohled. [7]

Použití aktivity jako ovladače a zobrazení je problematické ze dvou důvodů. Zaprvé je to v rozporu s účelem MVC, kterým bylo rozdělit odpovědnost mezi tři samostatné součásti softwarového systému. Za druhé, použití aktivity jako regulátoru v MVC vytváří problém pro testování jednotek. [6] [7]



Obrázek 1:.Realistická aplikace modelu MVC pro Android

Zdroj: Přepracováno podle [7]

Využití architektury MCV je možné bez následků v ukázkových aplikacích, nebo v malých řešeních, která je možné a vhodné otestovat částečně ručně.

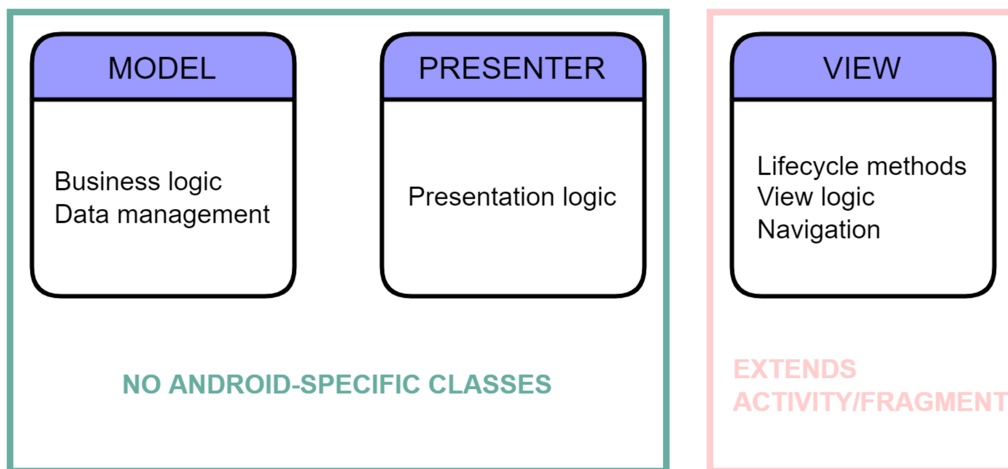
2.2 MVP (Model – View – Presenter)

Jak jste viděli v kapitole MVC, zatímco vzor architektury teoreticky umožňuje aplikaci dosáhnout oddělení problémů a schopnosti testování jednotek, MVC na Androidu tak docela v praxi nefunguje. Problém byl v tom, že *Aktivita* Androidu bohužel sloužila jako zobrazení i jako ovladač. V ideálním případě by bylo požadováno nějakým způsobem přesunout ovladač z *Activity* do jeho vlastní třídy, aby bylo možné ovladač otestovat. [7]

Jedním z architektonických vzorů, které tohoto cíle dosahují, je MVP, což je zkratka pro Model View Presenter a skládá se z následujících částí:

- Model je datová vrstva zodpovědná za obchodní logiku.
- Pohled zobrazuje uživatelské rozhraní a naslouchá akcím uživatele. Obvykle se jedná o aktivitu (nebo fragment). [7]
- Prezentátor komunikuje s modelem i pohledem a stará se o prezentační logiku.

Zatímco pohled dědí z *Activity* nebo *Fragment*, Model a Prezentátor nedědí třídy specifické pro platformu Android a z velké části by neměly obsahovat třídy specifické pro platformu Android. Toto pravidlo umožňuje, aby byl model a prezentátor testován s tímto vzorem. [7]

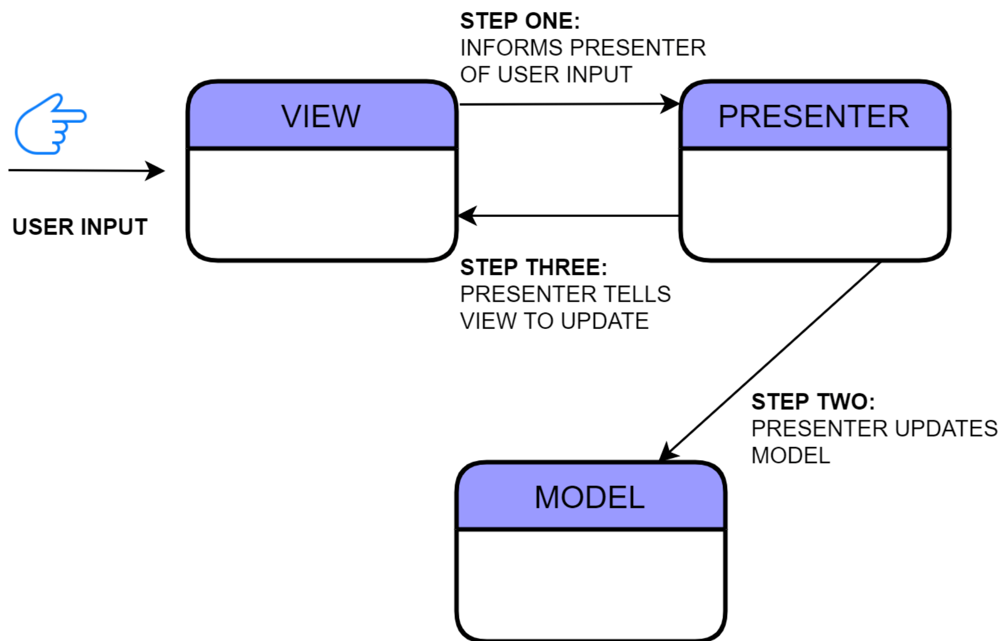


Obrázek 2:.Presenter a Model nesmí obsahovat třídy specifické pro platformu Android

Zdroj: Přepracováno podle [7]

Ve vzoru MVP jsou tedy role modelu a pohledu ekvivalentní těm v MVC, kromě toho, že aktivita nebo fragment je nyní explicitně přiřazena roli pohledu. Řadič byl nyní nahrazen třídou Prezentátor, která podle výše uvedené definice plní stejnou roli jako teoretický řadič. [7]

Existuje však několik klíčových charakteristik vzoru MVP, které jej odlišují od MVC a umožňují mu pracovat na Androidu: pořadí, ve kterém se události vyskytují, oddělení pohledů a modelů a použití rozhraní. Na rozdíl od MVC, kde řadič musí být hlavním vstupním bodem pro aplikaci, v MVP je hlavním vstupním bodem pohled. [7]



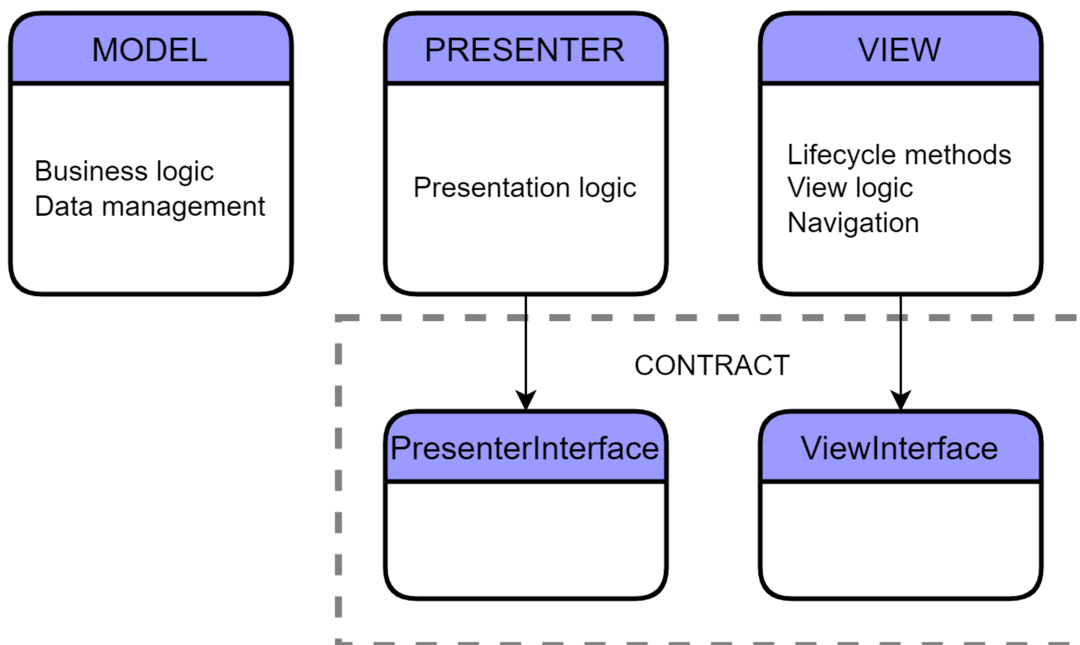
Obrázek 3:.Model MVP

Zdroj: Přepracováno podle [7]

S výjimkou tříd Android Architecture Component, které jsou platné v presenterech, nesmí presenter obsahovat odkazy na žádné třídy specifické pro platformu Android, jako je Context, View nebo Intent. Toto pravidlo umožňuje psát pravidelné testy JUnit pro prezentujícího. Presenter však samozřejmě potřebuje komunikovat se zobrazením, což znamená, že potřebuje odkaz na aktivitu, třídu specifickou pro platformu Android. Ale při testování jednotek se musíte odkázat na tuto třídu. [7]

Způsob, jak tento problém vyřešit, je vytvořit rozhraní pro Presenter a Pohled, přičemž rozhraní ponecháte v jediné třídě zvané Contract class. Třída Presenter implementuje PresenterInterface a *Activity* implementuje *ViewInterface*. [7]

Třída Presenter pak bude obsahovat odkaz na instanci ViewInterface spíše než odkaz na samotnou aktivitu. Presenter a Pohled spolu tedy komunikují spíše prostřednictvím rozhraní než skutečných implementací, což je obecně dobrý princip návrhu softwaru, aby byly tyto dvě třídy odděleny. V tomto případě má také další výhodu odebrání třídy *Activity* specifické pro Android z Presenter, který umožňuje simulovat ViewInterface v Presenter pro testování jednotek. [6] [7]



Obrázek 4:.Vytváření rozhraní pro Presenter a View

Zdroj: Přepracováno podle [7]

Je to ideální řešení pro malé aplikace, ale při použití ve velkých projektech se stává nutností psát spoustu dalšího kódu, který je v budoucnu obtížné logicky uložit.

2.3 MVVM (Model – View – ViewModel)

MVVM je zkratka pro Model-View-ViewModel. MVVM je architektonický vzor, jehož hlavním cílem je dosáhnout oddělení zájmů jasným vymezením rolí každé z jeho vrstev:

- Pohled zobrazuje uživatelské rozhraní a informuje ostatní vrstvy o akcích uživatele.
- ViewModel poskytuje informace k pohledu.
- Model získává informace ze zdroje dat a poskytuje je ViewModels. [7]

Na první pohled je MVVM velmi podobný architektonickým vzorům MVP a MVC. Hlavní rozdíl mezi MVVM a těmito vzory je v tom, že je věnována zvláštní pozornost tomu, aby ViewModel neobsahoval žádné odkazy na pohledy. ViewModel poskytuje pouze informace a je mu jedno, co je spotřebovává. To usnadňuje vytvoření vztahu jeden k mnoha, kde mohou pohledy vyžadovat informace z jakéhokoli modelu pohledu, který potřebují. [7]

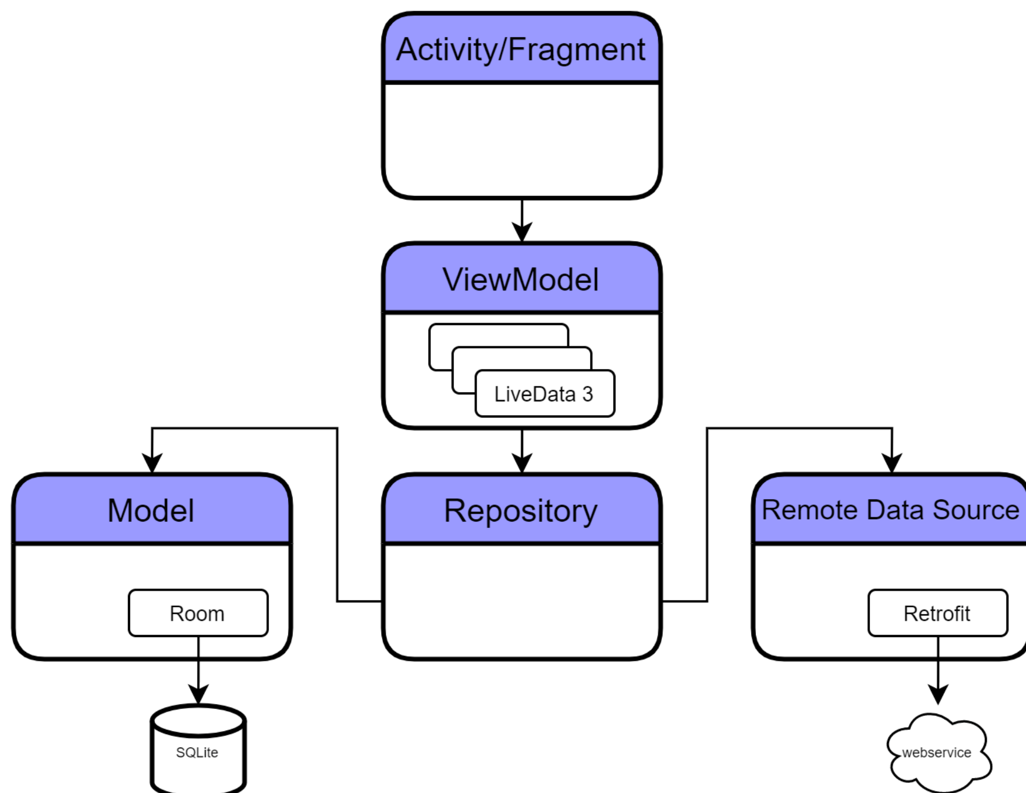
Také je třeba poznamenat, že pokud jde o vzor architektury MVVM, je ViewModel také zodpovědný za odhalování událostí, které mohou pohledy pozorovat. Tyto události mohou být

tak jednoduché, jako je nový uživatel ve databázi, nebo dokonce aktualizace celého seznamu adresářů. [7]

Jedním z problémů, který se vyskytuje v některých architekturách, zejména v MVC, je to, že obchodní logiku je poměrně obtížné testovat kvůli nedostatečnému oddělení od prezentační logiky. Omezením veškeré manipulace s daty na model zobrazení a jeho zachováním bez kódu zobrazení se obchodní logika stává použitelnou pro testování jednotek, protože ji lze spustit bez použití běhového prostředí Android. [7]

Dalším problémem se vzorem MVC je to, že obvykle existují nesrovnalosti v tom, jaký kód by měl kam jít. Někdy, když se kód nevejde do modelu nebo pohledu, je vložen do ovladače. To často vede k běžnému problému známému jako „fat controllers", který způsobuje, že třídy kontrolérů jsou příliš velké a obtížně se udržují. [7]

MVVM řeší problém fat controllers tím, že poskytuje lepší oddělení problémů. Přidání ViewModelů, jejichž hlavním účelem je být zcela odděleno od pohledů, snižuje riziko příliš velkého množství kódu v jiných vrstvách. [7]



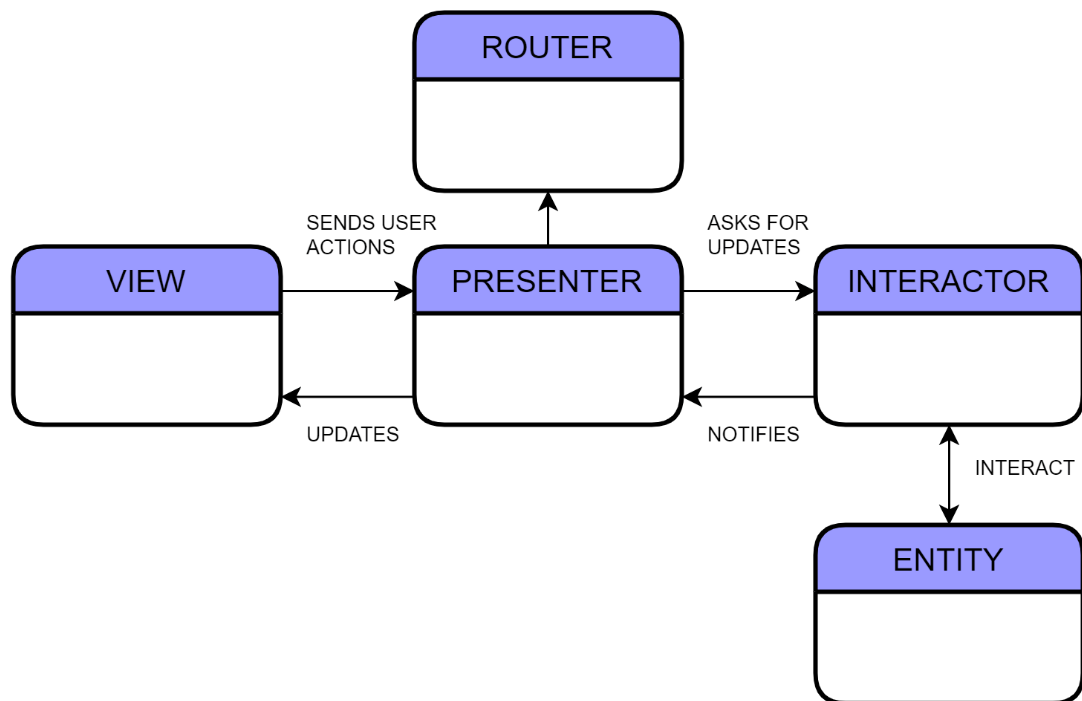
Obrázek 5:.Model MVVM

Zdroj: Přepracováno podle [14]

Hlavní nevýhodou tohoto architektonického vzoru je, že může být příliš složitý pro aplikace s poměrně jednoduchým uživatelským rozhraním. Přidání maximální úrovně abstrakce k takovým aplikacím může vést ke standardnímu (opakujícímu se) kódu, který pouze ztíží pochopení základní logiky. [6] [7]

2.4 VIPER (View – Interactor – Presenter - Router)

VIPER je zkratka pro View, Interactor, Presenter, Entity a Router. Všechny předchozí vzory se spoléhaly na tři úrovně abstrakce a VIPER na pět. Tato pětivrstvá struktura je navržena tak, aby vyrovnala pracovní zátěž mezi různými entitami a poskytla nejvyšší úroveň modularity. [7]



Obrázek 6:.Model VIPER

Zdroj: Přepracováno podle [7]

Role každého z nich je následující:

- Pohled zobrazuje uživatelské rozhraní a informuje předvádějícího o akcích uživatele.
- Interactor provádí jakékoli akce a logiku spojenou s entitami.
- Presenter působí jako vedoucí oddělení. Říká pohledu, co má zobrazit, určuje trasu, kterou se má přejít na další obrazovky, a informuje uživatele o všech nezbytných aktualizacích.

- Entita představuje data aplikace.
- Směrovač se stará o navigaci v aplikaci. [7]

Stejně jako ostatní architektonické vzory má VIPER za cíl vytvořit čistší a modulárnější strukturu, která izoluje závislosti aplikace, zlepšuje udržovatelnost a testovatelnost. [7]

VIPER je skvělý architektonický vzor zaměřený na to, aby každá třída v aplikaci získala jedinečnou oblast odpovědnosti. Poskytuje nejvyšší úroveň abstrakce mezi vzory, které zde byly doposud popsány, jako je MVC, MVP nebo MVVM. [7]

Vhodné použití VIPER:

- Kódová základna se dramaticky rozroste v krátkém časovém období.
- Je zapotřebí využít šablonu architektury, která usnadňuje a urychluje přidávání nových funkcí.
- Během vývoje je potřeba neustále psát unit testy.
- Ladění velkých tříd ještě většími metodami. [7]

Nevhodné použití VIPER:

- Pět různých úrovní abstrakce pro vzor architektury VIPER může způsobit napsání příliš mnoho standardního (opakujícího se) kódu při spouštění nového projektu, což zpomalí raný vývoj.
- VIPER je využit na nevhodných projektech, tedy takových, které mají poměrně jednoduchou kódovou základnu. V tomto případě mohou být vhodnější jiné vzory, jako je MVC nebo MVP. [7]

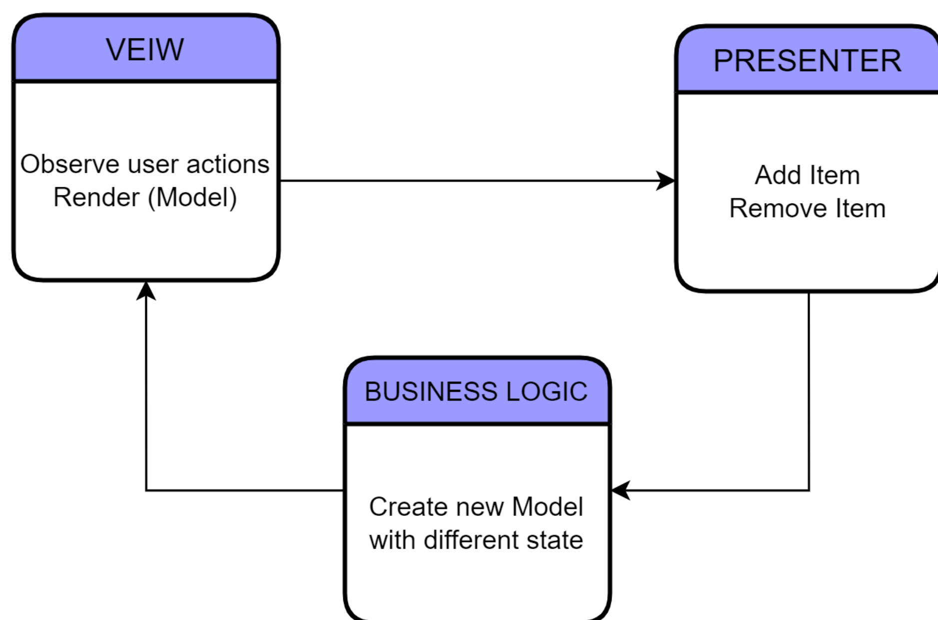
2.5 MVI (Model – View – Intent)

MVI je zkratka pro Model-View-Intent. MVI je jedním z nejnovějších architektonických vzorů pro Android. Architektura byla inspirována jednosměrnou a cyklickou povahou frameworku Cycle.js a do světa Androidu ji přenesl Hannes Dorfaman. [7]

MVI funguje úplně jiným způsobem než jeho vzdálení příbuzní jako MVC, MVP nebo MVVM.

Role každé z jeho složek je následující:

- Model představuje stav. Modely v MVI musí být neměnné, aby umožňovaly jednosměrný tok dat mezi nimi a ostatními vrstvami architektury.
- Intent představuje záměr nebo přání uživatele provést akci. Pro každou akci uživatele obdrží pohled záměr, který bude prezentující pozorovat a převeden do nového stavu v modelech.
- Presenter stejně jako v MVP jsou reprezentovány rozhraními, která jsou následně implementována do jedné nebo více aktivit nebo fragmentů. [7]



Obrázek 7:.Model MVI

Zdroj: Přepracováno podle [7]

Hlavní výhody MVI jsou:

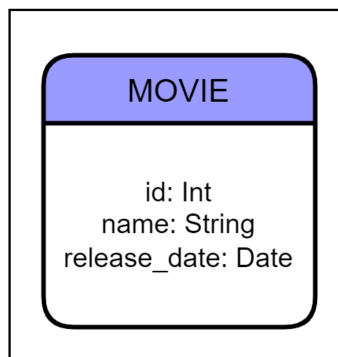
- Jednosměrný a kruhový tok dat pro aplikaci.
- Stabilní stav po celou dobu životního cyklu pohledu.
- Neměnné modely, které poskytují robustní chování a bezpečnost vláken ve velkých aplikacích. [7]

Pravděpodobně jedinou nevýhodou používání MVI namísto jiných architektonických vzorů pro Android je to, že křivka učení tohoto vzoru bývá o něco vyšší, protože potřebujete mít slušné množství znalostí v jiných středně/pokročilých tématech, jako je reaktivní programování, multitasking, vlákna a RxJava. Proto mohou být jiné architektonické vzory, jako je MVC nebo MVP, pro začínající vývojáře androidů jednodušší. [7]

3 OBJEKTOVĚ RELAČNÍ MAPOVÁNÍ

ORM jsou nástroje, které vám umožňují získávat, odstraňovat, ukládat a aktualizovat obsah relační databáze pomocí programovacího jazyka podle výběru. ORM jsou implementovány jako knihovny nebo rámce, které poskytují další vrstvu abstrakce nazývanou datová vrstva, která umožňuje lépe interagovat s databází pomocí syntaxe podobné objektově orientovanému světu. [1]

Pro kvalitnější pochopení, jak fungují ORM, je zde uveden příklad na třídě Movie se třemi vlastnostmi: id, name a release_date.



Obrázek 8:.Filmová třída s vlastnostmi

Zdroj: Přepracováno podle [1]

Toto je diagram tříd představující třídu Movie. Stejně jako u většiny objektově orientovaných jazyků má každá z těchto vlastností specifický datový typ, například Int, String nebo Date. Pomocí ORM lze tuto třídu Movie snadno použít k vytvoření nové tabulky v databázi. V ORM jsou třídy tabulkou v databázi a každá vlastnost je sloupec. Například předchozí třída Movie by byla převedena na tuto tabulku: [1]

id	name	release_date
1	Interception	2010-07-08
2	The Dark Knight	2008-07-14

Obrázek 9:.Filmova tabulka

Zdroj: Přepracováno podle [1]

Každý ze sloupců bude mít také datový typ, který nejlépe reprezentuje původní datový typ původní vlastnosti. Například String bude přeložen jako varchar a Integer jako Int. Způsob vytváření nových záznamů v tabulkách se u každé implementace liší. Některé ORM například automaticky vytvářejí nové položky pokaždé, když je vytvořena nová instance třídy. Jiné ORM, jako je Room, používají k dotazování tabulek Data Access Objects nebo DAO. [1]

3.1 OrmLite

Object Relational Mapping Lite poskytuje některé odlehčené funkce pro zachování objektů Java v databázích SQL, přičemž se vyhýbá složitosti a režii standardních balíčků ORM. *ORMLite* přímo volá rozhraní API databáze Android pro přístup k databázím SQLite. [10]

3.2 SugarORM

Knihovna *Sugar ORM* byla postavena tak, aby měla jednoduchý, stručný a čistý integrační proces s minimální konfigurací. Automatické pojmenování tabulek a sloupců prostřednictvím reflexe. A podporu migrací mezi různými verzemi schémat. [11]

3.3 GreenDAO

GreenDAO je open – source Android ORM, díky kterému je vývoj databází SQLite velmi rychlý. To zbavuje vývojáře nízkoúrovňových požadavků na databáze a šetří čas na vývoj. [12]

3.4 Active Android

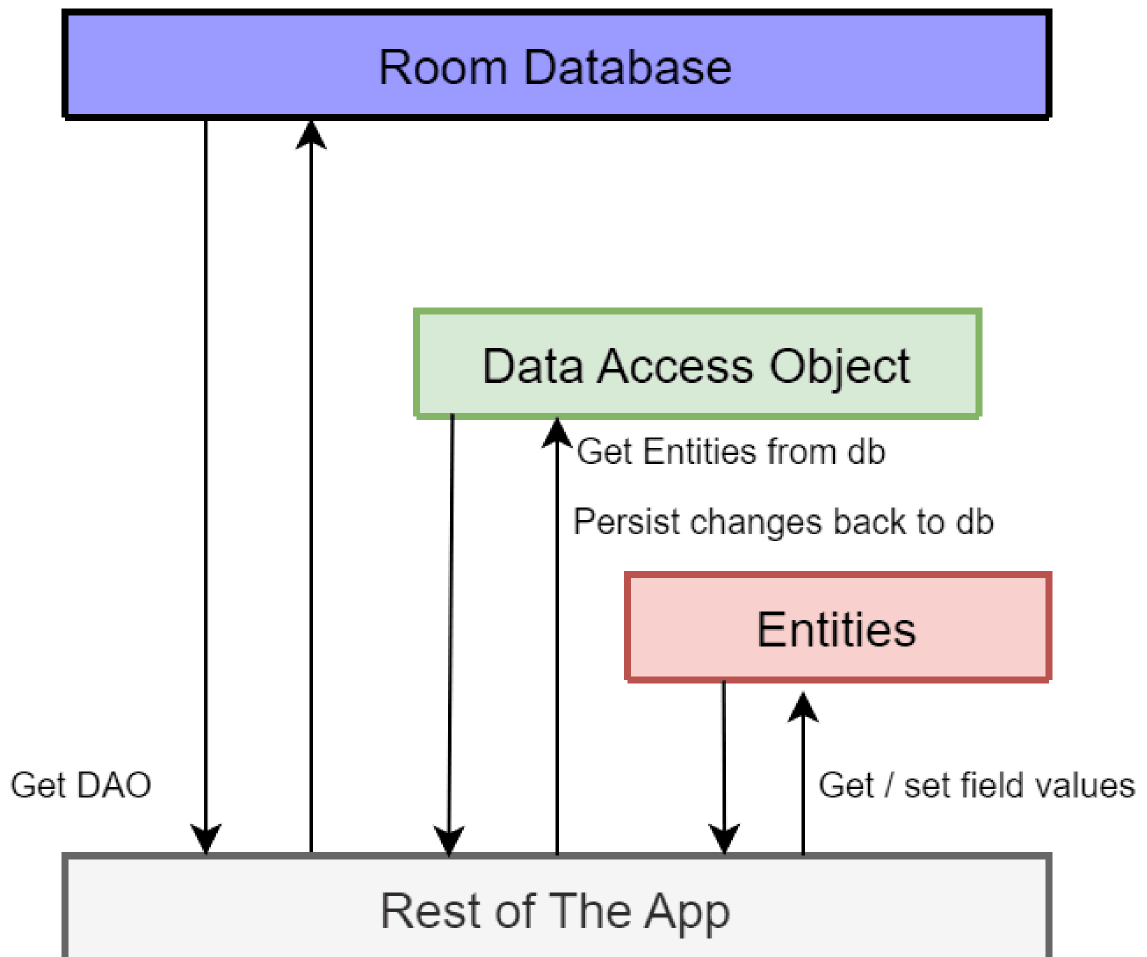
ActiveAndroid funguje jako jakýkoli objektový relační mapovač mapováním tříd Java na databázové tabulky a mapováním proměnných členů třídy Java na sloupce tabulky. Prostřednictvím tohoto procesu se každá tabulka mapuje na model Java a sloupce v tabulce představují příslušná datová pole. Podobně každý řádek v databázi představuje určitý objekt. To umožňuje vytvářet, upravovat, mazat a dotazovat se v databázi SQLite pomocí objektů modelu namísto surového SQL. [13]

3.5 Room perzistentní knihovna

Na začátku roku 2017 Google představil knihovnu *Room* persistence library jako součást jetpack compose.

3.5.1 Room architektura

Na zařízení jsou data uložena v lokální databázi SQLite. *Room* poskytuje nad rámec běžných SQLite API další vrstvu, která se vyhne psaní velkého množství standardního (opakujícího se) kódu pomocí třídy SQLiteOpenHelper. [1] [8]



Obrázek 10:.Schéma architektury knihovny Room

Zdroj: Přepracováno podle [8]

3.5.2 Definice entity

Entity v *Room* představují tabulky v databázi a jsou obvykle definovány v Kotlin jako datové třídy. *Room* vám poskytuje mnoho anotací, které vám umožňují definovat, jak bude datová třída převedena na tabulku SQLite. [1] [8]

Entity v *Room* jsou definovány řadou různých anotací třídy. Níže jsou uvedeny nejběžnější anotace, které budete používat při definování objektu:

Anotace *@Entity* deklaruje, že anotovaná třída je entitou *Room*. Pro každou entitu je v přidružené databázi SQLite vytvořena tabulka pro uložení dat. Ve výchozím nastavení *Room* vždy používá jako název tabulky název třídy, ale lze použít vlastnost tableName v anotaci *@Entity* k nastavení jiného názvu. [1] [8]

3.5.3 Druhy vztahů

Relace typu 1:1, vztah jedna ku jedné mezi dvěma objekty je vztah, ve kterém každá instance nadřazeného objektu odpovídá přesně jedné instanci podřízeného objektu a naopak.

Relace typu 1:N, vztah jedna k mnoha mezi dvěma objekty je vztah, ve kterém každá instance nadřazeného objektu odpovídá nule nebo více instancím podřízeného objektu, ale každá instance podřízeného objektu může odpovídat pouze jedné instanci nadřazeného objektu.

Relace typu M:N, vztah mnoho pro mnoho mezi dvěma objekty je vztah, ve kterém každá instance nadřazeného objektu odpovídá nule nebo více instancím podřízeného objektu a naopak. [9]

3.5.4 Šablona DAO

DAO je zkratka pro Data Access Object. DAO jsou rozhraní nebo abstraktní třídy, které *Room* používá k interakci s databází pomocí anotovaných funkcí. DAO obvykle implementují jednu nebo více operací CRUD na konkrétní entitě. [1]

Objekty pro přístup k databázi jsou běžně známé jako DAO. Objekty DAO jsou objekty, které poskytují přístup k datům aplikace a díky nimž je *Room* tak výkonná, protože odstraňují velkou část složitosti interakce se skutečnou databází. Použití DAO místo tvůrců dotazů nebo přímých dotazů usnadňuje interakci s databází. Tím je možné se vyhnout všem potížím s laděním dotazů. Poskytují také lepší oddělení zájmů pro vytvoření strukturovanější aplikace a zlepšení testovatelnosti. [1]

V *Room* jsou DAO definovány jako rozhraní nebo abstraktní třídy. Jediný rozdíl mezi oběma implementacemi je v tom, že abstraktní třída může dodatečně přijmout instanci *RoomDatabase* jako parametr konstruktoru. Jsou také užitečné pro definování velkých databázových transakcí, pomocí anotace *@Transaction* na metodě a interního volání několika různých metod. [1]

DAO jsou definovány jako abstraktní třídy nebo rozhraní, protože *Room* se stará o vytvoření každé implementace DAO v době kompilace a generuje kód obchodní logiky pro definice. Ve

výchozím nastavení *Room* nepodporuje přístup k databázi v hlavním vláknu, protože dlouhotrvající operace, jako jsou transakce databáze, mohou způsobit zablokování nebo selhání aplikace. [1]

4 NÁVRH

Tato kapitola popisuje návrh a analýzu mobilní aplikace pro platformu Android. Hlavním účelem je zpracování a prezentace dat získaných od společnosti SpaceX, která se soustředí na poskytování služeb spojených s vesmírnými a předorbitálními lety. V další kapitole praktické části práce je popsáno propojení backendové části aplikace s mobilním řešením pomocí technologie API.

4.1 Analýza

V této podkapitole jsou vypsány funkční a nefunkční požadavky a vybrané technologie, které tyto požadavky splňují. V oblasti vývoje softwaru slouží funkční požadavky ke stanovení cílů pro vývojáře. A to nejen název úkolů, ale i celý obsah všech možných odpovědí na otázky, které mohou během procesu vývoje vyvstat. Obvykle se skládají ze tří položek, a to očekávání od vývojáře, scénáře použití funkce a nástroj pro vizualizaci nápadů. Případ užití ukazuje, co uživatel v roli, která mu byla přidělena, dělá pro dosažení konkrétního výsledku, a co k tomu potřebuje. Případy užití jsou popisem chování uživatele při interakci s vyvíjeným produktem, jinými slovy při přechodu na funkcionalitu.

Vývoj aplikace pro tuto práci si jako hlavní cíl bere praktickou ukázku, jak ukládat data, výstupní aplikace tak v reálném světě nemá uplatnění, ale způsob a postup vývoje je přínosem pro trvalé ukládání dat na platformě Android. Níže jsou proto uvedeny pouze tabulky s funkčními a nefunkčními požadavky.

4.1.1 Funkční požadavky

ID	Popis funkčního požadavků
FR1	K přenosu a výměně informací bude použit koncový bod SpaceX API.
FR2	Načítání, perzistentní ukládání a prezentace firemních informací.
FR3	Načítání, perzistentní ukládání a prezentace o všech letech společnosti.
FR4	Načítání, perzistentní ukládání a prezentace o všech raketách vlastněných společností.
FR5	Zobrazení podrobnějších informací o jednotlivých letech.
FR6	K přepínání mezi sekcí bude použita navigační panel (navigation drawer).
FR7	Filtrování letů podle různých parametrů (všechny, úspěšné, neúspěšné).
FR8	Při delším stahování se objeví ukazatel průběhu.

Tabulka 1:.Funkční požadavky

Zdroj: Vlastní zpracování

4.1.2 Nefunkční požadavky

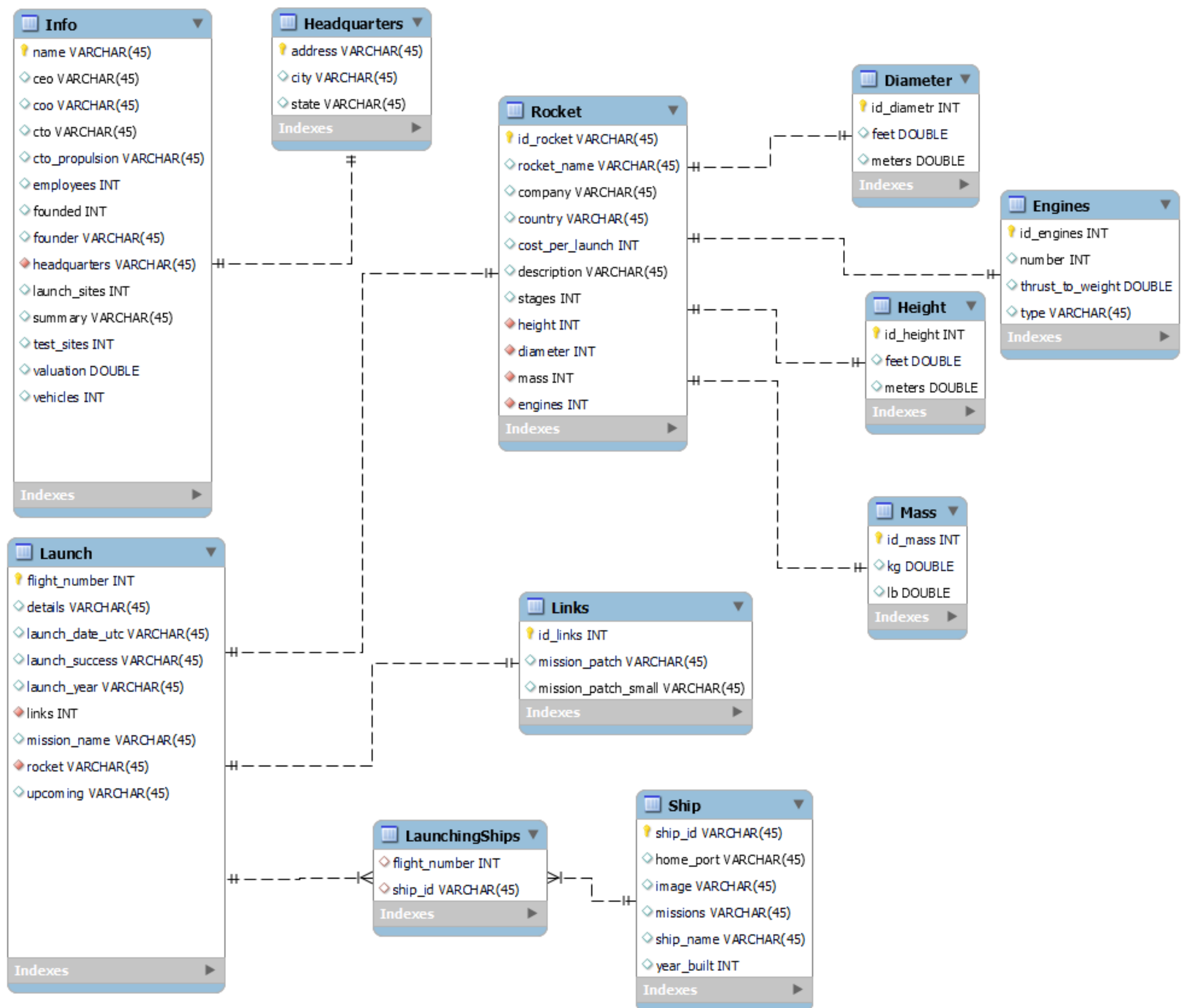
ID	Popis nefunkčního požadavků
NFR1	Aplikace bude naprogramována v jazyce Kotlin.
NFR2	Návrh bude splňovat všechna doporučení (Material design).
NFR3	Minimální úroveň SDK musí být 29, což odpovídá Androidu 10 kódové označení Red Velvet Cake.
NFR4	Cílové SDK verze 31 odpovídá Androidu verze 12 nebo Snow Cone.
NFR5	Všechny obrazovky budou používat architekturu MVVM.
NFR6	Bude používa externí knihovna Retrofit pro ověřování a interakci s SpaceX API.
NFR7	Pro parsování odpovědi ze serveru v formátu JSON do kódu v Kotlině bude použita externí knihovna Moshi.

Tabulka 2:.Nefunkční požadavky

Zdroj: Vlastní zpracování

4.2 Návrh databáze

Na základě požadavků popsaných v podkapitole Analýza byl vyvinut model relační databáze. Výsledný datový model obsahuje celkem 5 tabulek (Info, Launch, Ship, LaunchingShips, Rocker) a dále embedded tabulky.



Obrázek 11:.Relační databázový model

Zdroj: Vlastní zpracování v programu MySQL Workbench

5 IMPLEMENTACE

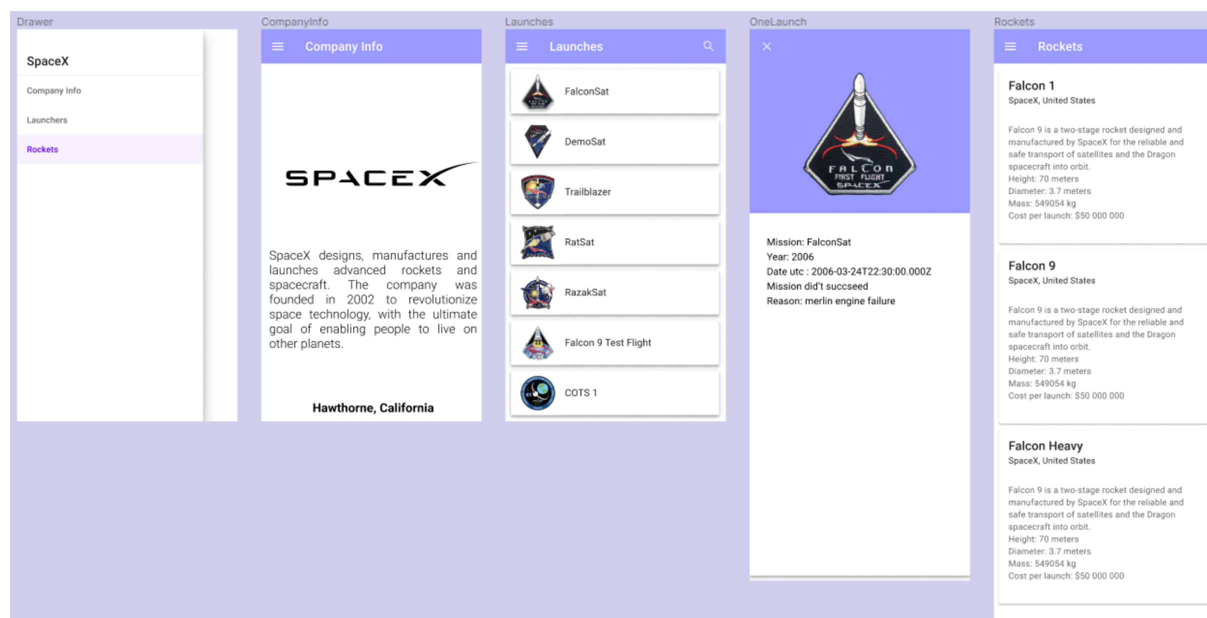
Tato kapitola pojednává o nástrojích používaných k vývoji a na příkladech vysvětluje, jak řešit konkrétní problémy.

Pro implementaci byl použit programovací jazyk Kotlin, který od května 2019 Google deklaruje jako preferovaný jazyk pro vývoj na platformě Android. V době psaní této práce byla Java zcela nahrazena Kotlinem. [18]

Vývojové prostředí použité při implementaci praktické části Android Studio Bumblebee. Figma byla použita k vytvoření grafických návrhů. Na základě kapitoly Architektury aplikací byl zvolen architektonický vzor MVVM. Jednou z výhody použití při vývoji této aplikace byla dobrá integrace Android Architecture Componets a snadné přidávání dalších funkčních modulů v budoucnu při rozšiřování funkčnosti aplikace.

5.1 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je při vývoji aplikací velmi důležité, protože je jedním z hlavních ukazatelů úspěchu. Pokud aplikace nevypadá graficky poutavě, pak ji uživatel nebude s oblibou používat tak často. Splnění tohoto požadavku je usnadněno předpisy společnosti Google s názvem Material design. Tento grafický nástroj poskytuje rozvržení pro hojně využívané prostředí pro vývoj designu aplikací.

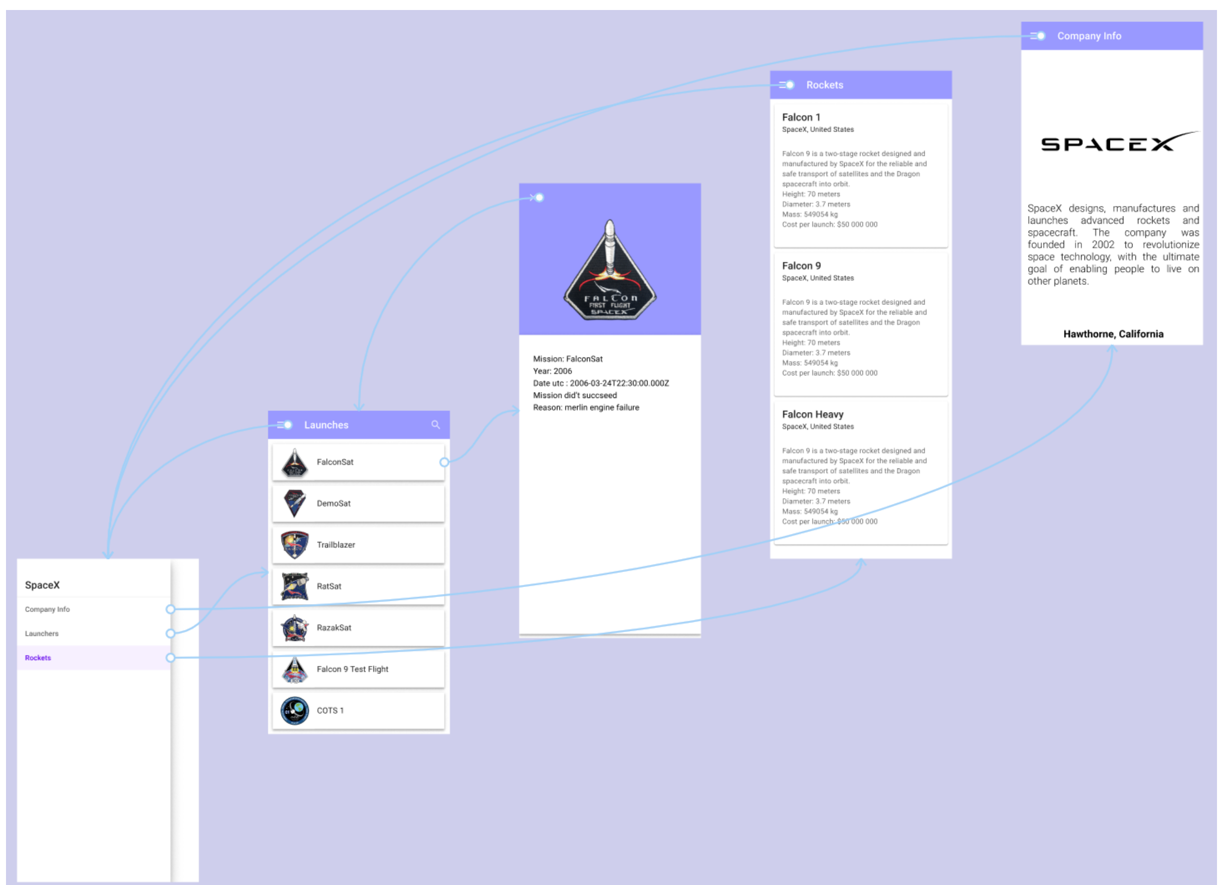


Obrázek 12:.Grafický návrh aplikace SpaceX

Zdroj: Vlastní zpracování v programu Figma

5.1.1 Navigace v aplikaci

Navigace je implementována pomocí komponenty Navigation drawer. Vizuální přechody mezi obrazovkami jsou na obrázku 13 znázorněny jako modré čáry. Přechody se provádějí v závislosti na nastavení zařízení pomocí tlačítek nebo gest. Jak gesta, tak tlačítka byla úspěšně testována.



Obrázek 13:.Grafické znázornění navigace v aplikace SpaceX

Zdroj: Vlastní zpracování v programu Figma

5.2 Připojení k REST API a zpracování výsledků

Práce s aplikací REST API zahrnuje dvě důležité součásti. Retrofit usnadňuje získání a načtení JSON prostřednictvím webové služby založené na REST. Retrofit konfiguruje, který převodník se použije k serializaci dat. Druhou součástí je právě převodník. V současné době je nejoblíbenější díky své široké funkčnosti Moshi.

Zdrojový kód 1 ilustruje inicializace singletonové třídy a přístupové funkce *makeRetrofitService*, která poskytuje klientovi Retrofit s přednastaveným převodníkem JSON do Kotlinu.

```
object RetrofitClient {
  const val BASE_URL = "https://api.spacexdata.com/v3/"

  val moshi = Moshi.Builder()
    .addLast(KotlinJsonAdapterFactory())
    .build()
  val apiClient: SpaceXApiServices = Retrofit.Builder()
    .baseUrl(BASE_URL)
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .build()
    .create(SpaceXApiServices::class.java)

  fun makeRetrofitService(): SpaceXApiServices {
    return apiClient
  }
}
```

Zdrojový kód 1:.RetrofitClient.kt

Zdroj: Vlastní zpracování

5.3 Perzistentní ukládání s využitím knihovny Room

Knihovna Room vyžaduje definici databázových tabulek, definici objektů přístupu k datům, aby se zabránilo přímé komunikaci s databází, a definici vztahů mezi tabulkami.

5.3.1 Databázový model

K vytvoření tabulky *Room* navrhuje použít anotace. Vzhledem k tomu, že konvence jazyků Kotlin a sql jsou odlišné, je požadovaný název tabulky napsán v uvozovkách a předán jako parametr u anotace *@Entity*, výchozí je název třídy.

Anotace *@ColumnInfo* se používá ze stejných důvodů jako anotace *@Entity*. Pokud název proměnné v Kotlin neodporuje názvu v SQL, pak zodpovědnost přebírá anotace *@Entity*, která chápe, že vše v konstruktoru jsou sloupce, a automaticky to vygeneruje.

Anotace *@PrimaryKey* je samovysvětlující a používá se u proměnné, která bude sloužit jako primární klíč v tabulce.

V neposlední řadě. Je použita velmi důležitá anotace *@Embedded*. V modelech pomáhá udržovat kód čistý a ukazuje vnoření jedné tabulky do druhé, nebo jinými slovy umožňuje používat typy dat, který není definován v SQL.

Všechny výše uvedené anotace a jejich použití jsou uvedeny ve Zdrojovém kódu 2.

```
@Entity(tableName = "launches_table")
data class LaunchLocal constructor(
   val details: String?,
   @PrimaryKey
   @ColumnInfo(name = "flight_number")
   val flightNumber: Int?,
   @ColumnInfo(name = "launch_date_utc")
   val launchDateUtc: String?,
   @ColumnInfo(name = "launch_success")
   val launchSuccess: Boolean?,
   @ColumnInfo(name = "launch_year")
   val launchYear: String?,
   @Embedded
   val links: Links?,
   @ColumnInfo(name = "mission_name")
   val missionName: String?,
   val rocket: String?,
   val upcoming: Boolean?
)

@Entity(tableName = "launches_table")
data class Links constructor(
   @ColumnInfo(name = "mission_patch")
   val missionPatch: String?,
   @ColumnInfo(name = "mission_patch_small")
   val missionPatchSmall: String?
)
```

Zdrojový kód 2:.LaunchLocal.kt

Zdroj: Vlastní zpracování

5.3.2 Data access object

Proč jsou potřeba objekty DAO, je popsáno v kapitole Šablona DAO. K deklaraci objektu je použita anotace *@Dao*. K provádění příkazů s tabulkou je využit *@Insert* a *@Query*. Specifikované ve Zdrojovém kódu 3.

```
@Dao
interface LaunchWithShipDao {
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun insertAll(launches: List<LaunchLocal>)

   @Query("SELECT * FROM launches_table")
   fun getLaunches(): LiveData<List<LaunchLocal>>

   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun insertAllShips(ships: List<ShipLocal>)
}
```

Zdrojový kód 3:.LaunchWithShipDao.kt pro přístup k databázi

Zdroj: Vlastní zpracování

5.3.3 Databázové vztahy

Je-li zapotřebí využitídotazování nad tabulkami se vztahy many-to-many. Jsou vyžadována další datova třída které pomocí anotace *@Relation* označí vztah viz Zdrojový kód 4 a mezitabulka, která je zobrazena na Zdrojový kód 5.

```
data class LaunchesWithShips(
   @Embedded
   val launch: LaunchLocal,
   @Relation(
      parentColumn = "flight_number",
      entityColumn = "ship_id",
      associateBy = Junction(LaunchShipCrossRef::class)
   )
   val ships: List<ShipLocal>
)
```

Zdrojový kód 4:.LaunchesWithShips.kt

Zdroj: Vlastní zpracování

```
@Entity(primaryKeys = ["flight_number", "ship_id"], tableName = "launch_ship_cross_ref")
data class LaunchShipCrossRef(
   @ColumnInfo(name = "flight_number")
   val flightNumber: Int,
   @ColumnInfo(name = "ship_id")
   val shipId: String
)
```

Zdrojový kód 5:.LaunchShipCrossRef.kt propojovací tabulka

Zdroj: Vlastní zpracování

Chcete-li získat přístup k databázi prostřednictvím objektu DAO s relacemi tabulek many-to-many, existují dvě další metody. První se používá k uložení identifikátorů lodí a letů do mezitabulky. Anotace použitá pro vnořování je stejná jako v předchozích funkcích popsaných výše. Metoda pro získání všech lodí používaných při letu na základě identifikačního čísla lodi používá anotaci @*Transaction*. Při použití na neabstraktní metodě abstraktní třídy Dao bude odvozená implementace metody provádět super metodu v databázové transakci Zdrojový kód 6.

```
@Dao
interface LaunchWithShipDao {
  /**
   * Other functions
   */
  @Insert(onConflict = OnConflictStrategy.REPLACE)
  fun insertLaunchShipCrossRef(crossRef: LaunchShipCrossRef)

  @Transaction
  @Query("SELECT * FROM launches_table WHERE flight_number = :flight_number")
  fun getShipsWithLaunches(flight_number: Int): LiveData<LaunchesWithShips>
}
```

Zdrojový kód 6:.LaunchWithShipDao.kt pro přístup k tabulce se vztahy many to many

Zdroj: Vlastní zpracování

5.3.4 Inicializace databáze

Pro vytvoření databáze byla definována abstraktní třída, která rozšiřuje *RoomDatabase*. Tato třída je anotována *@Database*, uvádí entity obsažené v databázi a DAO, které k nim přistupují. Použití Singleton, což je tvůrčí návrhový vzor, který umožňuje zajistit, aby třída měla pouze jednu instanci, a zároveň k této instanci poskytuje globální přístupový bod. K vytvoření singletonového objektu v jazyce Kotlin stačilo klíčové slovo *objekt*. Objekt typu singleton se inicializuje pouze tehdy, když je požadován poprvé. V našem případě hned po spuštění aplikace. Více zdrojový kód 7.

```
@Database(entities = [
  CompanyInfoLocal::class, LaunchLocal::class, RocketLocal::class,
  ShipLocal::class, LaunchShipCrossRef::class],
  version = 1)
abstract class AppDatabase : RoomDatabase() {
  abstract fun companyDao(): CompanyDao
  abstract fun rocketDao(): RocketDao
  abstract fun launchWithShipDao(): LaunchWithShipDao

  companion object {
    @Volatile
    private var INSTANCE: AppDatabase? = null
    private const val DB_NAME = "AppDatabaseSpaceX"

    fun getDatabase(context: Context): AppDatabase {
      synchronized(this) {
        var instance = INSTANCE

        if (instance == null) {
          instance = Room.databaseBuilder(
            context.applicationContext,
            AppDatabase::class.java,
            DB_NAME
          ).build()
          INSTANCE = instance
        }
        return instance
      }
    }
  }
}
```

Zdrojový kód 7:.AppDatabase.kt

Zdroj: Vlastní zpracování

5.4 Bezpečnost a ochrana dat

Bezpečnost uživatelů je jedním z hlavních požadavků při vývoji moderního softwaru. Platforma Android má své vlastní interní uživatelské bezpečnostní systémy, které musí každý vývojář dodržovat.

Jedním z prvních byl použit permission nebo povolení. K provádění určitých operací nebo přístupu k určitým zdrojům je vyžadováno povolení. Oprávnění mohou být dvou typů, a to normální a nebezpečná. Rozdíl mezi nimi je v tom, že nebezpečná oprávnění mohou být použita k získání osobních údajů.

Normálními oprávněními jsou napřílad používání bluetooth a nfc pro přenos dat, nastavení budíku atd. Všechna oprávnění jsou popsána v souboru *AndroidManifest.xml*. Při vývoji aplikace bylo použito a deklarováno jedno oprávnění z této skupiny, a to oprávnění k přístupu na internet.

Nebezpečná oprávnění, jak již bylo popsáno výše, je skupina, pomocí které lze přijímat jakákoli osobní data, například přístup ke kalendáři, fotoaparátu, kontaktům nebo umístění smartphonu. Při implementaci této aplikace nebyla použita žádná oprávnění z této kategorie.

Vývojáři přidali další ochranu oprávnění od verze Android 11. Nejnebezpečnější oprávnění ke kameře, umístění a mikrofonu jsou vyžadována při každém použití aplikace uživatelem. Pokud aplikace nebyla uživatelem delší dobu používána déle než měsíc, pak platforma automaticky resetuje oprávnění, i když jej uživatel udělil dříve. Po prvním odmítnutí uživatelem se na další dotaz objeví další možnost, která zabrání této aplikaci vyžadovat oprávnění. V důsledku toho není vyžadováno žádné další povolení kromě skutečně nutného připojení k internetu.

Aby se předešlo možnému útoku uživatele s názvem Code Injection. Při implementaci filtrování letů místo textového pole bylo použito rozevírací menu pro zabránění ztrátě a distribuci uživatelských dat. Bylo přijato rozhodnutí nesbírat žádná data od uživatele. V případě nutnosti získat v budoucnu osobní údaje bylo zvažováno řešení pomocí šifrování dat.

Protože aplikace vylučuje možnost instalace z oficiálního obchodu Google Play. Jedná se o bezpečnostní riziko nejen při kompletní výměně aplikace za podobnou, ale i při opravě některých jejích částí. K odstranění této možnosti se používá nástroj s názvem ProGuard. Jednou z užitečných funkcí je kódování celého projektu s výjimkou důležitých částí, jejichž změna by znamenala nefunkčnost. To útočníkovi ztěžuje porozumět zdrojovému kódu a provádět jakékoli změny.

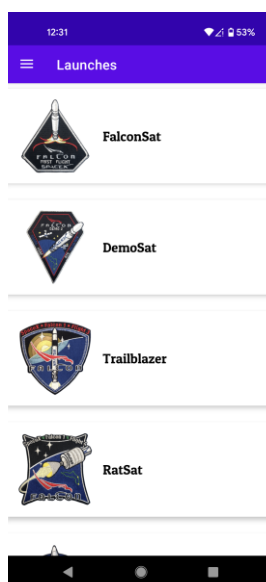
6 TESTOVÁNÍ

Tato kapitola popisuje typy a metody testování používané k ověření funkčnosti aplikace SpaceX. Teoretická testovací pyramida se skládá z doporučených podílů 70 procent unit testů, 20 procent integračních testů a 10 procent end-to-end testů. [17]

V závislosti na typu testování může být nutné použít skutečné mobilní zařízení. V této práci bylo použito zařízení Google Pixel 4a s operačním systémem Android verze 12.

6.1 End – to – end testy

Tyto testy testují velké části aplikace, přesně simulují skutečné použití, a proto jsou obvykle pomalé. Testování musí být provedeno na skutečných zařízeních. Hlavním cílem praktické části je ověřit, zda budou data stažená ze vzdáleného serveru trvale uložena v paměti zařízení. Výchozí situace je taková, že aplikace na zařízení nikdy nebyla spuštěna, takže databáze je prázdná. Při prvním spuštění aplikace dochází k inicializaci databáze a odeslání požadavku na server. Pokud není k dispozici připojení k internetu, zobrazí se pop-up okno, která zobrazí chybovou hlášku. Po úspěšném připojení na internet a přijetí dat, se tato uloží do databáze, ta se následně v aplikaci zpracují a zobrazí se uživateli. V případě odpojení od internetu jsou data stále součástí zařízení, a tak i po restartu aplikace bere data z databáze, samozřejmě pokud není databáze prázdná. Sled akcí pro ověření se skládá z několika kroků.

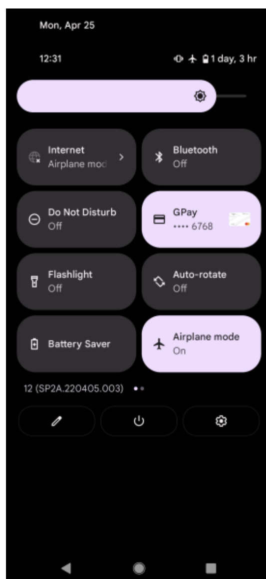


Obrázek 14:.Reprezentace dat přijatých ze SpaceX API

Zdroj: Vlastní zpracování pomocí standardní funkcionality Android OS

Aplikace se spouští s funkčním připojením k internetu, proto se očekává uložení a zobrazení dat přijatých ze serveru, výsledek je ke zhlédnutí na obrázku 14.

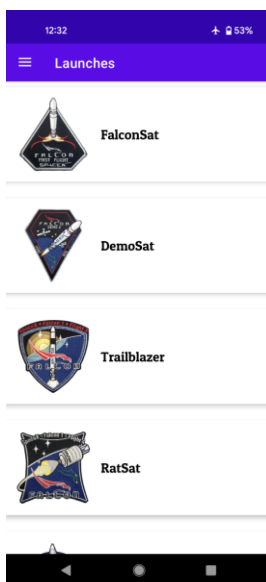
Dalším krokem je simulace nepřítomnosti internet, a to za předpokladu, že databáze již byla naplněna, k tomu stačí přepnout telefon do režimu letadla, to je zobrazeno na obrázku 15.



Obrázek 15:.Aktivace režimu Letadlo

Zdroj: Vlastní zpracování pomocí standardní funkcionality Android OS

Při znovuspuštění aplikaci, je výsledek patrný na obrázku 16, což dává jistotu, že vše funguje podle očekávání.



Obrázek 16:.Reprezentace dat, když je zapnutý režim Letadlo

Zdroj: Vlastní zpracování pomocí standardní funkcionality Android OS

6.2 Integrační testy

Tato kategorie testuje interakci několika tříd tak, aby se při společném použití chovaly podle očekávání. Použití virtuálních a skutečných zařízení je nejlepší strategií pro tuto kategorii testů. Testování je rychlejší než end-to-end testy díky použití automatického přístupu.

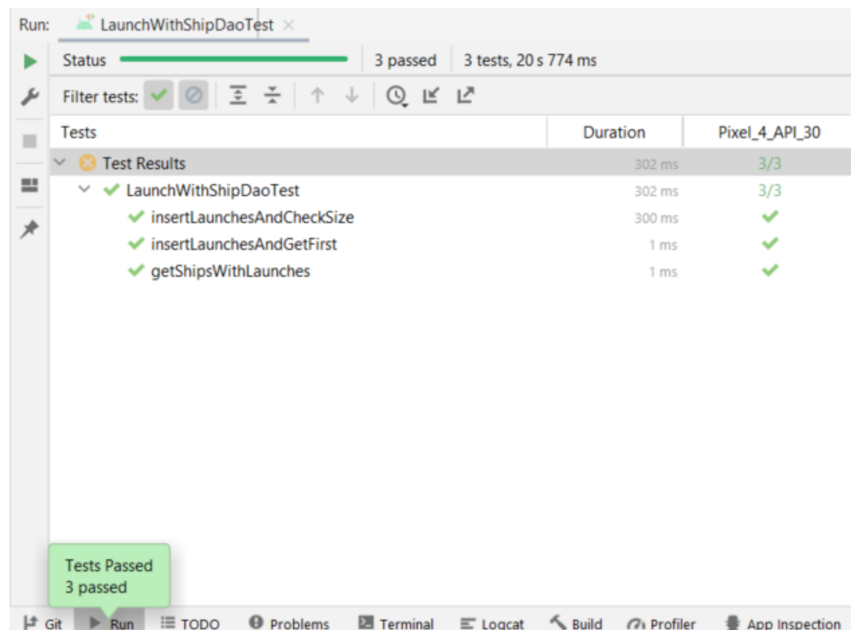
Testování třídy LaunchWithShipDao probíhá pomocí dodatečně vytvořené třídy se stejným názvem a slovem Test na konci. Uvnitř třídy jsou vytvořeny metody, které reprezentují ty testované. V těle metody jsou volány funkce a výsledek je porovnáván s očekáváním. Pro snadnou implementaci je použit framework JUnit. Cílem je otestovat základní funkce přístupu k databázi.

Funkce insertLaunchesAndGetFirst přidá do databáze falešný seznam letů a poté požádá o první vložený let. Získaný výsledek se porovná s tím, který se umístil jako první. Hlavním účelem je zkontrolovat, zda byl první let skutečně uložen jako první.

Následující test nazvaný insertLaunchesAndCheckSize uloží falešný seznam letů do databáze v určitém počtu a poté se do databáze dotáže na počet záznamů, které obsahuje. Motivací testu je zjistit, zda byly skutečně uloženy všechny požadovaný lety.

Poslední test se nazývá getShipsWithLaunches a jeho účelem je uložit falešný seznam letů a poté uložit falešný seznam lodí používaných k přepravě. Protože je to vztah mnoho k mnoha. Další vztahy jsou také zachovány. Při znalosti počtu lodí použitých k přepravě rakety s identifikačním číslem 1 je do databáze zadán dotaz, kolik lodí bylo použito a počet je porovnán s přijatým.

Po spuštění všech testů, úspěšný výsledek je znázorněn na Obrázku 17.



Obrázek 17:.Výsledky spuštění třídy LaunchWithShipDaoTest.kt

Zdroj: Vlastní zpracování v programu Android Studio

6.3 Unit testy

Rozsah unit testů je malý a zajišťují, že izolované jednotky kódu fungují tak, jak se předpokládalo. Unit testy by měly testovat jednu proměnnou a nespoléhat se na žádné externí závislosti. U unit testů je také velmi běžné využití plné funkčnosti frameworku Junit. Poskytuje tvrzení k identifikaci testovací metody. Tento nástroj nejprve otestuje data a poté je vloží do části kódu. Jedná se o více automatizované testy, díky nimž jsou tyto testy rychlejší.

Obvykle se používá na nejmenší možné jednotce. Může to být jak třída, tak nejčastěji funkce. Díky tomu se vždy přesně ví, kde se stala chyba.

V této práci bylo hlavním cílem trvalé uchování dat a jejich reprezentace. Proto při vývoji obchodní logiky aplikace hlavní pozornost byla věnována maximálnímu omezení koncového uživatele pro zadávání dalších údajů. Aby se předešlo dodatečné kontrole zadaných údajů, což v konečném důsledku poskytuje další zabezpečení aplikace. Jediným interaktivním místem v aplikaci, které vyžaduje vstup uživatele, je filtrování odpalů raket pomocí rozbalovací nabídky. V této části funkcionality bylo plánováno použití jednotkových testů.

Plné otestování této funkce vyžadovalo tři podobné testovací scénáře, přičemž filtrovací nabídka má tři možnosti, a to zobrazit všechny lety, jen úspěšné a jen neúspěšné. Při výběru jedné z možností je porovnán výsledek s očekáváním. Při výběru jedné z možností navíc dochází ke kontrole a změnu názvu obrazovky na vybranou možnost.

Při pokusu o testování se ukázalo, že při použití komponenty nabídky možností (options menu) není možné otestovat nejmenší část bez ovlivnění výsledku. Což je v rozporu se základním principem unit testování.

ZÁVĚR

Mobilní aplikace umožňující persistentní ukládání dat využijí převážně uživatelé, kteří nemají potřebu být stále online. Důvody mohou být různé, ve vyspělých zemích není problém s pokrytím území signálem na rozdíl od zemí třetího světa. Obrovský problém je s cenovým aspektem kvůli velkým monopolům, které si za své služby účtují obrovské ceny. V důsledku těchto problémů je trvalé úložiště dat řešením pro všechny kategorie uživatelů.

Stanovený cíl bakalářské práce byl naplněn jak po stránce teoretické tak praktické.

Teoretické aspekty byly využity při tvorbě praktické části, kdy nejdříve došlo ke kompletní analýze funčkních a nefunkčních požadavků, ze kterých vyplynulo, že mobilní aplikace by měla být vyvíjena na některé z moderních architektur. Důležité bylo také použití určitých technologií podporovaných vývojářskou komunitou pro řešenou oblast a výrobcem operačního systému. Ve fázi implementace bylo zjištěno, že některé moderní knihovny nemají podrobnou dokumentaci, která by vysvětlovala složitější příklady. Často se proto autor musel obracet s žádostí o radu a vysvětlení na pokročilejší programátory.

V průběhu testování bylo zjištěno, že všechny cíle praktického výstupu byly splněny. V rámci testování autor došel také závěru, že bylo nevýhodné použití druhého hlavního klíče s hodnotou řetězce, v mezitabulce s vztahy many-to-many. Tím pak vznikají problémy s výkonem při velkém počtu přístupů k tabulce, což ovšem v tomto praktickém případě nehraje velkou roli kvůli malému množství dat přijímaných ze serveru, ale v případě rozšíření praktického výstupu je dobré tomuto věnovat. Další možností rozšíření je použití doplňkových informací z backendových služeb pro rozšíření funkčnosti. Vnitřní úpravy využívající již uložené informace umožňují sestavit statistiku úspěšnosti letu.

Při zpracování bakalářské práce si autor osvojil více technologií pro práci s perzistentním úložištěm dat a práci s externími službami. Dále se zdokonalil ve vytváření spojení a parsování přijatých dat pomocí nástroje Moshi a Retrofit.

POUŽITÁ LITERATURA

[1] BAILEY, Jennifer, Dean DJERMANOVIć, Aldo Olivares DOMINGUEZ, Fuad KAMAL, Subhrajyoti SEN a Harun WANGEREKA. *Saving Data on Android: learn Jetpack DataStore, Room, Firebase & SQLite with Kotlin*. Second edition. McGaheysville: Razeware, 2021. ISBN 978-1950325436.

[2] Data and file storage overview. *Developer.android.com* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://developer.android.com/training/data-storage

[3] Access app-specific files. *Developer.android.com* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://developer.android.com/training/data-storage/app-specific

[4] Save data using SQLite. In: *Developer.android.com* [online]. 2021 [cit. 2022-03-07]. Dostupné z: https://developer.android.com/training/data-storage/sqlite

[5] RIZAL, Muhammad. Android SQLite Database and Content Provider Tutorial. *Academia.edu* [online]. 2013, 3-6 [cit. 2022-03-07].

[6] Android Architecture Patterns. In: *Geeksforgeeks.org* [online]. [cit. 2022-03-07]. Dostupné z: https://www.geeksforgeeks.org/android-architecture-patterns/

[7] CHENG, Yun a Aldo Olivares DOMÍNGUEZ. *Advanced Android App Architecture: Real-world app architecture in Kotlin 1.3*. First Edition. McGaheysville: Razeware, 2019. ISBN 978-1942878698.

[8] Save data in a local database using Room. In: *Developer.android.com* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://developer.android.com/training/data-storage/room

[9] Define relationships between objects. In: *Developer.android.com* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://developer.android.com/training/data-storage/room/relationships

[10] WATSON, Gray, 2021. ORMLite: Using With Android. *Ormlite.com* [online]. [cit. 2022-04-18]. Dostupné z: https://ormlite.com/javadoc/ormlite-core/doc-files/ormlite.html#Use-With-Android

[11] SATYA. Sugar ORM. *Satyan.github.io* [online]. [cit. 2022-04-18]. Dostupné z: https://satyan.github.io/sugar/

[12] GreenDAO: Android ORM for your SQLite database, 2020. *Greenrobot.org* [online]. [cit. 2022-04-18]. Dostupné z: https://greenrobot.org/

[13] PARDO, Michael. ActiveAndroid. *Activeandroid.com* [online]. [cit. 2022-04-18]. Dostupné z: http://www.activeandroid.com/

[14] DHANANIA, Yashovardhan, 2020. Using the Android MVVM Pattern with Firebase.

In: *Medium.com* [online]. [cit. 2022-04-20]. Dostupné z: https://medium.com/firebase-developers/android-mvvm-firestore-37c3a8d65404

[15] WALKER, Andy, 2021. Report: The average Android phone offered nearly 100GB storage in 2020. *Androidauthority.com* [online]. [cit. 2022-04-23]. Dostupné z: https://www.androidauthority.com/average-smartphone-storage-1213428/

[16] Distribution of worldwide YouTube viewing time as of 2nd quarter 2021, by device, 2021. *Statista.com* [online]. [cit. 2022-04-23]. Dostupné z: https://tinyurl.com/ywb4bym4/

[17] *Concept: Testing Strategy* [online], 2021. [cit. 2022-04-25]. Dostupné z: https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-test-doubles#2

[18] LARDINOIS, Frederic. Kotlin is now Google's preferred language for Android app development. *Techcrunch.com: techcrunch.com* [online]. 2019 [cit. 2022-04-25]. Dostupné z: https://tinyurl.com/tb9vf7b4/

PŘÍLOHY

Příloha A – Mobilní aplikace – SpaceX

Obsahuje:

1. Zdrojový kód aplikace
2. Instalační soubor apk
3. Postup pro nasazení praktického výstupu