VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Katedra technických studií

Implementace mobilní aplikace umožňující navigaci po budově za pomoci navigačních dat

bakalářská práce

Autor práce: Jiří Nenáhlo

Vedoucí práce: Ing. Marek Musil

Jihlava 2020

Vysoká škola polytechnická Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Jiří Nenáhlo**
Studijní program:	Elektrotechnika a informatika
Obor:	Aplikovaná informatika
Název práce:	**Implementace mobilní aplikace umožňující navigaci po budově za pomoci navigačních dat**
Cíl práce:	Cílem práce je vytvoření mobilní aplikace pro platformu Android, která na základě dat ve formátu JSON a grafických podkladů (uchovaných na serveru ve formátu svg a stažených ze serveru ve formátu png) bude schopna pomoci uživateli při pohybu po budově. Aplikace požadovaná data získá ze serveru přes REST API a bude schopna provozu offline. V aplikaci bude možnost vyhledat danou místnost či událost v budově, zobrazit podrobnější informace o daném místě a najít cestu mezi dvěma zadanými místy. Výchozí místo může být specifikováno zadáním nejbližší místnosti. Aplikaci bude možno použít pro jakoukoli budovu.

Ing. Marek Musil
vedoucí bakalářské práce

doc. Ing. Zdeněk Horák, Ph.D.
vedoucí katedry
Katedra technických studií

Abstrakt

Práce se zabývá tvorbou aplikace pro operační systém Android, která pomáhá s navigací po budově díky výpočtu nejkratší cesty mezi místnostmi. Pro výpočet cesty se využívá algoritmu A*. Díky tomuto dynamickému výpočtu cest není aplikace limitována pouze na jednu konfiguraci budovy. Aplikace získává aktuální navigační data ze serveru, tudíž je pak možno jednoduše upravovat navigační data. Synchronizace dat je řešena přes REST API s cachováním poskytnutých dat pro offline provoz v nativní Android databází SQLite.

Klíčová slova

Android, Astar, Kotlin, Neverlost, Pathfinding

Abstract

Project describes creation of an Android application that aims to help users with navigation around a given building faster by calculating the shortest path between any given places inside of it. Those calculations are done using A* algorithm. Thanks to the dynamic pathfinding possible with A*, this project is not limited to one specific building configuration. The application gets its up-to-date navigation data from a server, allowing the administrator to change the navigation data. REST API is used for data synchronization with the addition of response caching in Android’s native SQLite database, therefore enabling the application to function offline.

Key words

Android, Astar, Kotlin, Neverlost, Pathfinding

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Souhlasím s umístěním bakalářské práce v knihovně VŠPJ a s jejím užitím k výuce nebo k vlastní vnitřní potřebě VŠPJ.

Byl/a jsem seznámen/a s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutím licence.

V Jihlavě dne

…….
Podpis studenta

Poděkování

Děkuji Ing. Marku Musilovi za odborné vedení práce, vstřícnost a trpělivost. Dále děkuji kolegům z firmy Etnetera a.s za cenné rady, zkušenosti a v neposlední řadě kamarádovi Danielovi Matějkovi za ochotu a tvorbu backendu pro aplikaci, která je tématem této bakalářské práce.

Obsah

Seznam obrázků

Seznam použitých zkratek

API	Application Programming Interface
DAO	Database Access Object
FAB	Floating Action Button
IDE	Integrated Development Environment
MVVM	Model-View-ViewModel

Úvod

Bakalářská práce popisuje tvorbu aplikace, která slouží jako pomocník pro navigaci budovou. Nápad na tuto aplikaci se zrodil během studia na VŠPJ, když jsem zjistil, že často po výuce předmětu trávím čas zjišťováním místa výuky následující hodiny. Řekl jsem si, že by tento proces potřeboval zefektivnit. V tu chvíli se nabízely dvě řešení. Zapamatovat si označení a rozložení učeben po škole, nebo využít digitálních plánů budovy a vytvořit interaktivní mapu, která bude schopna ukázat cestu mezi učebnami.

Jelikož jsem se již dříve v rámci vývoje her setkal s problematikou pathfindingu (hledání nejkratší cesty mezi dvěma body), řekl jsem si, že využiji těchto znalostí a vytvořím Android aplikaci řešící problém navigace po škole. Android jsem si vybral proto, že tento systém běžel na mém prvním smartphonu, začínal jsem se na této platformě učit jazyk Java a líbilo se mi, že mohu nahlédnout do samotného kódu systému (kromě proprietárních Google aplikací). Mobilní zařízení trpí slabším výkonem hardwaru, tudíž jsem zvolil pro výpočet cesty algoritmus A*. Vzhledem k jeho výskytu v počítačových hrách, kde je rychlost prioritou, jsem usoudil, že s provozem algoritmu na mobilním hardwaru nebude problém. Na aplikaci jsem začal pracovat v rámci předmětu Programování pro mobilní platformy. Tehdy vznikl funkční koncept aplikace, který se dá ještě o mnohé funkce rozšířit a vylepšit.

Aplikace mi už teď ve fázi funkčního prototypu spolehlivě pomáhá s nalezením učeben a myslím si, že by mohla usnadnit pohyb po VŠPJ i ostatním studentům, hlavně těm, kteří nastupují do prvního semestru. Rozhodl jsem se ji v rámci své bakalářské práce rozšířit.

Motivace

Tento projekt vznikl během výuky předmětu Programování pro mobilní platformy jakožto semestrální projekt. Aplikace uměla navigovat jen po budově VŠPJ, ale ušetřila mi čas zkoumáním informačních cedulí a hledání učeben bylo hned snazší. Rozhodl jsem se na tomto konceptu aplikace pracovat i nadále, rozšířit jej o další funkce a během vývoje si rozšířit obzory o další nové postupy a technologie vývoje pro platformu Android.

Cíl práce

Vytvoření mobilní aplikace pro platformu Android, která na základě dat ve formátu JSON a grafických podkladů (uchovaných na serveru ve formátu svg a stažených ze serveru ve formátu png) bude schopna pomoci uživateli při pohybu po budově. Aplikace požadovaná data získá ze serveru přes REST API a bude schopna provozu offline. V aplikaci bude možnost vyhledat danou místnost či událost v budově, zobrazit podrobnější informace o daném místě a najít cestu mezi dvěma zadanými místy. Výchozí místo může být specifikováno zadáním nejbližší místnosti. Aplikaci bude možno použít pro jakoukoli budovu.

1 Úvod do problému, funkční požadavky aplikace

Co se navigace ve vnitřních prostorech týče, kromě mého prototypu aplikace existuje několik již hotových řešení, třeba aplikace Google Indoor Maps, která podporuje rozsáhlé budovy jako jsou například letiště či nákupní centra. Dále aplikace Path Guide, která poměrně detailně zobrazuje itinerář cesty, ale je schopna navigovat pouze po daných předem nahraných cestách ostatních uživatelů. Můj funkční prototyp aplikace není závislý na předem definovaných cestách, ale aby šel jednoduše použít i pro menší budovy s vlastními mapovými podklady, je třeba jej upravit.

Původní verze navigační aplikace pracuje s pevně danými daty nadefinovanými v samotném balíku aplikace, není tu tedy možnost žádné jednoduché úpravy dat. Přitom použitý navigační algoritmus není limitovaný pouze na specifická data. Mezi hlavní úpravy tedy patří propojení aplikace na API, ze kterého bude získávat aktuální navigační data včetně mapových podkladů. Poté nebude limitována pouze na jednu budovu a vše bude záležet na konfiguraci dat na serveru. Samotný server poskytující navigační data s administračním rozhraním vytvořil kolega Daniel Matějka, který jeho implementaci popisuje ve své bakalářské práci. Tyto dvě samostatně vyvíjené části dohromady tvoří projekt **Neverlost**.

Díky výměně statických dat za dynamická bude v aplikaci možnost zobrazovat kromě místností i jednorázové naplánované události. Samotná aplikace se bude lišit pouze grafikou (brandingem), jako je ikona aplikace či barevné schéma prvků uživatelského rozhraní.

Vyhledávání dat bude obohaceno o rychlejší vstupní metodu dat v podobě čtení QR kódu, po jehož přečtení se vyplní název místa do obrazovky vyhledávání.

2 Použité technologie

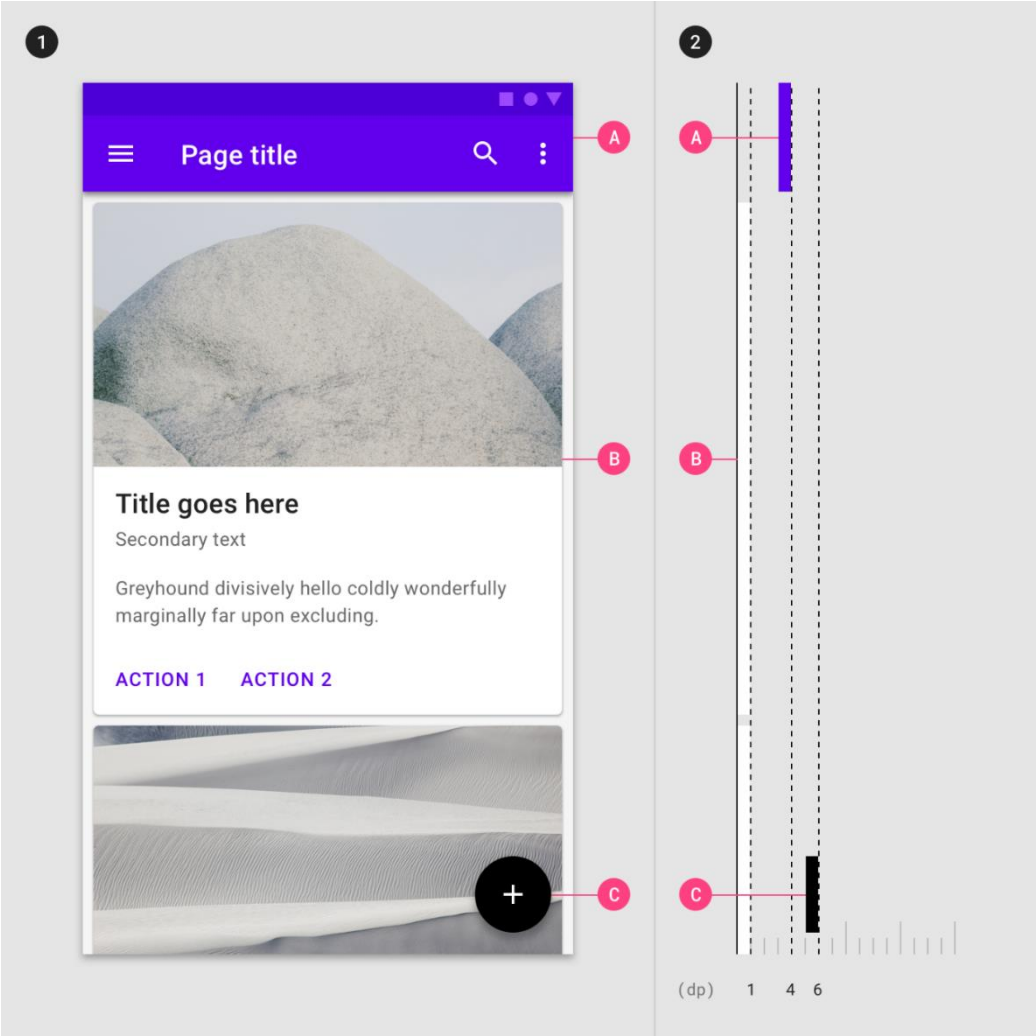
V této kapitole představím několik základních i složitějších technologií, které jsem zakomponoval do aplikace. Vývoj rychle posouvaly kupředu knihovny, díky kterým nebylo nutné psát spoustu generického kódu jen proto, abych aplikaci obohatil o základní funkce (redukce boilerplate kódu).

2.1 Android

Open-source operační systém od společnosti Google, běžící na upraveném Linuxovém jádře. (Code Aurora Forum, nedatováno) V dnešní době je systém určen pro širokou škálu zařízení různých velikostí, od hodinek po televize. (Google, nedatováno)

2.2 Material Design

Adaptivní systém pro návrh moderních uživatelských rozhraní původně vycházející z prostého chování papíru v prostoru. Každý designový prvek si můžeme představit jako list papíru. (Google, nedatováno)



Obrázek 1 – Vrstvení prvků v podání Material Designu (Google, nedatováno)

Jak je patrné z obrázku, vrstvením prvků kolem nich vzniká stín a s ním iluze hloubky.

Barevná paleta se skládá ve většině případů ze dvou odstínů primární a jedné doplňkové barvy. Součástí tohoto designu jsou také animace, které pomáhají zdůraznit reakci aplikace na vstup uživatele.

Narozdíl od předchozích iterací design systému Androidu (Holo) je Material Design univerzální, použitelný nejen pro mobilní aplikace díky schopnosti adaptace stavebních prvků různým velikostem displejů od smartphonů po televize (Android TV). (Google, nedatováno)

2.3 Kotlin

Java, jakožto dlouho používaný jazyk pro vývoj aplikací nejen pro Android, je velmi rozšířený programovací jazyk, ale dal by se určitě vylepšit. Při psaní se u něj setkáváme například s velikým množstvím boilerplate kódu a `NullPointerException` je zde běžná věc v případě neošetřených výrazů. Proto přišel na scénu jazyk Kotlin. Jeho překladač například nedovolí sestavit kód, pokud se v kódu vyskytují případy, které by mohly způsobit pád aplikace odkazováním na proměnnou s hodnotou `null`. Pokud je u proměnné možnost výskytu takového stavu, musí být řádně označena, aby práci s ní mohl překladač kontrolovat. Jinak dané proměnné hodnotu `null` nepřiřadíme. (JetBrains, nedatováno)

Obrázek 2 – Logo programovacího jazyka Kotlin (Breslav, 2016)

Podívejme se na část kódu pro datovou třídu v Javě (zkrácená varianta bez importů, deklarace package a boilerplate kódu v metodách):

```
public class Floor {

    private int mIndex;
    private String mBitmapName;
    private String mName;

    public Floor(int index, String name, String bitmapName) {
        mIndex = index;
        mName = name;
        mBitmapName = bitmapName;
    }

    // ... mIndex getter

    // ... mBitmapUrl getter

    // ... mBitmapName getter

    // ... hashCode() override

    // ... equals() override

}
```

Obrázek 3 – Ukázka třídy reprezentující podlaží v Javě

Stejná třída v jazyce Kotlin (zkrácená varianta bez importů a deklarace package):

```
data class Floor(val index: Int,
                 val name: String,
                 val bitmapName: String)
```

Obrázek 4 – Ukázka `data class` v jazyce Kotlin

Úspora kódu je zde značná. Takových případů je v Kotlinu mnoho. Díky této vlastnosti a celkové modernizaci jazyka oproti Javě jsem se rozhodl přepsat aplikaci do Kotlinu. Gettery/settery jsou generované automaticky, stejně tak jako implementace `hashcode()` a `equals()` pro porovnávání objektů. Stačí jen připsat do deklarace třídy klíčové slovo *data*. Kotlin je s Javou interoperabilní, lze v projektu používat oba jazyky, což se hodí pro užití starších knihoven psaných v Javě. Automaticky generované gettery/settery jsou pro případ volání z Javy pojmenovány podle názvů proměnných s prefixem get/set. (JetBrains, nedatováno)

Vývoje se ujalo české studio JetBrains, které vydalo integrované vývojové prostředí (IDE) IntelliJ IDEA, na čemž je založen IDE vydávaný Googlem, Android Studio – upřednostňovaný software pro vývoj aplikací pro Android. Není tedy divu, že jejich

Kotlin začal Google používat pro vývoj Android aplikací. První verze jazyka byla vydána v roce 2016. (Breslav, 2016)

2.4 REST API

Jeden z mnoha protokolů komunikace mezi serverem a klientskou aplikací za účelem výměny dat. Princip spočívá v získání dat (odpovědi) poté, co klient zašle požadavek na daný endpoint serveru, kde služba poskytuje požadovaná data. Požadavek může obsahovat parametry, na jejichž základě server vyhodnocuje požadavek (autorizační klíče pro použití API, identifikaci zařízení klienta, parametry volané metody).

Retrofit

Nástroj pro usnadnění vývoje Android aplikací, který umožňuje vyřizování síťových požadavků a řeší zpracování dat, přičemž redukuje psaní boilerplate kódu.

```
@GET("getGridData")
fun getGridData(): ImmutableGridData
```

Obrázek 5 – Definice interface pro získávání dat ze serveru pomocí knihovny Retrofit

Retrofit lze pomocí anotací jednoduše aplikovat na další typy požadavků, či jednoduše namapovat parametry funkcí do adresy požadavku či nastavit parametry požadavku v případě požadavků metody POST. (Square, Inc., nedatováno)

JSON

Jednoduchý, univerzální jazyk pro přenos dat mezi počítačovými systémy. Skládá se z objektů a snadno se s jeho daty manipuluje. Objekt může obsahovat různé typy hodnot:

- řetězec
- číslo (celé, desetinné)
- bit (`true`, `false`)
- prázdnou hodnotu (`null`)
- vnořený objekt obsahující další hodnoty
- pole výše uvedených hodnot (Internet Engineering Task Force (IETF), nedatováno)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "gender": "male",
    "age": 32,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumbers": [
        { "type": "home", "number": "212 555-1234" },
        { "type": "fax", "number": "646 555-4567" }
    ]
}
```

Obrázek 6 – Struktura datového formátu JSON (Turi, nedatováno)

Gson

Nástroj pro konverzi textu do instance třídy v objektově orientovaném programovacím jazyce (a naopak). Místo zdlouhavého manuálního parsování struktury odpovědi serveru do požadovaného objektu umí Gson automaticky namapovat JSON strukturu objektu do jeho proměnných buď díky shodě názvu JSON pole a proměnné objektu, nebo podle anotace u dané proměnné, kde je definován její název pole v JSONu. Tento nástroj dokáže eliminovat značnou část boilerplate kódu nutného pro parsování odpovědi ze serveru. (Google, nedatováno)

Abychom využili GSON ke konverzi dat získaných skrze Retrofit, stačí přidat GSON factory pro konverzi dat při tvoření instance Retrofitu:

```
Retrofit.Builder()
      ...
      .addConverterFactory(GsonConverterFactory.create())
      ...
      .build()
```

Obrázek 7 – Propojení knihoven GSON a Retrofit

Pokud se názvy proměnných v objektu i v JSONu neshodují, musíme nadefinovat jejich název v JSONu pomocí anotace `SerializedName`, kde uvedeme řetězec příslušící dané proměnné. Pokud je v projektu použit obfuskátor kódu, může názvy proměnných ve finálním balíku aplikace změnit. Je proto tedy vhodné buď použít tuto anotaci

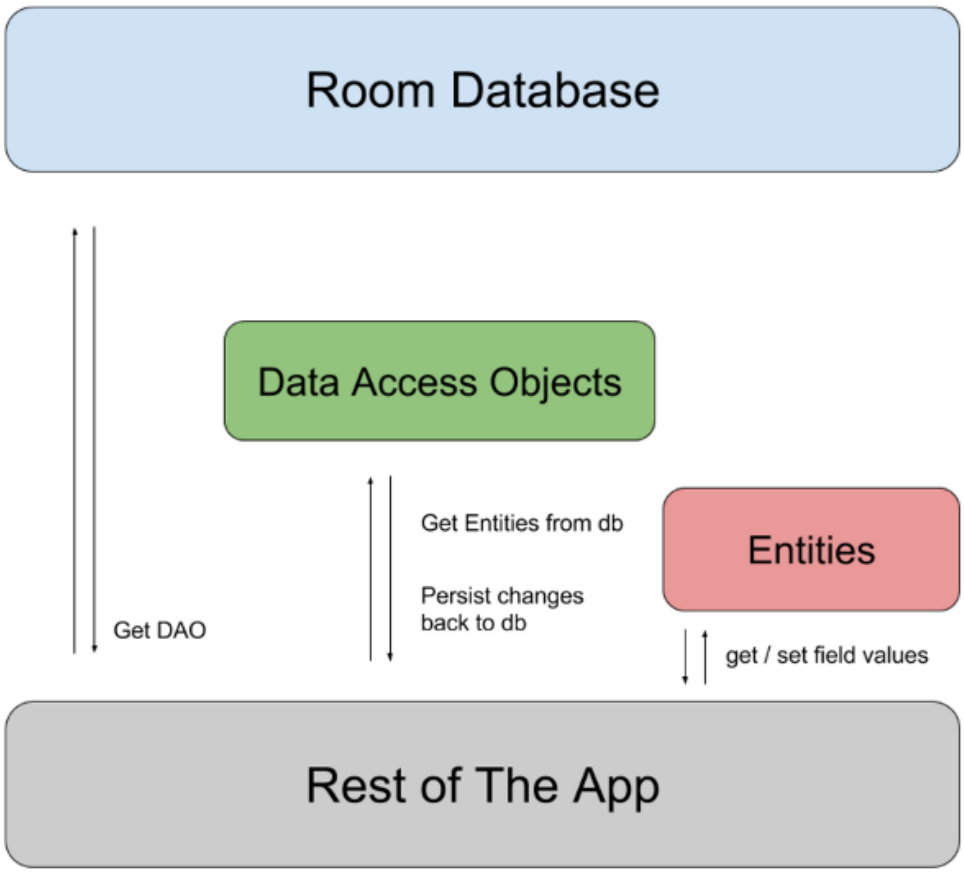
i v případě, že se názvy shodují, nebo nadefinovat výjimku tak, aby se pro třídy reprezentující JSON data obfuskace neprováděla.

2.5 SQLite

Kompaktní, rychlý open-source databázový software psaný v jazyce C (SQLite Consortium, nedatováno). K sestavení nepotřebuje vysoký počet dalších knihoven a je možné celý zdrojový kód získat jako jeden soubor, což nejen že zjednoduší implementaci do větších projektů, ale také umožní překladači lépe optimalizovat kód tohoto databázového softwaru o 5–10 % oproti klasické vícesouborové variantě zdrojového kódu. (SQLite Consortium, nedatováno)

Room

Stejně tak, jako Retrofit usnadňuje práci se zpracováním dat při komunikaci se serverem, knihovna Room usnadňuje práci s lokálními daty v databázi pomocí anotací. Struktura aplikace, která pracuje s objekty v databázi je následující:



Obrázek 8 – Princip funkce databázových objektů knihovny Room (Google, nedatováno)

Místo zdlouhavého psaní boilerplate kódu pro práci s databází nadefinujeme jednoduchý Database Access Object (DAO) pomocí interface, který obsahuje funkce pro práci s objekty v databáze (Google, nedatováno):

```
@Dao
interface ImmutableGridDataDao {
   @Query("SELECT * FROM ImmutableGridData")
   fun loadGridData(): ImmutableGridData

   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun update(gridData: ImmutableGridData)
}
```

Obrázek 9 – DAO interface pro zpracování knihovnou Room

Funkce načítající data (zde `loadGridData`) je označena anotací Query, která na základě SQL dotazu z databáze vrátí požadovaný objekt (v tomto případě `ImmutableGridData`). Název tabulky je ve výchozím stavu název třídy objektu, který metoda vrací.

Funkce s anotací Insert slouží ke vložení dat parametru funkce do databáze. V tomto případě je ještě definována agresivní strategie vložení při konfliktu – přepis existujících dat. Objekt, se kterým DAO pracuje, musíme upravit, aby jej bylo možné zpracovat knihovnou. Přidáme ke třídě anotaci `Entity` a označíme jeden unikátní parametr třídy anotací PrimaryKey, jako kdybychom definovali sloupec s vlastností `PRIMARY KEY` v SQL (Google, nedatováno):

```
@Entity
class ImmutableGridData(@PrimaryKey var id: Int)
```

Obrázek 10 – Anotace vlastního objektu pro zpracování knihovnou Room

Aby byla knihovna schopná ukládat vlastní objekty do databáze, je v některých případech potřeba dodefinovat TypeConvertery. `TypeConverter` třída slouží, jak již název napovídá, ke konverzi dat mezi výstupním objektem dat z databáze (`String`) do námi definované třídy objektu. Vzhledem k tomu, že v projektu pracuji s Gsonem, použijeme tuto knihovnu i pro konverzi dat v databázi následujícím způsobem, ve funkci s anotací `TypeConverter`:

```
class RoomTypeConverters {
   @TypeConverter
   fun toPlaceList(value: String?): List<Place> {
      val type: Type = object : TypeToken<List<Place>?>() {}.type
      return Gson().fromJson(value, type)
   }

   // Convertery pro další objekty
}
```

Obrázek 11 – Ukázka implementace TypeConverteru

Po přípravě anotací a implementace TypeConverterů se dostáváme k poslednímu kroku – samotné definici databáze. Tu provedeme opět další třídou s anotacemi:

```
@Database(
      entities = [ImmutableGridData::class],
      version = 1)
@TypeConverters(RoomTypeConverters::class)
abstract class AppDatabase : RoomDatabase() {
   abstract fun gridDataDao(): ImmutableGridDataDao
}
```

Obrázek 12 – Definice třídy databáze knihovny Room

Pomocí anotace `Database` zahrneme všechny objekty, které chceme v databázi uchovávat, nastavíme verzi a přidáme TypeConvertery. Pak už zde jen stačí nadefinovat abstract gettery pro všechna DAO, jejichž pomocí budeme získávat objekty z databáze a Room vygeneruje potřebný boilerplate kód. Zbývá tedy už jen získat referenci na tuto databázovou třídu, pomocí které získáme přístup k objektům pomocí DAO funkcí:

```
Room.databaseBuilder(
      applicationContext,
      AppDatabase::class.java, DB_NAME
).build()
```

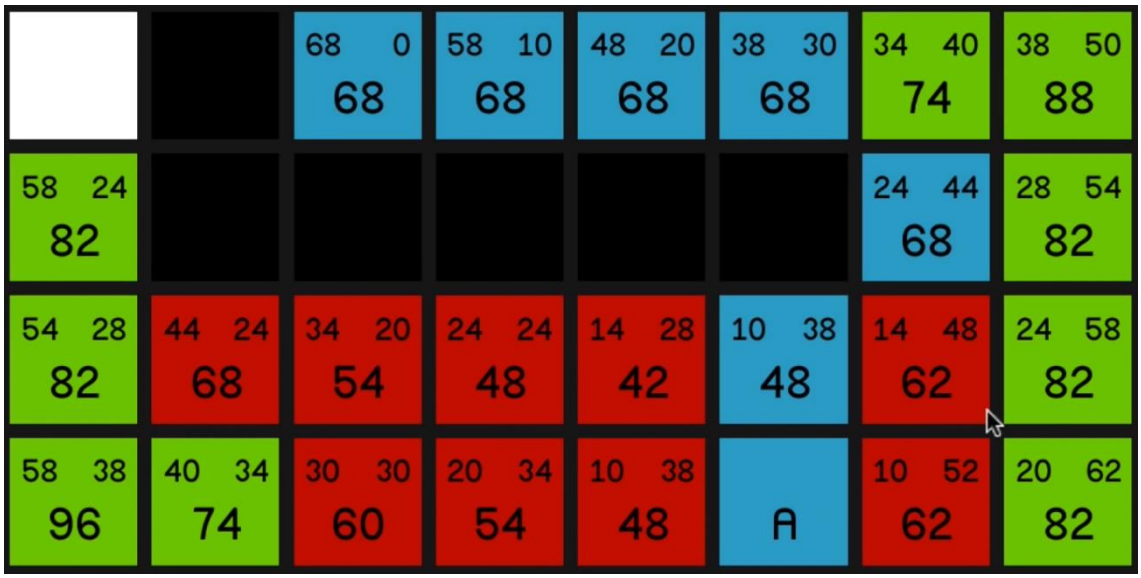
Obrázek 13 – Kód zprostředkovávajícího instanci databáze knihovny Room

2.6 A*

Algoritmus prohledávání grafu do šířky za účelem nalezení nejkratší cesty mezi dvěma body.

Struktura

Představme si graf jako libovolně velikou mřížku (grid), kde každé pole obsahuje nódu (node). Nódy reprezentují místo v prostoru. (Lague, 2014)



Obrázek 14 – Grafická reprezentace procházení grafu algoritmu A (Lague, 2014)*

Pole v mřížce může mít libovolné rozměry, ale musíme být schopni vždy určit vzdálenost mezi dvěma libovolnými nódy v mřížce.

A* je varianta algoritmu Dijkstra, která bere v potaz při prohledávání grafu také vzdálenost nódy od nódy koncové, čímž může být několikanásobně rychlejší, jelikož nemá tendenci prohledávat nódy zbytečně daleko od cíle. (Patel, nedatováno)

Nódy mají následující parametry:

- `gCost` – Vzdálenost od počáteční nódy (číslo v levém horním rohu nódy v obr. 8).
- `hCost` – Vzdálenost od konečné nódy (číslo v pravém horním rohu nódy).
- `fCost` – Součet `gCost` a `hCost`. Nejnižší `fCost` indikuje nejvhodnější nódu k prozkoumání v cyklu hledání algoritmu (číslo uprostřed nódy).
- `parent` – Nóda, ze které se algoritmus dostal na tuto nódu. Na jeho konci použijeme tyto nódy k rekonstrukci kompletní cesty.

Průběh algoritmu

Na následujícím popisu, podle kterého lze algoritmus A* implementovat, si též vysvětlíme jeho průběh:

1. Definujeme seznam nód `OPEN`. Bude obsahovat všechny nódy, které je třeba zpracovat.
2. Definujeme seznam nód `CLOSED`. Bude naplněn již zpracovanými nódy ze seznamu `OPEN` (v obr. 8 červené nódy)
3. Přidáme výchozí nódu do seznamu `OPEN`.
4. Vstupujeme do cyklu o předem nespecifikovaném počtu iterací.
5. Definujeme nódu `CURRENT` – vybereme ze seznamu `OPEN` tu s nejnižším parametrem `fCost`. Tuto nódu vymažeme ze seznamu `OPEN` a přidáme do seznamu `CLOSED`.
6. Pokud nóda `CURRENT` je shodná s tou, ke které jsme chtěli najít cestu, algoritmus přerušíme. Cesta byla nalezena.
7. Vstoupíme do iterátoru seznamu se všemi sousedními nódami kolem nódy `CURRENT` (bez nód, které již obsahuje seznam `CLOSED`).
8. Pokud nová vzdálenost po navštívení sousední nódy z nódy `CURRENT` je nižší než `gCost` sousední nódy, nebo sousední nóda není v seznamu `OPEN`:
    a. Nastavíme `gCost` sousední nódy na tuto novou vzdálenost.
    b. Spočteme `hCost` sousední nódy.
    c. Přiřadíme parametr parent sousední nódy na nódu `CURRENT`.
    d. Pokud sousední nóda není v seznamu `OPEN`, přidáme jí tam.
9. Průběh algoritmu se opakuje v cyklu od kroku 5 do doby, než bude platit podmínka v kroku 6.

Poté, co algoritmus proběhne, musíme provést retracing všech nód od té konečné pomocí iterace přes proměnnou `parent`. Tím dostaneme seznam nód, po kterých když půjdeme, dostaneme nejkratší cestu mezi dvěma zadanými místy směrem zpátky k počáteční nódě. Pokud tento seznam obrátíme, získáme požadovanou cestu skrz nódy v mřížce – v obr. 8 modré nódy. (Lague, 2014)

3 Návrh struktury aplikace

Aplikace pracuje s daty z následujících API endpointů:

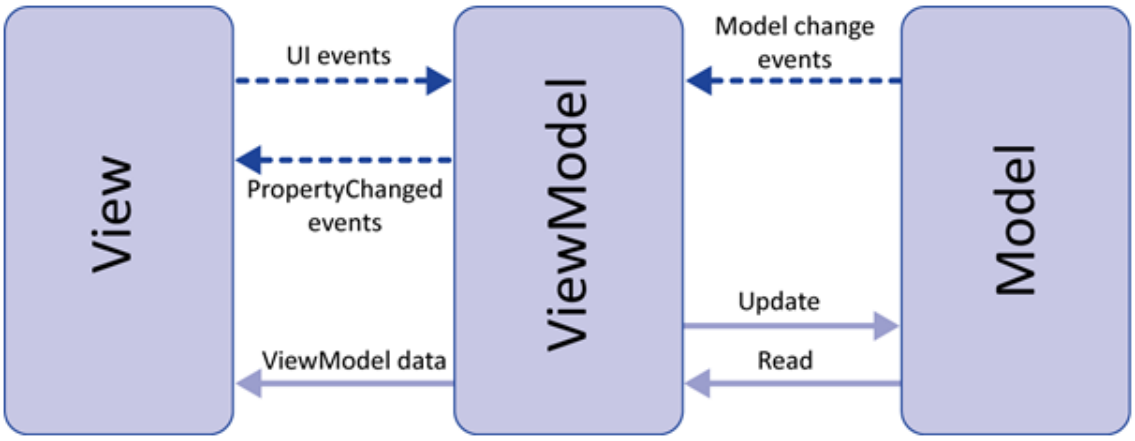
- `GET /client/getGridData`
  (navigační data pro A*, formát JSON)
- `GET /client/getSearchableData`
  (data pro vyhledávání a detail místa na mapě, formát JSON)
- `GET /client/getGridImage/{floorName}`
  (PNG obrázek patra s názvem `{floorName}`)

3.1 Model-View-ViewModel (MVVM)

Struktura MVVM se v Android aplikacích používá pro synchronizaci dat mezi pamětí a komponentou, která daná data zobrazuje. Pomocí této techniky zajistíme, aby se při získání nových dat správně aktualizovaly všechny související komponenty.

Skládá se ze tří objektů:

- **Model** – Datový objekt reprezentující data, která potřebujeme zobrazit.
- **View** – Komponenta, která tato data zobrazí.
- **ViewModel** – Pomocný objekt sloužící jako prostředník pro komunikaci mezi Modelem a View. Obsluhuje aktualizaci Modelu s daty a poskytuje observable objekty.



Obrázek 15 – Princip architektury MVVM (Saleh, 2017)

Observable objekt obaluje objekt s daty a poskytuje callback, který se vykoná v případě modifikace objektu. Pomocí ViewModelu, který tyto objekty z Modelu poskytuje, tedy tímto zajistíme, aby se při každé změně Modelu aktualizovala přiřazená komponenta v uživatelském rozhraní s novými daty. V případě implementace MVVM architektury neaktualizujeme komponenty v UI přímo, ale uvnitř triggeru, který se zavolá po změně dat. (Saleh, 2017)

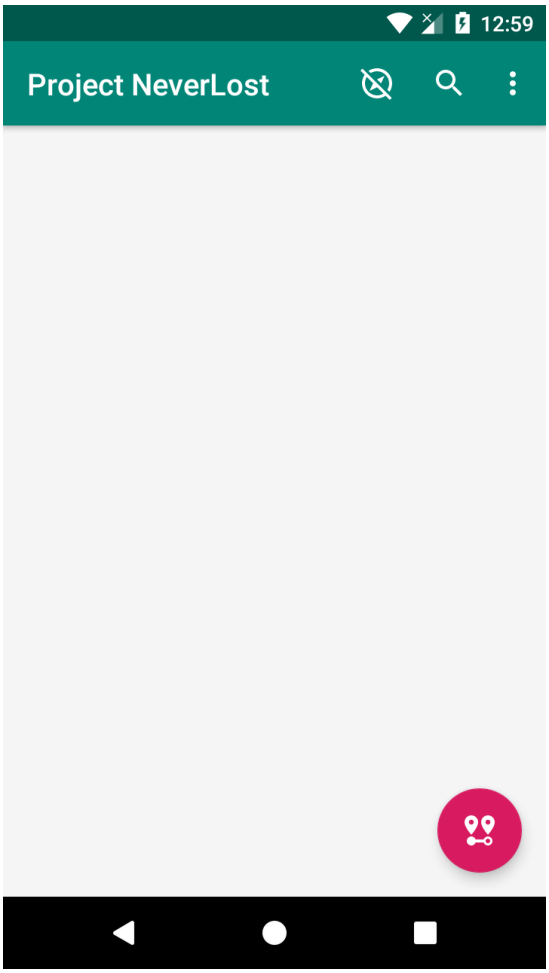
3.2 Repository pattern

Se strukturou MVVM je úzce spjaté repozitáře (Repository). Jsou to objekty, které zajišťují získávání dat z jednoho či více zdrojů (například web + caching pomocí lokální databáze). Pomáhají izolovat datovou a zobrazovací vrstvu aplikace tak, že data v Repository jsou přístupná komponentám uživatelského rozhraní až skrze ViewModel.

3.3 Uživatelské rozhraní

Rozhraní aplikace se skládá ze čtyř obrazovek (`Activity`) a jednoho vysunovacího listu (`BottomSheetBehavior`), který slouží pro vstup dat navigace a po vyhledání cesty jako itinerář.

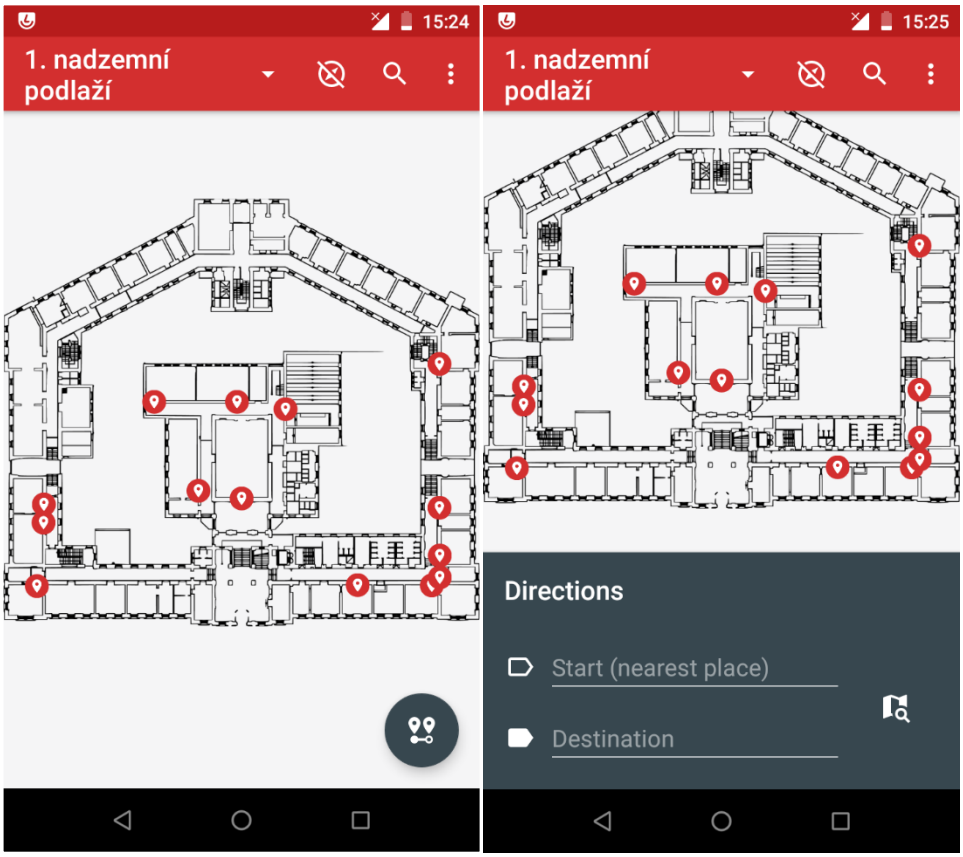
Barvy a ikony aplikace lze upravit, aby ji bylo možné využít nejen pro jednu specifickou budovu. V základním provedení se aplikace zobrazuje s výchozí barevnou kombinací, kterou Android Studio generuje pro nový projekt:



Obrázek 16 – Barevná kombinace unbranded flavoru aplikace

K tomu slouží flavory – varianty aplikací vycházející z hlavního codebase s menšími úpravami souborů použitými pro sestavení balíku aplikace. Momentálně jsou v projektu dva flavory – `branded` (barvy a konfigurace serveru pro navigační data budovy VŠPJ) a `unbranded` (výchozí barvy projektu původně připraveného Android Studiem, sloužící pro demonstraci alternativní verze aplikace s jinou grafikou a konfigurace serveru).

Hlavní obrazovka



Obrázek 17 – Hlavní obrazovka aplikace

Hlavní obrazovce dominuje mapa, která je tvořená třemi hlavními navrstvenými komponenty měnícími se s výběrem podlaží, které lze vybrat kliknutím na vyskakovací menu menu (`Spinner`) nahoře v červené liště (`Toolbar`):

- **`ImageView`** pro zobrazení plánu vybraného podlaží. Zobrazuje obrázek z `/client/getGridImage` endpointu.
- **`NavMeshView`** – vlastní komponenta zobrazující navigační data (cesty, segmenty trasy, v testovacích verzích nódy pro ladění dat, lze zapnout v nastavení). Zobrazuje data z `/client/getGridData` endpointu.
- Ikony interaktivních míst na mapě (`PlaceView`) jsou umístěny v kontejneru `PlaceViewContainer`, což je klasický `ViewGroup`. Rozmísťuje ikony po mapě, a umožňuje zvýraznit na mapě hledané místo.

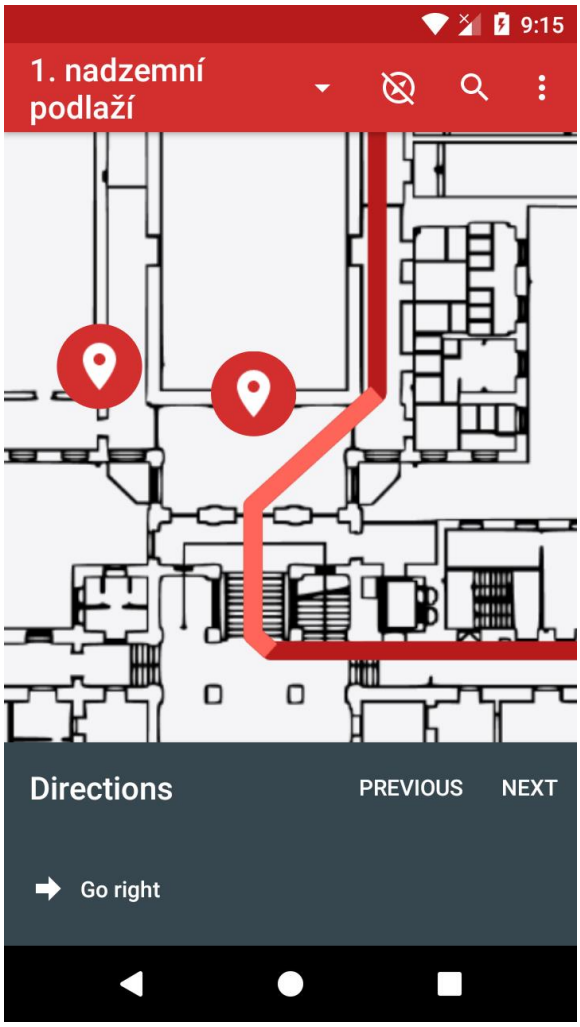
  Po kliknutí na ikonu místa se otevře obrazovka s detailnějšími informacemi o místě. Zobrazuje data z `/client/getGridData` endpointu.

Všechny tyto vrstvy jsou součástí `MapView`. Aby bylo možné s touto interaktivní mapou poskládanou z vrstev pohybovat a přibližovat, je `MapView` ve skutečnosti upravený `ZoomLayout` – komponenta z knihovny `com.otaliastudios:zoomlayout` zajišťující přibližující gesta pro jakékoli `View`, pro které je tato třída parentem. Jelikož mapou lze také otáčet podle kompasu zařízení (po přepnutí ikony s kompasem v horní části obrazovky), `MapView` se stal součástí komponenty `RotateLayout`. Podle názvu je patrné, že se jedná o layout, který umí otáčet s jemu příslušícími komponenty. Tady by se mohla logika otáčení z `RotateLayout` spojit s kódem `MapView` a zjednodušilo by se jedno vnoření layoutu v XML, ale v rámci zachování modularity jsem se rozhodl, že kód tříd zůstane v původním stavu.

BottomSheetView a itinerář

Součástí hlavní obrazovky je také vysouvací list (`BottomSheetView`) u dolního okraje, který lze, pokud je schovaný, zobrazit stiskem kulatého tlačítka v rohu obrazovky – FABu (Floating Action Button). Lze ho opět zavřít zasunutím ke spodnímu okraji obrazovky. Zobrazuje layouty pro dva stavy navigace:

- Vstupní pole pro data navigace – před vyhledáním trasy komponenta ukáže komponenty pro vstup dat potřebných pro navigaci (výchozí, koncový bod) a tlačítka pro potvrzení/výpočet trasy. Po kliknutí na jakékoli políčko pro zadávání místa se otevře obrazovka vyhledávání.
- Itinerář – po výpočtu trasy se layout vysouvacího listu změní na seznam připomínající itinerář trasy. Obsahuje základní orientační pokyny k pohybu po trase.

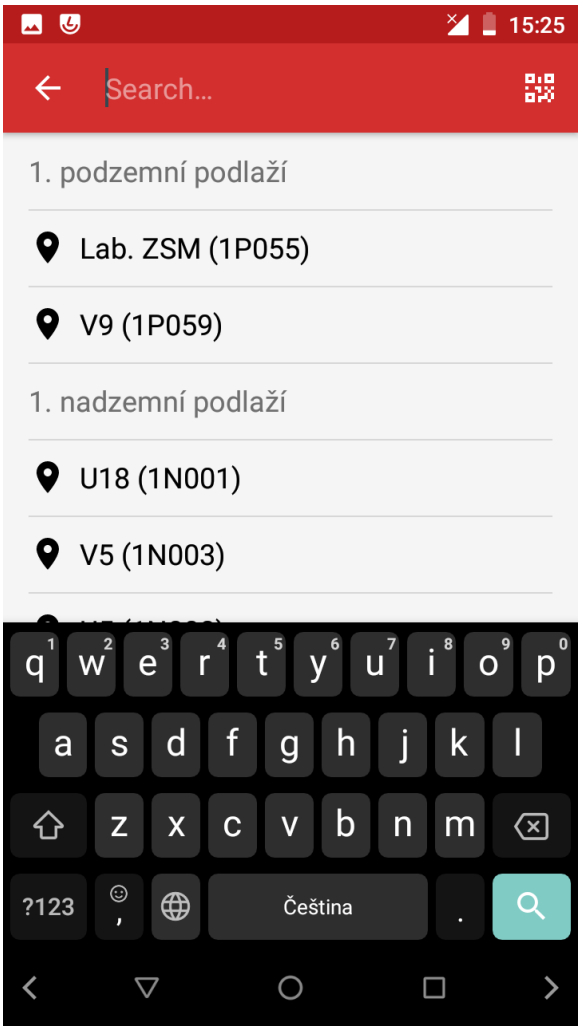


Obrázek 18 – Itinerář a segment trasy

V itineráři se zobrazují objekty `PathSegment`, které reprezentují, jak už jejich anglický název napovídá, segment cesty. Tlačitky vedle nadpisu itineráře lze mezi nimi přepínat, pokyny pro aktuální segment cesty jsou viditelné, i když není celý itinerář otevřený. Každý segment je tvořen čárou procházející maximálně čtyřmi body na mapě. Podle jejich vzájemné pozice v seznamu nód (výstupu algoritmu A*) je vypočítán směr, kterým se musíme vydat (doleva, doprava, po schodech nahoru, dolů).

Vyhledávání

Obrazovka umožňující uživateli vyhledávat místa na mapě. Zobrazuje data z `/client/getSearchableData` endpointu. Hlavní komponentou je seznam míst (`RecyclerView`) roztříděný podle podlaží, který můžeme filtrovat pomocí vyhledávacího políčka v `Toolbaru`.



Obrázek 19 – Vyhledávácí obrazovka

Lze ji otevřít z hlavní obrazovky pomocí ikony lupy v `Toolbaru` aplikace, kliknutím na zadávací políčka pro místa ve vysouvacím seznamu nebo skrz URL začínající následujícím řetězcem:

neverlost://

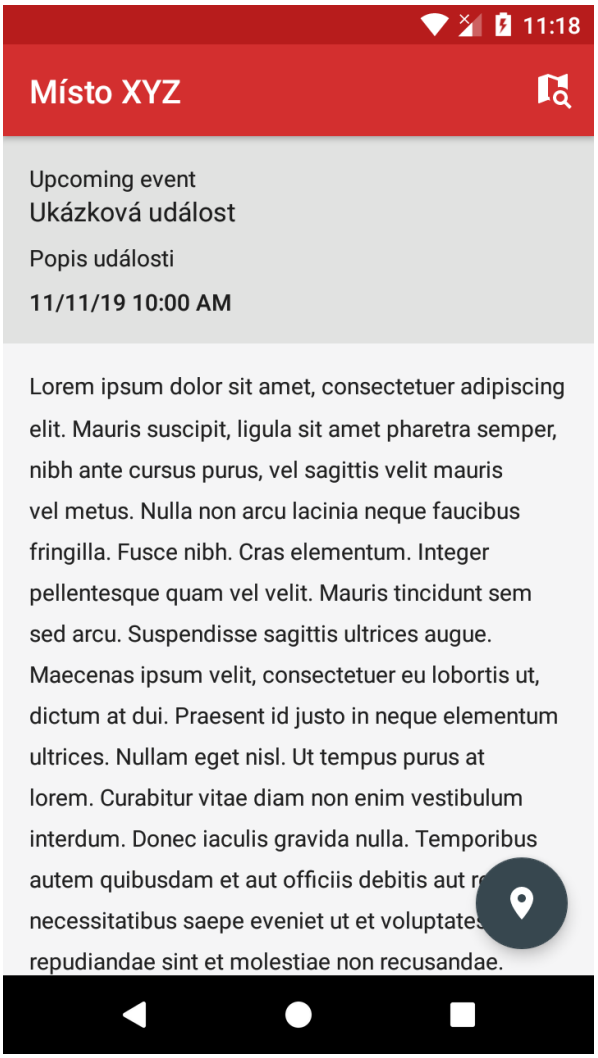
To proto, aby bylo možné vyhledávat místa pomocí QR kódů z aplikací třetích stran. Stačí oskenovat QR kód s tímto textem, který na konci obsahuje název místa, aby se otevřela vyhledávací obrazovka a název se předvyplní do vyhledávacího políčka. Pro případ, že uživatel nemá k dispozici aplikaci na čtení QR kódů, obsahuje aplikace vlastní, kterou otevře ikona QR kódu v `Toolbaru`.

Po kliknutí na místo v seznamu se otevře detailní obrazovka s daným místem (pokud vyhledávání bylo otevřeno skrz QR kód nebo ikonu lupy na hlavní obrazovce) nebo se

místo vyplní do příslušného políčka ve vysouvacím seznamu (pokud se vyhledávání otevřelo kliknutím na vstupní políčka navigace na hlavní obrazovce).

Detail místa na mapě

Zobrazí detailnější informace o místě a v případě, že se zde koná nějaká zvláštní událost, detaily o ní.



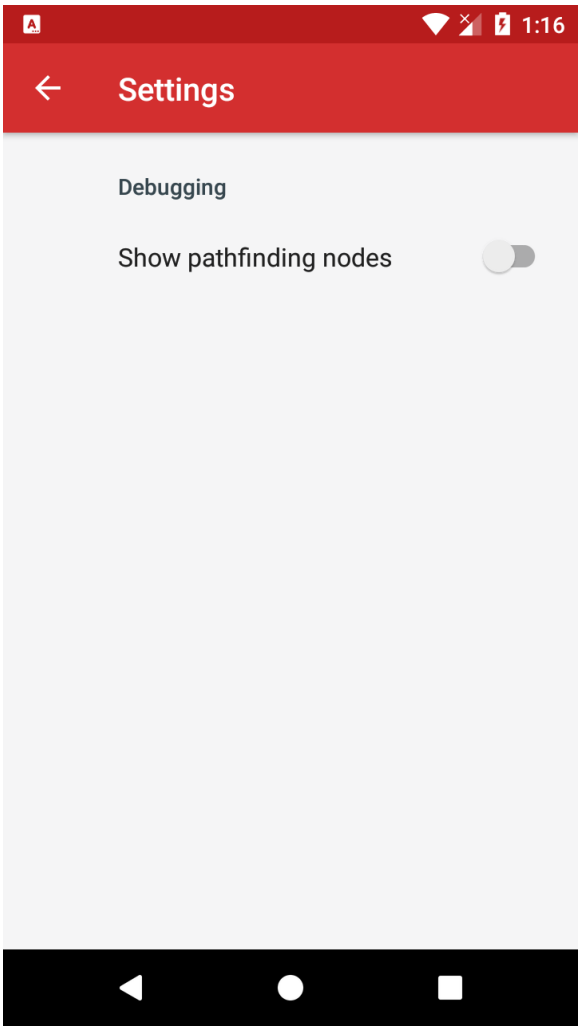
Obrázek 20 – Obrazovka s detailem místa s nadcházející událostí

Jsou zde k dispozici dvě tlačítka:

- Zvýraznění místa na mapě – pokud hledáme určité místo, po kliknutí na toto tlačítko se mapa na něj přiblíží a zvýrazní se jeho ikona.
- Nastavit jako koncový bod navigace – FAB, po jehož kliknutí se vyplní místo do políčka koncového bodu navigace ve vysouvacím seznamu na hlavní obrazovce.

Nastavení

V `debug` verzi je zde možnost graficky znázornit nódy grafu A* na mapě. Slouží k ladění dat, které přicházejí ze serveru. Layout pro tuto obrazovku se na rozdíl od jiných získává z XML souboru obsahující pouze speciální komponenty propojenými potřebnými parametry přímo v tomto souboru s třídou `SharedPreferences`, která umožňuje pracovat s jednoduchými datovými objekty uchovávanými v `/data` oddílu zařízení i po zavření aplikace.



Obrázek 21 – Obrazovka nastavení

4 Implementace navigace pomocí A*

Teď, když máme představu, jak vypadá rozhraní aplikace, můžeme se podívat na konkrétní implementaci algoritmu A*. V rámci organizace jsou všechny objekty pracující s algoritmem ve vlastní package s názvem „pathfinding“.

- **`GridData`** – datový objekt grafu algoritmu. Obsahuje data potřebná pro průběh algoritmu, funkce a hodnoty pro práci s nódami (hodnoty vzdáleností vypočítané podle velikosti mapy a počtu nód mřížky v každém směru).
- **`NavMesh`** – objekt zprostředkovávající samotný průběh algoritmu. Využívá dat a funkcí objektu `GridData`.
- **`Node`** – datový objekt reprezentující nódu v grafu, který algoritmus prochází.
- **`NodeHeap`** – implementace negenerické varianty datové struktury haldy (`Heap`) ke zrychlení prohledávání nódy s nejnižším ohodnocením (`fCost`).

Abychom získali cestu mezi dvěma místy na mapě, využijeme metody

```
fun getPath(
      startPlace: Place,
      targetPlace: Place,
      gridData: GridData
): List<Node>?
```

Obrázek 22 – Definice metody `NavMesh.getPath`

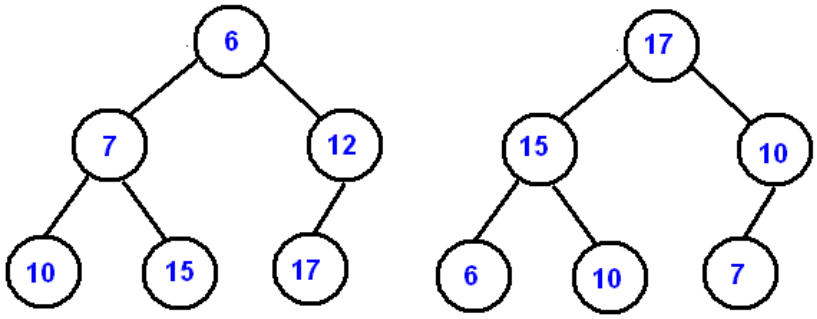
objektu `NavMesh`. Jako vstupní parametry je třeba poslat dva objekty reprezentující místo na mapě (`Place`) – počáteční a konečné místo. Z nich si funkce získá jejich parametr `nodeIndex` a objekt `GridData` pomocí tohoto indexu poskytne data o nódě. Pokud jakékoli místo neodpovídá nódě na mapě, algoritmus vyhodí výjimku, jelikož pravděpodobně přišla vadná data ze serveru. Po úspěšném dokončení algoritmu dostaneme seznam nód, v opačném případě dostaneme prázdnou referenci (v Kotlinu značenou otazníkem na konci deklarace typu – nullable type).

Jelikož aplikace by měla umět navigovat i napříč několika patry, bylo třeba upravit graf nód (mřížku) tak, aby byla definována pro každé patro zvlášť s propojujícími nódy reprezentující schody. Tím pádem muselo dojít k několika úpravám původního algoritmu:

- Funkce, která vrací seznam sousedících nód musí umět najít nejbližší schody pro oba směry (nahoru, dolů). Využívá se zde identifikace pozice nód pomocí objektu `Index3`, který, jak již název napovídá, obsahuje souřadnice indexu pro všechny tři směry. Stačí tedy zjistit, zda mřížka o patro nahoře či dole má na stejném místě nódu reprezentující schody a přidat ji do seznamu sousedících nód.
- Funkce výpočtu vzdálenosti mezi dvěma nódy (přímo, diagonálně) musí brát v potaz ještě vzdálenost diagonálně v horizontálním a vertikálním směru zároveň (tělesová úhlopříčka krychle o vzdálenostech rovných velikosti políčka pro nódy v mřížce).

4.1 Heap

Nejdéle v celém algoritmu trvá hledání nódy s nejnižší hodnotou `fCost` (součet vzdálenosti od počáteční nódy a vzdálenosti od nódy koncové). Tento úkon můžeme zoptimalizovat použitím vhodné datové struktury – haldy (`Heap`). Ta je tvořena objekty, kde každý objekt (parent, kořen) má pod sebou další dva objekty (child) a vypadá následovně:



Obrázek 23 – Struktura heapu: zleva MinHeap, MaxHeap (S.Adamchik, 2009)

Pokud jsme schopni vyjádřit stav objektu číslem, můžeme jej vložit do této struktury. Objekt s nejnižší hodnotou bude buď na začátku haldy (jedná se o `MinHeap`) nebo naopak s nejvyšším (`MaxHeap`). V tomto případě budeme vkládat nódu podle parametru `fCost` a vybírat objekty podle nejnižšího čísla. Jedná se tedy o strukturu `MinHeap`. Ta se řídí jedním jednoduchým pravidlem:

Každý objekt (parent) musí mít menší číselnou hodnotu než oba jeho následující (child) objekty.

Vložení objektu probíhá tak, že se vloží na konec heapu (na obrázku pozice vpravo dole), porovnají parent/child objekty patřící k danému objektu v heapu, pokud pravidla neplatí, objekty v konfliktu s pravidly si vymění pozice (prohodí se) a cyklus řazení pokračuje s novými parent/child objekty nově patřící k původně vloženému objektu. Při získávání objektu z heapu se vrátí kořenový objekt a nahradí se hodnotou z konce heapu, načež se opět provede řazení podle onoho jednoho jednoduchého pravidla. Tímto se zajistí prezence objektu s nejnižším číslem na začátku heapu. Podobně je to i pro MaxHeap, s rozdílem ve znaménku při porovnávání čisel objektů (menší než, větší než).

Rychlost řazení heapu spočívá v tom, že neporovnáváme objekty s celým obsahem heapu při jejich vložení či vyjmutí, ale pouze s jeho částí.

5 Propojení dat s uživatelským rozhraním

Nejdříve je potřeba propojit XML definice Views s kódem psaným v Kotlinu, abychom s komponenty uživatelského rozhraní vůbec mohli manipulovat. Jeden ze způsobů získání reference `View` definovaného v XML s daným ID je v Kotlinu následující:

```
private lateinit var loadingIndicator: ProgressBar

// uvnitř OnCreate()
loadingIndicator = findViewById(R.id.vLoadingIndicator)
```

Obrázek 24 – Ukázka získání reference View v XML z Kotlinu

Je to zbytečně dlouhé a dalo by se to napsat přehledněji. Proto jsem v aplikaci využil plugin `kotlin-android-extensions`, který generuje pro `Views` definované v XML proměnné toho typu, jaký je definovaný v XML (Kotlin Foundation, nedatováno). Tudíž pouze stačí přidat import (v případě, že se `View` nachází v layoutu s názvem `main_activity`):

```
import kotlinx.android.synthetic.main.main_activity.*
```

Obrázek 25 – Kotlin synthetic import

Pak k `View` přistupujeme přímo pomocí jeho ID tak, jak je definované v XML (case-sensitive), jako v případě jakéhokoli jiného objektu.

Existují diskuze o tom, zda je používání tohoto pluginu dobrý nápad, jelikož tímto způsobem může dojít k pádům aplikace kvůli referenci neexistujících komponent v XML souboru layoutu. To může nastat, pokud je definovaný podobný layout pro jinou konfiguraci (veliost displeje), ve kterém daná komponenta chybí. K pádům také dojde, pokud bychom omylem zaměnili import z jiné aktivity a snažili se manipulovat s komponentou, která má v obou layoutech stejné ID. Po zvážení všech pro a proti jsem se stejně rozhodl plugin použít, jelikož chybný import by stejně byla chyba programátora, ne pluginu a scénář s chybějícími komponenty v jiných konfiguracích zde nenastane.

5.1 Repository pattern a MVVM

Aplikace využívá struktury MVVM – `Views` reagují na změnu dat vlastností ve ViewModelu, ten komunikuje s Repository, které se stará o získávání dat ze serveru (Retrofit) a používá caching databázi ke které přistupuje pomocí DAO (Room).

Nyní si ukážeme průběh načítání dat pro A* (`GridData`) napříč všemi spolu souvisejícími objekty.

V XML je definované `MapView`, které bude data zobrazovat. Potřebujeme ViewModel, který bude poskytovat `LiveData<GridData>`, což je obalovací objekt (wrapper), který umožňuje ostatním reagovat na změny obalovaného objektu následujícím způsobem:

```
// Definujeme ViewModel
val viewModel: MainActivityViewModel by lazy {
   ViewModelProvider(this)[MainActivityViewModel::class.java]
}

// V OnCreate
viewModel.gridData.observe(this, Observer { gridData ->
    // Reagujeme na změny objektu GridData
})
```

Obrázek 26 – Ukázka reakce na změny objektu pomocí `LiveData` wrapperu

Co se UI týče, stačí poslat nová data do `MapView` a dalších `Views` (`Spinner` pro výběr podlaží). ViewModel slouží jako prostředník pro komunikaci s Repository a obsahuje proměnnou `gridData`, která je pouze odkazem na původní objekt, který Repository načítá:

```
private val gridDataRepository = GridDataRepository(application)
val gridData = gridDataRepository.gridData
```

Obrázek 27 – Propojení s Repository ve ViewModelu

ViewModel slouží pouze jako propojovací vrstva oddělující třídy UI a ty, co datové objekty modifikují. Z ViewModelu se dále dostáváme k samotnému načítání objektu, definováno v Repository:

```
private val immutableGridData = Transformations.switchMap(revision) {
   object : NetworkBoundResource<ImmutableGridData, ImmutableGridData>() {
      override fun saveCallResult(item: ImmutableGridData) =
            gridDataDao.update(item)

      override fun shouldForceFetch(data: ImmutableGridData?) =
            preferences.revisionCheckIntervalExceeded()

      override fun loadFromDb() = gridDataDao.loadGridData()
      override fun createCall() = webService.getGridData()
   }.asLiveData()
}
val gridData: LiveData<GridData?> = Transformations.map(immutableGridData) {
    immutableGridData -> immutableGridData.data?.toMutableGridData()
}
```

Obrázek 28 – Načítání objektu `GridData` v Repository

V aplikaci jsou dva objekty `GridData`. Jeden obsahuje pouze parametry získávané ze serveru/ukládané do databáze (`ImmutableGridData`) a druhý se vytváří na základě těchto dat, s jeho daty je možné manipulovat a obsahuje metody a proměnné potřebné k průběhu A* algoritmu (`GridData`). Nacházejí se také v odlišných balíkách (package). Nejprve se získají `ImmutableGridData` ze serveru nebo databáze. Databáze slouží k uložení dat pro možnost provozu offline, pokud nejsou tyto data k dispozici lokálně, Retrofit se postará o získání objektu ze serveru. O koordinaci cache a volání API se stará `NetworkBoundResource`. Tato třída je k dispozici pro inspiraci v oficiálních repozitářích Google spolu s dalšími potřebnými pomocnými třídami. (Google, nedatováno)

Pomocí `Transformations` třídy při definici objektu můžeme zajistit, aby delkarovaný objekt reagoval na změny parametru funkce `switchMap` nebo map a sám se podle toho upravil (v tomto případě se spustí fetch dat) provedením deklarovaného delegátu funkce. Objekt zobrazený v UI se vytvoří podle objektu, jaký se podařilo načíst buď z cache nebo serveru. Stejným způsobem je získávání dat závislé na revizi dat na serveru. Pokud se změnila revize, nebo uplynul interval aktualizace dat, je třeba načíst nová data ze serveru bez ohledu na stav cache.

Načítání dat probíhá při otevření aplikace nebo pokaždé, když se zařízení připojí k internetu a předchozí komunikace se serverem nebyla úspěšná. Díky tomu aplikace nepotřebuje v UI komponenty pro aktualizaci dat, jelikož reaguje na změnu stavu konektivity sama za pomoci třídy `ConnectionStateMonitor`, která funguje jako wrapper pro `NetworkCallback`:

```
class ConnectionStateMonitor(context: Context,
                             private val listener: ConnectionStateListener
) : NetworkCallback() {
    ...
    fun register() = connectivityManager.registerNetworkCallback(
        networkRequest,
        this)
    fun unregister() = connectivityManager.unregisterNetworkCallback(this)
    ...
}

// K dispozici je v ConnectionStateListener interface metoda
fun onNetworkAvailabilityChanged(available: Boolean)
```

Obrázek 29 – Ukázka implementace `ConnectionStateMonitor`

Každá Aktivita, která obsahuje prvky reagující na stav načítání dat z Repository, vychází z `LoadingAwareActivity`, která se stará o změny UI v závislosti na aktuálním stavu

načítání dat (ukáže/skryje `ProgressBar`, zobrazí/skryje obsah). Při změně stavu konektivity volá metodu pro aktualizaci dat ve svých ViewModelech – všechny ViewModely nacházející se v implementacích této `Activity` vycházejí z třídy `LoadingAwareViewModel` a poskytují funkci pro vynucení načítání dat.

5.2 Seznamy, ViewHolder pattern

Komponenty spravující seznam položek (`RecyclerView`, `ListView` a další) ideálně pracují na principu poolingu (recyklace již inicializovaných `Views`). Aby se seznam posouval plynule i při větším počtu položek, nenačítají se všechny položky naráz, ale pouze ty, které jsou vidět na obrazovce. Pokud se seznam posune například dolů, položka na začátku seznamu se přesune na konec. Přesun je mnohem rychlejší, než tvořit nové instance a odstraňovat ty staré, ale je potřeba zajistit, aby obsah přesunutého `View` odpovídal datům objektu na nové pozici. V aplikaci je pro zobrazení položek v seznamech pro vyhledávací aktivitu a itinerář použit `ViewHolder`, který spolupracuje s `RecyclerView` a umožňuje pooling. Abychom mohli zobrazit vlastní objekty v seznamu, musíme nejdříve vytvořit adaptér, odkud `RecyclerView` získává data:

```
class DirectionsAdapter : RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    private var data = mutableListOf<PathSegment>()
    private lateinit var onItemClickCallback: (Int, PathSegment) -> Unit
    ...
    override fun getItemCount() = data.size

    fun updateDataSet(data: List<PathSegment>?) {
       this.data.clear()
       if (data != null) {
          this.data.addAll(data)
       }
       notifyDataSetChanged()
    }
    ...
}
```

Obrázek 30 – Ukázka implementace adaptéru pro itinerář

V momentě, kdy adaptéru předáme data k zobrazení pomocí metody `updateDataSet`, naplníme vlastní seznam s objekty, které chceme zobrazovat. V tomto momentě tento seznam nemá ještě žádný vliv na zobrazovaná data. Jakmile zavoláme metodu `notifyDataSetChanged`, `RecyclerView` podle metody `getItemCount` zjistí počet položek, které je třeba zobrazit a teprve zde začínáme pracovat s naším seznamem `data`. Seznam bude mít takový počet položek, kolik je objektů v tomto seznamu. Následuje

vytvoření samotných `Views` pro položky seznamu. K tomu slouží `ViewHolder`. V adaptéru nadefinujeme následující třídu:

```
internal inner class DirectionsViewHolder(v: View) : RecyclerView.ViewHolder(v) {
   private val icon: ImageView = v.findViewById(R.id.vDirectionsItemIcon)
   private val text: TextView = v.findViewById(R.id.vDirectionsItemText)

   private val item get() = data[adapterPosition]

   init {
      v.setOnClickListener {
         onItemClickCallback(
               adapterPosition,
               item)
      }
   }

   fun update() {
      val viewContext = text.context
      text.text = item.getText(viewContext.resources)
      icon.setImageResource(item.iconResId)
   }
}
```

Obrázek 31 – Třída `ViewHolderu` v seznamu pro itinerář

Tento `ViewHolder` obsahuje reference na komponenty položky seznamu, která má reprezentovat objekt. O recyklaci položky se nemusíme starat, je pouze potřeba doplnit metody pro práci s `ViewHolderem`:

```
override fun onCreateViewHolder(
      parent: ViewGroup,
      viewType: Int
): RecyclerView.ViewHolder =
DirectionsViewHolder(LayoutInflater.from(parent.context)
      .inflate(R.layout.directions_item, parent, false))

override fun onBindViewHolder(
      holder: RecyclerView.ViewHolder,
      position: Int
) = (holder as DirectionsViewHolder).update()
```

Obrázek 32 – Metody pro obsluhu `ViewHolderu`

Tyto metody zajistí vytvoření instance `ViewHolderu` a aktualizaci dat při přesunu položek v seznamu (volání funkce `update`). Seznam bude funkční, jakmile tento adaptér přiřadíme `RecyclerView` a naplníme ho daty:

```
directionsAdapter = DirectionsAdapter()
directionsAdapter.setOnItemClickCallback { ... }
...
with(vDirectionsSheetList) {
    ... // LayoutManager, decorator,...
    adapter = directionsAdapter
}

// Změna dat v seznamu
directionsAdapter.updateDataSet(pathSegments)
```

Obrázek 33 – Ukázka použití `RecyclerView` adaptéru

5.3 Předávání dat mezi obrazovkami

Aby bylo možné vybírat počáteční/koncové místo na mapě, bylo třeba postarat se o předání objektu Place mezi aktivitami pomocí objektu `Intent`. Do něj lze vložit a poslat mezi aktivitami několik základních datových typů, mezi kterými je i `Parcelable`. Je to speciální datový typ používaný v Androidu pro rychlou serializaci a deserializaci dat přesně k tomuto účelu. Abychom mohli serializovat objekt do `Parcelable`, v Kotlinu jeho třídu stačí pouze označit anotací `Parcelize`, jelikož v projektu jsou povoleny experimentální Kotlin Android extensions. To samé je třeba udělat i u všech dalších vnořených tříd. Poté můžeme (nejen) objekty typu `Place` poslat pomocí metody `Intent.putExtra`.

Komunikace mezi aktivitami probíhá tak, že z jedné aktivity otevřeme druhou pomocí `startActivityForResult`. Poté, co se tato nová aktivita ukončí, zpracujeme výsledek v metodě `onActivityResult`, kde dostaneme právě ten `Intent`, ve kterém se nachází potřebný objekt `Place`. Ve skutečnosti je to ale komplikovanější. Tento základní postup by stačil na předávání dat mezi dvěma aktivitami. Jelikož můžeme otevřít vyhledávání (dokonce přímo, bez předchozího otevření hlavní aktivity – například z externí QR čtečky), poté detail jakéhokoli místa a teprve odtud se můžeme rozhodnout, co s místem uděláme, je potřeba tento postup upravit.

Aktivitu pro vyhledávání můžeme otevřít proto, že potřebujeme vyhledat výchozí nebo koncové místo – v tomto případě se aktivita vyhledávání po kliknutí na místo zavře a pošle vybrané místo zpět do aktivity hlavní obrazovky s vysouvacím seznamem, kde se místo vyplní do příslušného políčka. Pokud vyhledávání otevřeme pomocí ikony lupy, vrátí se pomocí `Intentu` do aktivity hlavní obrazovky vybraný objekt `Place`.

```
override fun onPlaceSelected(place: Place) {
   if (intent.getIntExtra(ARG_REQUEST_TYPE, PICK_NONE_JUST_SEARCH) ==
PICK_NONE_JUST_SEARCH) {
      PlaceDetailActivity.startForResult(this, place.id)
   } else {
      val data = Intent()
      data.putExtra(MainActivity.ARG_PLACE_EXTRA, place)
      returnToMainActivity(data)
   }
}
```

Obrázek 34 – Funkce reagující na výběr místa ve vyhledávání

V metodě `onPlaceSelected` bylo potřeba vyřešit další problémy. Pokud by se po kliknutí na místo otevřela detailní obrazovka a na ní bychom se rozhodli, že chceme dané místo vyplnit jako koncové, po ukončení detailní aktivity bychom nevěděli, co kam vyplnit. Musíme tedy spustit detailní aktivitu tak, jako bychom od ní vždy čekali nějaký výsledek (`startActivityForResult`). I kdyby ji uživatel jednoduše zavřel tlačítkem zpět. Poté můžeme v `onActivityResult` reagovat na výsledek, na rozdíl od klasické `startActivity` metody. Součástí této metody je také kód pro zpracování názvu místa z QR kódu v případě, že se do vyhledávání vracíme z QR čtečky vestavěné v aplikaci.

Callbacky `onPlaceSelected` a `onActivityResult` spouštějí stejnou funkci – `returnToMainActivity`. Ta na základě toho, zda byla aktivita spuštěna pomocí `Intentu` z externí QR čtečky nebo ne, otevře či vrátí se do aktivity hlavní obrazovky a pošle data o místě a informace o tom, co je třeba s místem provést.

Nejobsáhlejší ze všech metod zajišťující komunikaci mezi aktivitami je `handleResultIntent` v aktivitě hlavní obrazovky. Její struktura je následující:

```
private fun handleResultIntent(data: Intent?, requestCode: Int) {
   if (data == null) return
   val place: Place? = data.getParcelableExtra(ARG_PLACE_EXTRA)

   when (requestCode) {
      PlaceSearchActivity.PICK_NEAREST -> {
         viewModel.setNearestPlace(place)
      }
      PlaceSearchActivity.PICK_DESTINATION -> { ... }
      else -> {
         val markActionExtra = data.getStringExtra(...)
         if (markActionExtra == null) {
            return
         }

         when (markActionExtra) {
            PlaceDetailActivity.MARK_AS_HIGHLIGHTED -> {
               viewModel.setSearchedPlace(place)
            }
            PlaceDetailActivity.MARK_AS_NEAREST -> { ... }
            PlaceDetailActivity.MARK_AS_DESTINATION -> { ... }
         }
      }
   }
}
```

Obrázek 35 – Zpracování dat z `Intentu` v aktivitě hlavní obrazovky

Zde se zpracují všechna data z postupně otevíraných aktivit a za pomoci ViewModelu se místa zadají tam, kam je třeba podle parametrů `requestCode` nebo `markActionExtra` posílaných mezi aktivitami.

6 Ověření funkčnosti, testování

Aplikaci s úpravami popsanými v této bakalářské práci jsem aktivně spolu s kolegou Danielem Matějkou, který vytvořil pro aplikaci API poskytující navigační data, využíval během vývoje v průběhu semestru. Během této doby se vyskytl v algoritmu pouze jeden zásadní problém narušující funkčnost aplikace, a to generování nesmyslně dlouhých tras zbytečně vedoucích po schodech do jiných pater, než je nezbytně potřeba.

Vzhledem k tomu, že jsem kód A* zkoušel před touto navigační aplikací i na několika projektech v Unity a algoritmus prošel i přepisem do C++ a pokaždé běžel bez problémů, jediné možné slabé místo mohlo být pouze u procházení několika grafů spojených schody (speciálními nódami). Po revizi tohoto kódu jsem narazil na chybu v metodě pro výpočet vzdáleností mezi nódami, kde bylo potřeba počítat se vzdálenosti ve třetím směru (vertikálně). Poté, co tato metoda byla o tento výpočet obohacena, nejevila aplikace další známky chyb navigace.

Pro případ, že by ze serveru přišla nevalidní data a aplikace by nebyla schopna najít cestu mezi místy na mapě, by ideálně měly být na serveru testy hned po úpravě navigačních dat. Tudíž pravděpodobnost, že tento stav nastane, je v produkčním prostředí díky těmto opatřením téměř nulová.

Závěr

Aplikace je použitelná v různých budovách díky individuální konfiguraci navigačních dat. Díky použitým knihovnám neobsahuje projekt příliš boilerplate kódu a díky funkcím jazyka Kotlin jsem spokojen s celkovým průběhem vývoje. Problémy nastaly až ve chvíli, kdy bylo potřeba upravit algoritmus tak, aby správně vyhodnotil cestu mezi patry. V prvních implementacích těchto změn byly generovány zbytečně dlouhé a nesmyslné trasy, což bylo způsobeno špatnou kalkulací vzdáleností mezi nódami ve třetí dimenzi. Nebylo nutné zasahovat do průběhu samotného algoritmu, stačilo upravit samotné výpočty.

Mnohem více času, než samotnému pathfindingu jsem věnoval struktuře aplikace. Původní Java aplikaci jsem přepsal do jazyka Kotlin, což v některých případech značně zjednodušilo či zkrátilo kód. Zakomponování struktury MVVM a Repository pattern do aplikace psané v Kotlinu nezabralo příliš času.

Do budoucna je tu možnost přidat další endpoint pro získání detailu o místě. Momentálně tato data aplikace získává z dat pro vyhledávání, což je výhodnější z hlediska ukládání dat o všech místech do cache naráz, ale mohly by tady nastat problémy během zpracovávání většího JSON souboru na některých zařízeních. Díky modulární architektuře aplikace je tu také do budoucna prostor například pro rozšíření o dependency injection a testy na fiktivních datech (mock repository, mock webservice), ale to už je téma pro jiné bakalářské práce.

Seznam použité literatury

BRESLAV, A., 2016. *Kotlin 1.0 Released: Pragmatic Language for JVM and Android | Kotlin Blog.* [Online]
Available at: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/
[Přístup získán 11 30 2019].

CODE AURORA FORUM, nedatováno *Code Aurora git repositories.* [Online]
Available at: https://source.codeaurora.org/quic/la/kernel
[Přístup získán 1 11 2020].

GOOGLE, nedatováno *Android Architecture Components | Android Developers.* [Online]
Available at: https://developer.android.com/topic/libraries/architecture
[Přístup získán 1 4 2020].

GOOGLE, nedatováno *google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back.* [Online]
Available at: github.com/google/gson
[Přístup získán 29 11 2019].

GOOGLE, nedatováno *Material Design.* [Online]
Available at: https://www.material.io/
[Přístup získán 27 11 2019].

GOOGLE, nedatováno *Save data in a local database using Room | Android Developers.* [Online]
Available at: https://developer.android.com/training/data-storage/room
[Přístup získán 30 3 2020].

INTERNET ENGINEERING TASK FORCE (IETF), nedatováno *JSON.* [Online]
Available at: json.org/
[Přístup získán 29 11 2019].

JETBRAINS, nedatováno *Kotlin Programming Language.* [Online]
Available at: kotlinlang.org
[Přístup získán 29 11 2019].

KOTLIN FOUNDATION, nedatováno *Kotlin Android Extensions - Kotlin Programming Language.* [Online]
Available at: https://kotlinlang.org/docs/tutorials/android-plugin.html
[Přístup získán 1 4 2020].

LAGUE, S., 2014. *(65) A* Pathfinding (E01: algorithm explanation) - YouTube.* [Online]
Available at: youtu.be/-L-WgKMFuhE
[Přístup získán 29 11 2019].

PATEL, A., nedatováno *Introduction to A*.* [Online]
Available at: theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
[Přístup získán 29 11 2019].

ADAMCHIK, V., 2009. *Heaps.* [Online]
Available at: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html
[Přístup získán 30 3 2020].

SALEH, H., 2017. *MVVM architecture, ViewModel and LiveData (Part 1) - ProAndroidDev.* [Online]
Available at: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1
[Přístup získán 27 11 2019].

SQLITE CONSORTIUM, nedatováno *About SQLite.* [Online]
Available at: sqlite.org/about.html
[Přístup získán 29 11 2019].

SQLITE CONSORTIUM, nedatováno *The SQLite Amalgamation.* [Online]
Available at: sqlite.org/amalgamation.html
[Přístup získán 29 11 2019].

SQUARE, nedatováno *Retrofit.* [Online]
Available at: https://square.github.io/retrofit/
[Přístup získán 24 3 2020].

TURI, G., nedatováno *Online JSON Viewer.* [Online]
Available at: jsonviewer.stack.hu
[Přístup získán 30 11 2019].