TECHNICKÁ UNIVERZITA V LIBERCI
Ekonomická fakulta

Vývoj moderní Android aplikace s využitím Jetpack Compose

Diplomová práce

Studijní program: N0688A140016 Systémové inženýrství a informatika
Autor práce: **Bc. Petr Horáček**
Vedoucí práce: Mgr. Tomáš Žižka, Ph.D.
Katedra informatiky

Liberec 2022

TECHNICKÁ UNIVERZITA V LIBERCI
Ekonomická fakulta

Zadání diplomové práce

Vývoj moderní Android aplikace s využitím Jetpack Compose

Jméno a příjmení: **Bc. Petr Horáček**
Osobní číslo: E20000502
Studijní program: N0688A140016 Systémové inženýrství a informatika
Zadávající katedra: Katedra informatiky
Akademický rok: **2021/2022**

Zásady pro vypracování:

1. Uvedení sady nástrojů Jetpack Compose
2. Vymezení moderní Android architektury
3. Návrh, vývoj a implementace aplikace
4. Uživatelské testování a možnosti rozšíření aplikace
5. Zhodnocení navržených řešení

Rozsah grafických prací:
Rozsah pracovní zprávy: 65 normostran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

- LEIVA, Antonio, 2016. *Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App*. CreateSpace Independent Publishing Platform. ISBN 978-1530075614.
- CASTILLO, Jorge a Andrei SHIKOV, 2021. *Jetpack Compose internals* [online]. Leanpub. [cit. 2021-9-21]. Dostupné z: https://jorgecastillo.dev/book/
- WENDERLICH, Ray, Tino BALINT a Denis BUKETA, 2021. *Jetpack Compose by Tutorials (First Edition): Building Beautiful UI With Jetpack Compose*. Razerware. ISBN 978-1950325122.
- WENDERLICH, Ray, Yun CHENG a Aldo O. DOMÍNGUEZ, 2019. *Advanced Android App Architecture (First Edition): Real-world app architecture in Kotlin 1.3*. Razerware. ISBN 978-1942878698.
- NIELSEN, Mike Møller, 2020. *Hands-On Kotlin Web Development with Ktor*. Packt Publishing. ISBN 9781838640767.
- PROQUEST, 2021. *Databáze článků ProQuest* [online]. Ann Arbor, MI, USA: ProQuest. [cit. 2021-09-26]. Dostupné z: http://knihovna.tul.cz

Konzultant: RNDr. Jiří Hrdina CSc. - programátor, Devro s.r.o

Vedoucí práce: Mgr. Tomáš Žižka, Ph.D.
Katedra informatiky

Datum zadání práce: 1. listopadu 2021
Předpokládaný termín odevzdání: 31. srpna 2023

L.S.

doc. Ing. Aleš Kocourek, Ph.D.
děkan

Ing. Petr Weinlich, Ph.D.
vedoucí katedry

V Liberci dne 1. listopadu 2021

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci 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.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

28. dubna 2022 Bc. Petr Horáček

Anotace

Tato diplomová práce se věnuje tématu vývoje Android aplikace za použití Android UI frameworku Jetpack Compose. Teoretická část práce je věnována představení frameworku Jetpack Compose, srovnání vývoje UI pomocí Jetpack Compose s předchozím způsobem vývoje a uvedení moderní architektury aplikací pro operační systém Android.

Praktická část práce je poté primárně věnována hlavnímu předmětu práce, kterým je vývoj Android aplikace s pomocí Jetpack Compose, včetně vybudování aplikačního backendu. Praktická část se dále také věnuje uživatelskému testovaní a celkovému zhodnocení vývoje.

Výsledkem diplomové práce je mobilní Android aplikace, jejíž UI je plně vybudováno pomocí frameworku Jetpack Compose a která slouží jako sociální síť pro studenty ubytované na kolejích Harcov. Diplomová práce může dále sloužit jako návod pro vývoj moderní Android aplikace podle principů Android aplikační architektury.

Klíčová slova: Jetpack Compose, composable funkce, Android architektura, Android Jetpack, injektáž závislostí, stránkování, MVVM, KTor, Heroku, MongoDB, Rest API, XML, Figma

Annotation

This thesis is about Android application development using the Android UI framework Jetpack Compose. The theoretical part of the thesis is devoted to introducing the Jetpack Compose framework, comparing UI development using Jetpack Compose with previous development methods, and introducing modern Android application architecture.

The practical part of the thesis is then primarily devoted to the main subject of the thesis, which is Android application development using Jetpack Compose, including building the application backend. Furthermore, the practical part also focuses on user testing and overall development evaluation.

The result of the thesis is a mobile Android application whose UI is fully built using the Jetpack Compose framework and which serves as a social network for students staying at the Harcov dormitory.. Furthermore, the thesis can serve as a guide for the development of a modern Android application according to the principles of Android application architecture.

Keywords: Jetpack Compose, composable functions, Android architecture, Android Jetpack, dependency injection, paging, MVVM, KTor, Heroku, MongoDB, Rest API, XML, Figma

Poděkování

Touto krátkou pasáží bych chtěl poděkovat všem, kteří se přímo či nepřímo podíleli na vzniku této práce. Velice děkuji vedoucímu práce Mgr. Tomáši Žižkovi Ph.D. za poskytování cenných rad a konstruktivních poznámek, díky kterým bylo možné dovést tuto práci do konce. Dále děkuji všem internetovým tvůrcům obsahu o vývoji Android aplikací, jejichž poskytnuté informace velkou mírou pomohli k vývoji finální aplikace, která je součástí této diplomové práce. Velké díky také patří Bc. Marii Opočenské za korekturu pravopisných a stylistických chyb. V neposlední řadě děkuji všem účastníkům testování aplikace, bez nichž by nebylo dokončení práce možné.

Dále děkuji celému mému okolí, které mě provázalo celým studiem na vysoké škole. Velký dík patří mé rodině a rodičům, bez jejichž finanční a mentální podpory by mé studium na vysoké škole nebylo možné. Díky patří také všem mým spolubydlícím na harcovské buňce E312, kteří mě všemi roky vysokoškolského studia provázeli. Nakonec děkuji také své přítelkyni za podporu ve chvílích, kdy se některé problémy zdály neřešitelné.

Obsah

Seznam obrázků

Seznam ukázek kódů

Seznam tabulek

Seznam použitých zkratek

API	Application Programming Interface
APK	Android Application Package
DBaaS	Database as a Service
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JC	Jetpack Compose
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JWT	Json Web Token
MVVM	Model View ViewModel
OHA	Open Handset Alliance
OS	Operační systém
PaaS	Platform as a Service
PNG	Portable Network Graphics
REST	Representational State Transfer
RGB	Red Green Blue
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SoC	Separation of Concerns
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VPC	Virtual Private Cloud
XML	Extensible Markup Language

Úvod

V poslední dekádě došlo na trhu s mobilními telefony k enormním změnám. Chytré, dotykové, mobilní telefony, které v dnešní době používá většina obyvatelstva, se staly běžným nástrojem každodenního života a je čím dál náročnější narazit na odlišný typ mobilního telefonu. Podobná situace se odehrává také na poli operačních systémů pro mobilní telefony, kde většinu tržního podílu tvoří dva operační systémy: Android a iOS.

Vzhledem k vysoké poptávce po chytrých telefonech a velkému množství producentů takových telefonů je poté pochopitelné, že se jednotliví výrobci snaží oproti ostatním výrobcům vyniknout, a to sice např. snižováním cen, ale také zdokonalováním technických parametrů nových modelů telefonů. A stejně, jako dochází ke snahám o hardwarová zdokonalování chytrých telefonů, vývoj a zdokonalování podstupují také prostředky k vytváření aplikací pro zmíněné operační systémy. A právě jeden z takových prostředků je hlavním tématem této diplomové práce.

Tato diplomová práce se primárně věnuje vývoji moderní Android aplikace s využitím Jetpack Compose. Jetpack Compose je nová, moderní, deklarativní sada nástrojů pro vytváření uživatelských rozhraní Android aplikací, jejíž verze 1.0 spatřila světlo světa v roce 2021 a která kompletně mění způsob, kterým byla uživatelská rozhraní Android aplikací doposud vyvíjena. Vývoj uživatelských rozhraní pomocí Jetpack Compose je ve světě vývoje Android aplikací v současné době velmi relevantním tématem, což je jeden z hlavních důvodů, proč bylo autorem práce toto téma diplomové práce zvoleno.

Diplomová práce se v teoretické části primárně věnuje představení základních principů vývoje uživatelských rozhraní pomocí Jetpack Compose. V teoretické části diplomové práce také dojde k uvedení hlavních pravidel moderní Android aplikační architektury. Praktická část se poté věnuje praktickému aplikování frameworku Jetpack Compose při vývoji aplikace s názvem Kolejka.

Cílem této práce je představit čtenářovi UI framework Jetpack Compose a sestavit funkční aplikaci, která vyhovuje moderní Android architektuře a plně využívá framework Jetpack Compose. Vyvinutá aplikace má za úkol sloužit jako sociální síť pro studenty ubytované na studentských kolejích Harcov. Dílčím cílem práce je porovnat vývoj Android aplikací pomocí Jetpack Compose a pomocí XML přístupu.

1 Úvod do Android OS a Jetpack Compose

První kapitola práce bude věnována krátkému představení operačního systému Android, v jehož prostředí poběží aplikace, jejíž vybudování bude popsáno v praktické části práce. Úvodní kapitola se bude dále věnovat sadě nástrojů Jetpack Compose, pomocí níž bude vytvořeno uživatelské prostředí aplikace.

1.1 Android

Android je open-source mobilní operační systém, který byl navrhnut pro přenosná, dotyková zařízení jako jsou smartphony a tablety. Operační systém Android funguje na bázi operačního systému Linux a byl původně vyvíjen firmou Android Inc., která je od roku 2005 dceřinou společností firmy Google (Gilski a Stefanski 2015) (Callaham 2021).

Velkým krokem pro vývoj operačního systému Android bylo zformování konsorcia Open Handset Alliance, jehož členy se staly hardwarové, softwarové a telekomunikační firmy, mezi něž patřily např. společnosti Acer, Samsung či T-Mobile. Open Handset Alliance byla již od počátku vedena firmou Google. Hlavním cílem OHA bylo vyvíjet technologie, které by značně snížily čas, náklady na vývoj a distribuci mobilních zařízení a služeb (Gilski a Stefanski 2015). Toto úsilí vedlo v říjnu 2008 k uvedení prvního telefonu s operačním systémem Android (ve verzi 1.0), který nesl název T-Mobile G1 (v některých státech byl telefon uveden pod jménem HTC Dream) (Callaham 2021).

K dnešnímu dni je operační systém Android nejprodávanějším mobilním operačním systémem na světě, s více než 70 % podílem na trhu (GlobalsStats 2022). To lze do určité míry přičíst velkému množství modelů mobilních zařízení s Android OS ve všech cenových relacích, díky čemuž je přístupný velkému počtu uživatelů. Nejaktuálnější verzí systému Android je verze 12, která byla vydaná na podzim roku 2021 (Google 2021).

Aplikace pro systém Android jsou vyvíjeny pomocí tzv. Android Software Development Kitu (SDK) v jazycích Java, Kotlin, ale i např. C++. Android SDK poskytuje knihovny pro tvorbu aplikací, ladící programy, či např. emulátor operačního systému (Techopedia 2020). Nejaktuálnější verze Android SDK je verze 31 (Google 2021).

Oficiálním integrovaným vývojovým prostředím (IDE) pro operační systém Android je Android Studio. Android Studio je postaveno na vývojovém prostředí IntelliJ IDEA, které

vyvíjí firma JetBrains sídlící v České republice, a bylo vytvořeno speciálně pro vývoj aplikací pro systém Android. Android Studio nabízí např. editor rozvržení, správce virtuálních zařízení, možnost integrace s platformou GitHub či propracované nástroje k testování aplikace (Google 2022).

Pro „sestavení“ (build) Android aplikace využívá Android Studio open-source nástroj Gradle, který slouží k automatizaci sestavování aplikací. Gradle kompiluje zdroje a zdrojový kód aplikací a balí je do souborů APK, které lze v systému Android nainstalovat. Dále umožňuje flexibilně konfigurovat projekt a jeho závislosti (dependencies) prostřednictvím konfiguračních souborů napsaných v jazycích Groovy nebo Kotlin (Google 2022).

1.2 Jetpack Compose

Sada nástrojů pro vývoj Android aplikací Jetpack Compose byla poprvé představena v roce 2019 firmou Google a verze Jetpack Compose 1.0 spatřila světlo světa v červenci roku 2021 (Lardinois 2019).

Zatímco dříve byla k vývoji Android aplikací vyžadována kombinace jazyků XML (starající se o vzhled UI) a Java/Kotlin (k programování funkcionalit aplikace), k tvorbě aplikací s Jetpack Compose stačí pouze jeden jazyk – Kotlin, programovací jazyk, který stejně jako vývojové prostředí, na kterém je založeno Android Studio, vyvíjí firma JetBrains. V současné době je Kotlin jazykem, který Google považuje za preferovaný jazyk k tvorbě Android aplikací (Hill 2020) (Google 2021).

Jetpack Compose je dle oficiální Google dokumentace (Google 2022, Thinking In Compose) „*deklarativní sada nástrojů, která zjednodušuje tvorbu a údržbu uživatelského rozhraní aplikace tím, že poskytuje deklarativní API, jenž umožňuje vykreslování uživatelského rozhraní bez nutnosti imperativních změn frontendových pohledů*“. Technika deklarativních uživatelských rozhraní spočívá v „rekonstrukci“ celého UI a aplikování pouze nezbytných změn v UI. Kromě Jetpack Compose se mezi deklarativní UI frameworky řadí také např. Flutter či React Native. Rekonstrukce celého UI je pochopitelně „drahá“ na zdroje, a to sice především na čas a výpočetní sílu. Tento problém je v případě Jetpack Compose řešen takzvanou rekompozicí, která bude vysvětlena později (Google 2022).

1.2.1 Composable funkce

Tzv. Composable funkce jsou základním stavebním kamenem uživatelských rozhraní v rámci Jetpack Compose. Jedná se o takové funkce, které na vstupu přijímají data a „emitují“ UI prvky (Google 2022). Tento popis bude dávat větší smysl na příkladu:

```
@Composable
fun DiplomkaComposable(
    icon: ImageVector,
    text: String
) {
    Row(modifier = Modifier.padding(4.dp)) {
        Icon(imageVector = icon, contentDescription = "")
        Spacer(modifier = Modifier.size(4.dp))
        Text(text = text)
    }
}
```

Ukázka kódu 1: Vytvoření composable funkce
(vlastní zpracování)

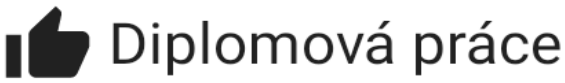
V ukázce kódu (Ukázka kódu 1) lze vidět příklad Composable funkce s názvem *DiplomkaComposable*. Všechny composable funkce začínají anotací *@Composable*, která informuje kompilátor, že tato funkce transformuje data do prvku uživatelského rozhraní.

Funkce v příkladu přijímá na vstupu dva argumenty: *icon* s datovým typem ImageVector (grafický objekt) a *text* ve formátu String. Tyto argumenty jsou dále využity jako vstupy interních prvků funkce *DiplomkaComposable*. Konkrétně se jedná o prvky **Icon** a **Text**, což jsou composable funkce, jež jsou poskytnuty v základu Jetpack Compose. Pro zobrazení vzhledu composable funkce lze použít anotaci *@Preview*.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    DP_UkazkyTheme {
        DiplomkaComposable(icon = Icons.Default.ThumbUp, text =
"Diplomová práce")
    }
}
```

Ukázka kódu 2: Použití funkce Preview
(zdroj: vlastní zpracování)

Pomocí této anotace lze vykreslit composable funkci tak, jak bude vypadat na obrazovce telefonu/tabletu. V autorem uvedeném příkladu bude composable funkce vypadat následovně (Obrázek 1):



Obrázek 1: Vykreslení jednoduché composable funkce
(zdroj: vlastní zpracování)

1.2.2 Modifikátory composable funkcí

Modifikátory v Jetpack Compose jsou třídy, které umožňují vizuálně, ale i funkčně upravovat jednotlivé composable funkce (Google 2022). Dle oficiální dokumentace mají modifikátory především za úkol následující:

- Měnit velikost, chování, vzhled a rozložení jednotlivých composable funkcí
- Dodávat informace, jako například štítky k označení přístupnosti (které jsou využívány např. službou Talkback, která slouží jako čtečka obrazovky pro osoby se zrakovými vadami)
- Zpracovávat uživatelské vstupy
- Doplňovat jednotlivé composable funkce o interakce – klikatelnost, zvětšovatelnost, apod.

Modifikátory lze využívat u composable funkcí, které jsou poskytnuty v základu jazyka, ale i v uživatelsky vytvořených composable funkcích. Ukázka modifikátorů bude opět uvedena na příkladu:

```
@Composable
fun DiplomkaComposable(
    modifier: Modifier = Modifier,
    icon: ImageVector,
    text: String) {
    Row(
        modifier = modifier.padding(4.dp)
    ) {Spacer(modifier = Modifier.width(4.dp))}}
```

Ukázka kódu 3: Ukázka modifikátorů composable funkcí
(zdroj: vlastní zpracování)

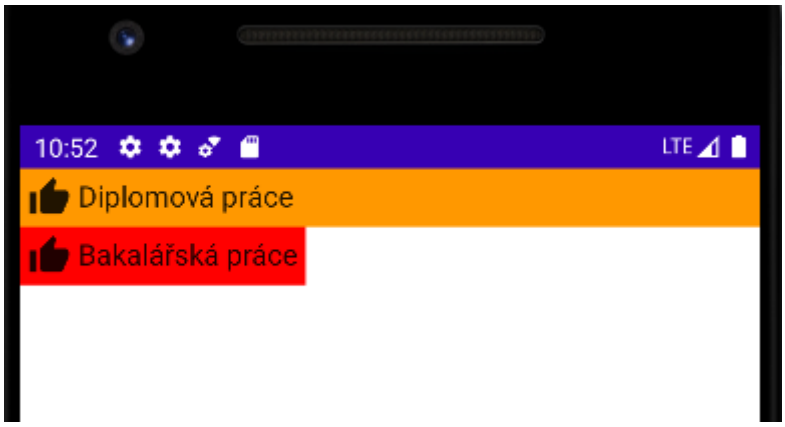
V ukázce kódu (Ukázka kódu 3) lze vidět, že mezi argumenty již dříve představené funkce *DiplomkaComposable* byl přidán argument **modifier**, což je objekt typu Modifier. Tento argument je dále použit v rámci celé composable funkce uvnitř prvku **Row**, kde je zároveň vyvolána jedna z mnoha funkcí objektu Modifier, a to sice funkce **padding**, která určuje šířku vnitřních okrajů daného prvku (JakPsátWeb 2016).

Pokud by byla funkce *DiplomkaComposable* využita v aplikaci, je následně dále možné upravovat její vlastnosti vyvoláním dalších funkcí objektu Modifier, viz příklad:

```
@Composable
fun DefaultPreview() {
    DP UkazkyTheme {
        var color by remember { mutableStateOf(Red) }

        Column {
            DiplomkaComposable(
                modifier = Modifier
                    .fillMaxWidth()
                    .background(Orange),
                icon = Icons.Default.ThumbUp,
                text = "Diplomová práce"
            )

            DiplomkaComposable(
                modifier = Modifier
                    .background(color)
                    .clickable { color = RandomColor() },
                icon = Icons.Default.ThumbUp,
                text = "Bakalářská práce"
            )
        }
    }
}
```

Ukázka kódu 4: Využití modifikátorů composable funkcí
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 4) lze vidět, že je zde dvakrát využita uživatelsky vytvořená composable funkce *DiplomkaComposable*, přičemž u prvního volání této composable funkce je u objektu Modifier vyvolána funkce **fillMaxWidth**, která informuje kompilátor o tom, že by měla daná composable zabírat celou šířku obrazovky, a funkce **background**, pomocí níž je možné nastavit barevné pozadí prvku. Funkce **background** je využita i v druhém použití *DiplomkaComposable*, vedle ní je ale také vyvolána funkce **clickable** - tzv. funkce vyššího řádu, tedy funkce, která přijímá jiné funkce jako své parametry. Funkce **clickable** v příkladu vykoná po kliknutí změnu barvy dané composable funkce.



Obrázek 2: Upravení composable funkce pomocí modifikátorů (zdroj: vlastní zpracování)

V případě modifikátorů je také lze třeba podotknout, že záleží na pořadí, v jakém jsou jednotlivé funkce objektu Modifier vyvolávány (Google 2022). Pokud by byl například na prvek aplikován **padding** a až poté funkce **clickable**, okraj zvětšený funkcí **padding** nebude na kliknutí reagovat. Změnou pořadí modifikátorů (clickable -> padding) již však bude reagovat celý prvek na kliknutí.

Ačkoliv je uživatelsky možné vytvořit vlastní funkce objektu Modifier, v rámci Jetpack Compose je jich velké množství zabudováno v základu – mezi autorem nejpoužívanější modifikátory patří například (kromě funkcí, které již byly zmíněny):

- **size** – nastaví přesnou velikost composable funkce
- **fillMaxHeight** – obdoba fillMaxWidth, zaplňuje ovšem celou výšku obrazovky (lze také využít **fillMaxSize**, která slouží k zaplnění celého prostoru, který může prvek využít)
- **matchParentSize** – funkce, která je použitelná pouze u composable funkce **Box** (která bude vysvětlena později), slouží k tomu, aby daný prvek zaplnil velikost svého „rodičovského“ prvku.
- **weight** – velikost jednotlivých Composable funkcí je dána obsahem, který „obalují“. Pomocí funkce weight je možné nastavit flexibilní velikost jednotlivých composable prvků (lze použít v composable funkcích Column a Row**,** které budou vysvětleny později)

```
@Composable
fun DefaultPreview() {
    DP UkazkyTheme {
        var color by remember { mutableStateOf(Red) }

        Row{
            DiplomkaComposable(
                modifier = Modifier
                    .weight(2f)
                    .background(Orange),
                icon = Icons.Default.ThumbUp,
                text = "Diplomová práce"
            )

            Text(
                modifier = Modifier
                    .weight(1f)
                    .background(Green)
                    .padding(5.dp),
                text = "Petr Horáček"
            )
        }
    }
}
```

Ukázka kódu 5: Použití modifikátoru weight
(zdroj: vlastní zpracování)

Obrázek 3: Vykreslení composable funkcí za pomoci modifikátoru weight
(zdroj: vlastní zpracování)

V uvedeném příkladu (Ukázka kódu 5) byla pro composable funkci *DiplomkaComposable* zvolána funkce **weight** s parametrem **2f**, a pro composable funkci Text **weight** s parametrem **1f**. To znamená, že tyto composable funkce zaberou dostupné místo „rodičovské“ composable funkce v poměru **2:1** (funkce weight je podobná vlastnosti flex-grow, jež se využívá v CSS v rámci tzv. Flexbox rozvržení (Coyier 2013).)

1.2.3 Stavy a rekompozice

Stav je důležitý pojem v rámci Jetpack Compose a z definice, kterou poskytuje oficiální dokumentace, se jedná o *„jakoukoliv hodnotu, která se mění v čase“* (Google 2022). V následujících situacích jsou popsány příklady stavu v Jetpack Compose:

- Pokud aplikace obsahuje composable funkci, která mění barvu pozadí, barvu pozadí lze brát jako stav.
- Pokud aplikace obsahuje composable funkci, která zobrazí obrázek po zmáčknutí tlačítka, zobrazitelnost obrázku je stav
- Pokud aplikace obsahuje textové pole, které ukazuje uživatelské jméno podle toho, jaký uživatel je přihlášený, text (textového) pole je stav
- …

Jak již bylo řečeno v úvodu práce, Jetpack Compose je deklarativní sada nástrojů k tvorbě UI, což znamená, že jediný způsob, jak UI změnit, je voláním určité composable funkce s jinými argumenty. Právě tyto argumenty reprezentují **stav** uživatelského rozhraní a každou změnou stavu composable funkcí dochází k tzv. **rekompozici** (Google 2022). Pro znázornění bude opět uveden příklad:

```
@Composable
fun DefaultPreview() {
    DP_UkazkyTheme {

        var color by remember { mutableStateOf(Red) }

        Row{
            DiplomkaComposable(
                modifier = Modifier
                    .weight(2f)
                    .background(color)
                    .clickable { color = RandomColor() },
                icon = Icons.Default.ThumbUp,
                text = "Diplomová práce"
            )
        }
    }
}
```

Ukázka kódu 6: Využití stavové proměnné
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 6) lze vidět, že je nejprve definována proměnná *color* s hodnotou **Red**. Tato proměnná je nastavena jako barva pozadí *DiplomkaComposable* a po kliknutí na tuto composable funkce by se barva pozadí měla náhodně změnit. K tomu ovšem nedochází, protože proměnná *color* není definována jako stav, tudíž se kliknutím na prvek barva nemění – Jetpack Compose si „není vědom“ změn (Klippenstein 2021).

Zároveň se composable prvek také při zapnutí nevykreslí v červené barvě (jak bylo definováno), protože k prvotní kompozici jsou také vyžadovány stavy.

Řešením by se mohlo zdát být uložení proměnné *color* jako objekt typu **MutableState**, což je tzv. sledovatelný (observable) datový typ, tedy typ, který umožňuje upozornit na změnu svých dat. MutableState uloží hodnotu jako stav. Definování proměnné pomocí MutableState by bylo následující (Ukázka kódu 7):

```
var color by mutableStateOf(Red)
```

Ukázka kódu 7: Objekt typu MutableState
(zdroj: vlastní zpracování)

I s touto definicí proměnné ovšem nastávají problémy. Použitím tohoto způsobu definice proměnné již nelze ani kód zkompilovat, neboť kompilátor hlásí error s hláškou „Vytvoření stavového objektu během kompozice bez použití *remember*“. Tento problém lze tedy jednoduše opravit, a to sice použitím klíčového slova **remember** (Ukázka kódu 8).

```
var color by remember {mutableStateOf(Red)}
```

Ukázka kódu 8: Použití klíčového slova remember
(zdroj: vlastní zpracování)

Remember je composable funkce, která si pamatuje hodnotu, jež jí je předána, a tuto hodnotu „vrací“, čímž je vytvořen stav, který se uchovává napříč rekompozicemi composable funkce (Elye 2021). Sama o sobě tato funkce není příliš nutná, protože pokud bychom měli hodnotu uloženou pouze pomocí remember (např. *var color by remember (Red)*), nelze ji měnit, jedná se o nezměnitelný stav. Tudíž je třeba v kombinaci s remember použít také MutableState. Klíčové slovo remember je nutné pouze uvnitř composable funkcí (a nelze ho využít jinde, takový kód se nezkompiluje). Mimo composable funkce lze ukládat stav např. pomocí ViewModel tříd, které budou vysvětleny později.

mutableStateOf je poté objekt rozhraní MutableState, který se využívá v případě, je-li vyžadována změna stavu composable funkce (Elye 2021).

Pokud by byl v rozsahu composable funkce použit pouze mutableStateOf k uložení informace (v situaci, kde by to bylo možné a kompilátor by v tom neviděl problém) (Ukázka kódu 9), bez remember by stejně nedocházelo ke správné funkčnosti:

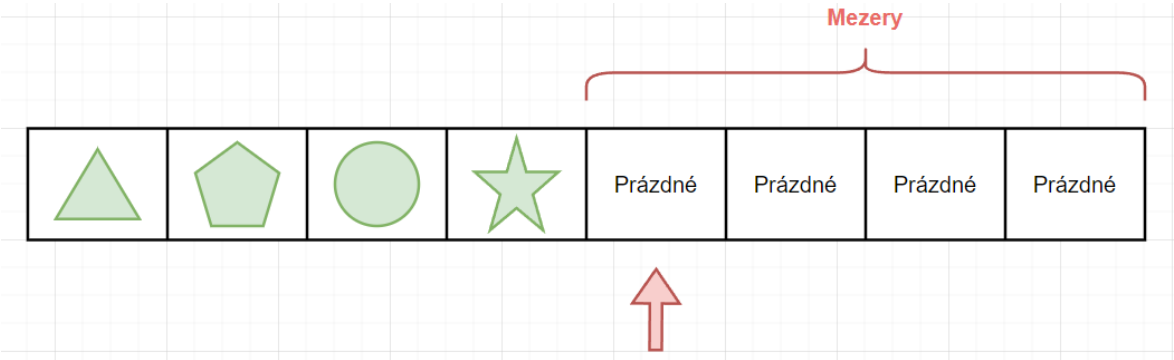
```
@Composable
fun DP Test() {
    var counter by mutableStateOf(0)

    Text(
        modifier = Modifier.clickable { counter++ },
        text = "Counter: $counter"
    )
}
```

Ukázka kódu 9: MutableState bez remember
(zdroj: vlastní zpracování)

Problém zde tkví v tom, že každým kliknutím by se sice zvedla hodnota proměnné *counter* o jedna, rekompozicí by ovšem znovu došlo k tomu, že proměnná načte hodnotu 0, neboť rekompozicí by se znovu vytvořila proměnná *counter*. V této situaci právě **remember** načte předchozí hodnotu stavové proměnné, která v rámci composable funkce přetrvá a nedojde k znovuvytvoření pokaždé, když proběhne rekompozice funkce (Elye 2021).

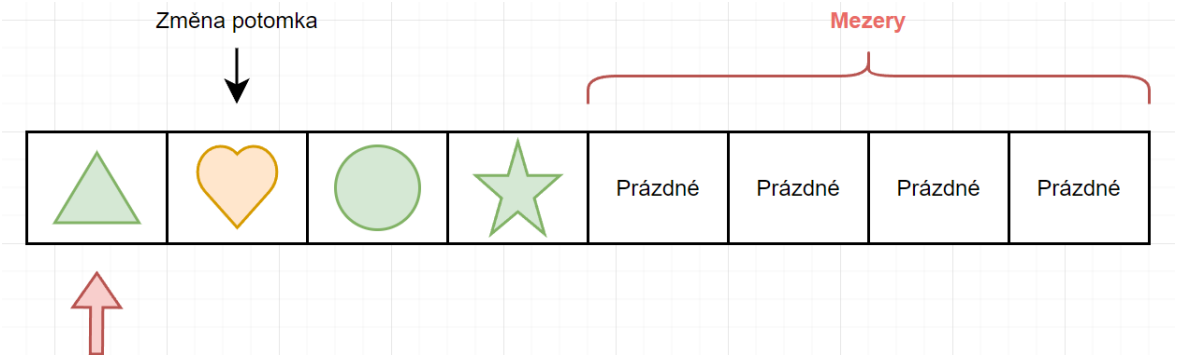
Kompozice a rekompozice composable funkcí je založena na datové struktuře zvané **gap buffer**. Gap buffer reprezentuje kolekci dat, která využívá aktuální index (kurzor) a v paměti je implementován jako jednodimenzionální pole. Toto pole je větší než kolekce dat, které reprezentuje a nevyužité místo pole se nazývá mezerou (gap) (Richardson 2020).

Ukládání composable funkcí si lze představit následovně:



Obrázek 4: Gap Buffer
(zdroj: vlastní zpracování dle (Richardson 2020))

Na obrázku (Obrázek 4) lze vidět pole, které je z části zaplněno objekty, které představují potomky „rodičovské“ composable funkce (které jsou samy composable funkcemi) a z části mezerami. Pole využívá také kurzor, který ukazuje na aktuální pozici v poli. Ve chvíli, kdy dojde k rekompozici „rodičovské“ composable funkce, kurzor se vrátí na první index pole a podle toho, zda se pro composable potomky změnila data, buď dojde k znovu vykreslení composable funkce či ne (Obrázek 5).



Obrázek 5: Změna v Gap Bufferu
(zdroj: vlastní zpracování dle (Richardson 2020))

Může také dojít k tomu, že mezi potomky composable funkce je vložen další potomek – v takovou chvíli mezery mění svou pozici na požadované místo a jedna z mezer se nahradí composable potomkem.

Z pohledu složitosti operací gap bufferu mají operace **get, insert a delete** složitost O(1) a operace **move** (přemísťování mezer) složitost O(n) (Richardson 2020).

V neposlední řadě je třeba zmínit tzv. **state hoisting**, pojem, který by se dal volně přeložit jako „zvedání stavu“. State hoisting je metodika přesouvání stavů do objektu, který danou composable funkci vyvolává, aby byla zachována bezstavovost této composable funkce (Google 2022). Obecný vzor pro state hoisting v Jetpack Compose tkví v nahrazování stavových proměnných v composable funkcích dvěma parametry:

- **value: T** (T značí jakýkoliv datový typ) - hodnota, která se zobrazuje
- **onValueChange: (T) -> (Unit)** - událost, která požaduje změnu hodnoty, kde T je navrhovaná nová hodnota (zde to ovšem není limitováno pouze na změny hodnot a je možné definovat různé události)

State hoisting bude ovšem lépe uveden na příkladu:

```
@Composable
fun DP Screen() {
    var name by remember { mutableStateOf("Petr Horáček") }
    DP Composable(text = name){
        name += " píše diplomovou práci"
    }
}
@Composable
fun DP Composable(
    text: String,
    onTextClick: () -> Unit,) {
    Text(modifier = Modifier.clickable { onTextClick() }, text = text)
```

Ukázka kódu 10: Použití state hoistingu
(zdroj: vlastní zpracování)

V příkladu (Ukázka kódu 10) jsou dvě composable funkce, *DP_Screen* a *DP_Composable*. *DP_Composable* představuje bezstavovou composable funkci, u které lze vidět, že přímo nevyužívá stavové proměnné, ale pro své vnitřní composable funkce (Text) utilizuje vstupní parametry *text* a *onTextClick*. Composable funkce *DP_Screen* následně slouží jako vyvolávací objekt *DP_Composable*.

Využívání state hoistingu poskytuje mnoho výhod, mezi něž patří například:

- **Zapouzdření** – pouze „stateful“ composable funkce mohou upravovat stavy, což zajišťuje vyšší přehlednost kódu
- **Možnost sdílení** – stavová proměnná, která byla přesunuta do vyvolávajícího objektu, může být použita v dalších composable funkcích (v uvedeném příkladu Ukázka kódu 10 by mohla být vytvořena composable funkce *DP_Composable2,* která by také přijímala vstupní argument typu String a při vyvolání objektem *DP_Screen* by mohla využít již dříve použitou stavovou proměnnou *name*)
- **Oddělitelnost** – stavová proměnná může být uchována na mnoha místech – v praxi tímto místem bývají nejčastěji třídy typu ViewModel, což jsou základní komponenty Android architektury, kterým bude věnována jedna z následujících kapitol (Google 2022)

1.2.4 Rozložení obrazovek v Jetpack Compose

Způsob, kterým je pomocí Jetpack Compose vytvářeno UI aplikací je fundamentálně odlišný od předchozího způsobu pomocí XML, a proto bude nyní vysvětleno několik základních principů, jak skládat a uspořádávat jednotlivé composable funkce aplikace.

Column a Row

Mezi základní „stavební kameny“ composable funkcí patří **Column** (sloupec) a **Row** (řádek). Jejich funkce je prostá: Column uspořádává jednotlivé composable prvky vertikálně pod sebe a **Row** uspořádává prvky horizontálně vedle sebe (Google 2022). Pokud by nebyl využit jeden z těchto prvků, UI aplikace bude vypadat následovně (Obrázek 6):

Obrázek 6: Rozložení bez použití Column či Row
(zdroj: vlastní zpracování)

```
@Composable
fun TestDP() {
    Text(text = "Ahoj.")
    Text(text = "Jmenuji se Petr Horáček.")
}
```

Ukázka kódu 11: Rozložení bez použití Column či Row
(zdroj: vlastní zpracování)

Jak je patrné, texty se překrývají, což je samozřejmě nežádoucí. Zabalí-li se composable prvky Text do prvku **Column**, UI bude mít podobu následující (Obrázek 8):

Obrázek 7: Rozložení s použitím Column
(zdroj: vlastní zpracování)

```
Column {
      Text(text = "Ahoj.")
      Text(text = "Jmenuji se Petr Horáček.")}
```

Ukázka kódu 12: Rozložení s použitím Column
(zdroj: vlastní zpracování)

A zabalením do prvku **Row** bude mít UI podobu následující (Obrázek 8):

Obrázek 8: Rozložení s použitím Row
(zdroj: vlastní zpracování)

```
@Composable
    Row {
        Text(text = "Ahoj.")
        Text(text = "Jmenuji se Petr Horáček.")
    }
}
```

Ukázka kódu 13: Rozložení s použitím Row
(zdroj: vlastní zpracování)

Column, stejně jako Row, přijímá na vstupu 3 argumenty. Prvním argumentem je **modifier**, modifikátor composable funkcí, který byl již představen dříve a má jej Column i Row. Při použití s composable funkcí Column bývá často využita funkce objektu Modifier **fillMaxSize**, neboť Column mnohdy bývá hlavním stavebním kamenem pro rozvržení obrazovky aplikace. Naopak u Row je poté často využívána funkce **fillMaxWidth**, která slouží k vyplnění celé šířky obrazovky.

Druhým argumentem composable funkce Column je poté **verticalArrangement**, který přijímá objekty typu **Arrangement.Vertical.** Tento argument slouží k vertikálnímu uspořádání prvků uvnitř sloupce (Google 2021). Je možné využít několika metod uspořádání, které Arrangement.Vertical poskytuje, mezi něž patří:

- **Arrangement.Top**
    - umístí prvky uvnitř sloupce svisle tak, aby byly co nejblíže hornímu okraji osy Y obrazovky

    - vizuálně: (Vršek obrazovky) 123 #### (Spodek obrazovky) (v tomto případě čísla reprezentují prvky uvnitř sloupce a křížky prázdný prostor)
- **Arrangement.Bottom**
    - umístí prvky uvnitř sloupce svisle tak, aby byly co nejblíže dolnímu okraji osy Y obrazovky
    - vizuálně: (Vršek obrazovky) ####123 (Spodek obrazovky)
- **Arrangement.Center**
    - umístí prvky uvnitř sloupce svisle tak, aby byly co nejblíže středu osy Y obrazovky
    - vizuálně: (Vršek obrazovky) ##123## (Spodek obrazovky)
- **Arrangement.SpaceBetween**
    - umístí prvky uvnitř sloupce svisle tak, aby byly rovnoměrně rozloženy na ose Y a zároveň aby před prvním prvkem a po posledním prvku nebylo žádné volné místo
    - vizuálně: (Vršek obrazovky) 1##2##3 (Spodek obrazovky) (v případě, že jsou ve sloupci pouze dva prvky, tak 1####2)
- **Arrangement.SpaceAround**
    - umístí prvky uvnitř sloupce svisle tak, aby byly rovnoměrně rozmístěny na ose Y, včetně volného místa před prvním prvkem a za posledním prvkem, ale zabírající poloviční prostor, než je jinak mezi dvěma po sobě jdoucími prvky
    - vizuálně: (Vršek obrazovky) #1##2##3# (Spodek obrazovky)
- Arrangement.SpaceEvenly
    - umístí prvky uvnitř sloupce svisle tak, aby byly rovnoměrně rozmístěny na ose Y, včetně volného místa před prvním prvkem a za posledním prvkem
    - vizuálně: (Vršek obrazovky) #1#2#3# (Spodek obrazovky) (Google 2021)

Obdobou verticalArrangement pro Row je **horizontalArrangement**, který oproti Column přijímá objekty typu Arrangement.Horizontal, jenž slouží k horizontálnímu uspořádání prvků uvnitř řádku. Metody má podobné jako Arrangement.Vertical, ale se dvěma rozdíly:

- Arrangement.Top je nahrazen Arrangement.Start, který prvky umísťuje horizontálně tak, aby byly co nejvíce vlevo na ose X

- Arrangement.Bottom je nahrazen Arrangement.End, který prvky umísťuje horizontálně tak, aby byly co nejvíce vpravo na ose X
- Zbytek metod funguje obdobně jako u funkce Column, akorát na ose X

Třetím argumentem composable funkce Column je následně **horizontalAlignment**, který přijímá objekty typu Alignment.Horizontal, který slouží k horizontálnímu zarovnání prvků uvnitř sloupce. Metody zarovnání nabízí tři (Google 2021):

- **Alignment.Start**
    - zarovná prvky k levému okraji sloupce
- **Alignment.End**
    - zarovná prvky k pravému okraji sloupce
- **Alignment.CenterHorizontally**
    - zarovná prvky na střed sloupce

Obdobou horizontalAlignment pro Row je verticalAlignment, který přijímá objekty typu Alignment.Vertical a také nabízí 3 metody:

- **Alignment.Top**
    - zarovná prvky k hornímu okraji řádku
- **Alignment.Bottom**
    - zarovná prvky k dolnímu okraji řádku
- **Alignment.CenterVertically**
    - zarovná prvky na střed řádku

Kromě Column a Row je běžně používanou composable funkcí k uspořádávání prvků na obrazovce také funkce **Box**. Box vyniká tím, že jednotlivé prvky, které obaluje, vrství na sebe (Google 2022). Výhodou composable funkce Box je také možnost libovolného zarovnání jednotlivých prvků, které obsahuje.

V kontextu rozložení v Jetpack Compose je také nutné zmínit composable funkci **Scaffold**, která poskytuje „sloty“ k přesnému umístění různých composable prvků. Mezi tyto prvky patří např. **top app bar**, horní lišta aplikace, která může obsahovat libovolná tlačítka k navigaci napříč aplikací, **bottom app bar**, dolní lišta aplikace, která je nejčastěji také využívána k navigaci, či např. **snackbar**, vyskakovací okénko, které slouží k zobrazení krátkých zpráv (nejčastěji informují o procesech aplikace) (Google 2022).

Všechny composable komponenty, jejichž umístění Scaffold řídí, patří mezi tzv. **Material komponenty**, komponenty (composable funkce), které jsou součástí **Material Designu**. Material Design je komplexní návrhový systém pro tvorbu uživatelských rozhraní vyvíjený firmou Google, který slouží nejen k designování nativních Android aplikací, ale také hybridních aplikací vytvořených jazykem Flutter a webových stránek (Material Design 2021).

Mezi další Material komponenty, které jsou v Jetpack Compose hojně využívány patří např. Button (tlačítko) či Card (karta, která může obsahovat libovolné UI prvky). Umístění těchto komponent ovšem není řízeno composable funkcí Scaffold (Material Design 2021).

V Jetpack Compose lze dále využívat i další metody k návrhu rozložení obrazovky (lze například použít tzv. layout modifier), nicméně většinu obrazovek lze vytvořit pomocí již zmíněných composable funkcí (Column, Row, ...), proto již nebudou ostatní metody dále rozebírány.

1.2.5 Vzhled obrazovek v Jetpack Compose

K vytváření konzistentního vzhledu jednotlivých obrazovek Android aplikací se v Jetpack Compose využívá tzv. „**theming**“ (tento pojem by se dal přeložit jako tematika či kompozice, ale v rámci práce bude autorem využíván pojem theming). Theming je v Jetpack Compose silně propojen s již zmíněným Material Designem, neboť jednotlivé Material komponenty jsou vybudovány na základech tzv. Material themingu, což je dle oficiální dokumentace „*systematický způsob, jak přizpůsobit Material Design, tak, aby lépe odrážel značku produktu*“ (Google 2021).

Theming (či Material theming) má 3 hlavní atributy, kterými jsou:

Color – v rámci themingu se považuje za nejlepší praktiku seskupování jednotlivých barev do datové třídy Colors (Material Design 2021). Pomocí této třídy lze definovat barvy aplikace podle specifikací Material Designu, které určují kolik barev a jaké typy barev by měla aplikace využívat. Specifikace typů barev jsou následující:

- **primary** – Primární barva aplikace, která je v dané aplikaci využívána nejčastěji, lze také definovat druhou variantu primární barvy, kterou lze využít např. ke zvýraznění

kontrastu mezi horní lištou aplikace a systémovou lištou (která bez definice druhé primární barvy využije hlavní primární barvu).

- **secondary –** Sekundární barva aplikace slouží k doplňování primární barvy, se kterou by měla ladit a zároveň poskytovat lepší barevný kontrast. Využívá se například pro ukazatele průběhu. Stejně jako u primární barvy lze definovat i druhou variantu sekundární barvy.
- **surface –** Tato barva by měla být využívána pro „povrchy“ composable funkcí, jako je například Card.
- **background –** Tato barva by měla být využívána především jako pozadí pod „posunovatelným“ obsahem (seznamy apod.).
- **error –** Barva, která by měla být využívána k indikaci chyb, které v aplikaci nastanou, například při zadání špatného hesla.
- **onColors** – Mezi další typy barev patří tzv. „on“ barvy, jedná se barvy, které by měly být používány na površích s předchozími typy barev - př. typ barvy *onPrimary* značí barvu, která se bude využívat např. pro text, který je umístěn na povrchu komponenty s primární barvou

Typography – slouží k seskupování textových stylů aplikace (Google 2021) podobně jako atribut Color. Typography definuje textové styly, které by aplikace měla (ale nemusí) využívat. Styly se podobají stylům značkovacího jazyka HTML a patří mezi ně např.:

- **h1** – pro velké nadpisy
- **body1 –** texty uvnitř karet, …
- a další: h2, h3, body2, ...

Shape – podobně jako předchozí dvě atributy, atribut Shape slouží k seskupování tvarů pro různé composable funkce aplikace (Google 2021), např. Card. Jetpack Compose k seskupování využívá třídu **Shapes**, která má 3 velikosti: small (využití: tlačítka, snackbary), medium (využití: karty, dialogy) a large (využití: drawery). Lze ovšem definovat i další velikosti a to pomocí tzv. extension funkcí, které nabízí jazyk Kotlin.

Při vytváření aplikace není povinností, aby programátor využíval tyto atributy, ale jejich použitím lze jednodušeji zachovat „čistota“ kódu (Google 2021). Jetpack Compose navíc třídy pro Color, Typography a Shape generuje při vytvoření JC projektu, což usnadňuje práci s počáteční konfigurací.

1.2.6 Grafika v Jetpack Compose

Jako poslední téma úvodní kapitoly bude stručně rozebrána grafika v Jetpack Compose. Mnoho aplikací vyžaduje vlastní komponenty s různými tvary, a právě k vytváření takových komponent slouží composable funkce **Canvas** (plátno). Uvnitř této composable funkce lze libovolně vykreslovat jednoduché tvary a přesně určovat jejich vzhled či polohu na obrazovce (Google 2022). Skládáním jednoduchých tvarů lze poté vytvářet přizpůsobitelnou, komplexní grafiku aplikace.

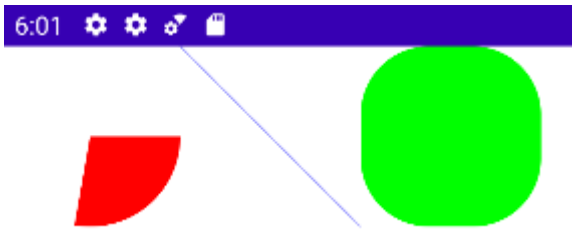
Composable funkce Canvas „odhaluje“ tzv. DrawScope, rozsah či oblast, kde mohou být využity speciální vykreslovací composable funkce, které jsou exkluzivní pro Canvas (Google 2022). Mezi tyto funkce patří např.:

- drawLine – nakreslí čáru mezi zadanými body pomocí zvolené barvy
- drawRect – nakreslí obdélník podle zadaného odsazení a velikosti
- drawCircle – nakreslí kružnici podle zadaných středových souřadnic a poloměru
- drawArc – nakreslí oblouk zmenšený tak, aby se vešel do daného obdélníku
- a další: drawOval, drawPoints, …

Ukázka využití Canvasu je uvedena na následujícím příkladu (Ukázka kódu 14):

```
@Composable
fun DP Canvas() {
        Canvas(modifier = Modifier.size(200.dp)) {
            drawArc(color = Red, startAngle = 0f, sweepAngle = 100f,
useCenter = true)
        }
        Canvas(modifier = Modifier.size(200.dp)) {
            drawLine(color = Blue, start = Offset(0f, 0f), end =
Offset(size.width, size.height))
```

Ukázka kódu 14: Použití composable funkce Canvas
(zdroj: vlastní zpracování)



Obrázek 9: Vykreslení tvarů pomocí composable funkce Canvas
(zdroj: vlastní zpracování)

Tato kapitola pochopitelně nepokrývá veškeré nástroje a možnosti, které Jetpack Compose nabízí, dalšími důležitými okruhy by pro Android vývojáře bylo např. řešení vedlejších efektů aplikace (v kontextu JC se vedlejšími efekty rozumí změny stavu aplikace, které se dějí mimo rozsah composable funkce (Google 2022)) či animace uživatelského rozhraní. Z pohledu autora práce ovšem není nutné se zaobírat těmito tématy, neboť nehrají příliš velkou roli v praktické části diplomové práce.

2 Srovnání Jetpack Compose a XML

Jak již bylo uvedeno v úvodní kapitole této diplomové práce, Jetpack Compose je **deklarativní** přístup k tvorbě UI, zatímco „tradiční“ způsob tvorby uživatelských rozhraní Android aplikací pomocí XML souborů se považuje za přístup **imperativní**. Rozdíl je mezi těmito přístupy následující:

- **Deklarativní přístup** – dle definice se jedná o „programování v jazycích, které odpovídají spíše mentálnímu modelu vývojáře než operačnímu modelu stroje“. Jiná definice popisuje deklarativní přístup jako „styl tvorby struktury a prvků počítačových programů, který vyjadřuje logiku výpočtu bez popisu jeho řídicího toku“. V jiných zdrojích lze poté naleznout vysvětlení deklarativního přístupu jako přístupu, který se více zabývá tím „*CO“* je potřeba operací aplikace vykonat (McGinnis 2016) (Haridas 2020).

- **Imperativní přístup** – podle dostupných definic lze tento přístup popsat jako „programovací paradigma, které používá příkazy měnící stav programu“. Na rozdíl od deklarativní přístupu se imperativní přístup zabývá tím „*JAK*“ je operace aplikace vykonána (Mejia 2019).

Z těchto způsobu vysvětlení ovšem nemusí být jasné, co jednotlivé přístupy opravdu představují. Proto se autor domnívá, že k objasnění problematiky je vhodnější popsat deklarativní a imperativní „smýšlení“ na příkladu:

- Muž si pomocí telefonu objednává jídlo z restaurace. Na druhé straně telefonátu je pracovník restaurace, který se muži položí otázku: „*Kam Vaše jídlo doručíme?*“. Pokud by muž, který si jídlo objednává, přemýšlel pomocí **imperativního** přístupu, odpověděl by např. větou: „Nejprve se doručovacím autem vydáte Petrovou ulicí na sever, až se dostanete na Diplomové náměstí, kde odbočíte doprava na Bakalářskou ulici. Na konci Bakalářské ulice najedete na kruhový objezd a vyjedete druhým výjezdem na Androidovu ulici, kde se nachází můj dům. Mé domovní číslo je 022“. Muž, který objednává, vysvětluje „*JAK*“ by k němu mělo být jídlo doručeno.

  Naopak, pokud by muž přemýšlel **deklarativním** přístupem, mohl by na otázku

odpovědět „Moje adresa je 022 Androidova ulice, Composov, 100 50, Česká republika“. Muž by tímto dal najevo „CO“ je cílem pro doručení jeho objednávky a dále by se již nezabýval tím, jakým způsobem (jakou cestou) mu bude jídlo doručeno.

Na závěr vysvětlování rozdílů mezi přístupy bude z hlediska relevantnosti k tématu JC vs. XML využit Haridasův (Haridas 2020) popis přístupů: „*Když je psáno deklarativní uživatelské rozhraní, říkáme počítači* ***„jaké uživatelské rozhraní se má zobrazit pro tuto hodnotu“*** *místo* ***"jak zobrazit uživatelské rozhraní pro tuto hodnotu“***.

Vysvětlením rozdílů mezi těmito důležitými pojmy ze světa softwarového vývoje lze nyní přejít k názorné ukázce mezi XML a Jetpack Compose. Pro oba typy vývojů bude vytvořena velmi jednoduchá příkladová aplikace, ve které se stisknutím tlačítka navýší zobrazené číslo o jedna.

2.1 XML

Pro tvorbu aplikací pomocí XML souborů je třeba, jak již bylo uvedeno v úvodu práce, využívání dvou jazyků: XML pro tvorbu UI a Java / Kotlin pro řízení funkcionalit aplikace. Při tvorbě uživatelského rozhraní je u tohoto typu vývoje důležité specifikovat typ rozložení, mezi něž patří tzv. LinearLayout či **ConstraintLayout**.

V uvedeném příkladu bude využit ConstraintLayout, což je typ rozložení, kde jsou veškeré komponenty rozloženy na základě vztahů mezi sebou a/nebo nadřazeným objektem, a to často tak, že jsou k sobě přichyceny. Tyto vztahy jsou definovány pomocí tzv. omezení (constraints) (Pavel 2018).

Kód uživatelského rozhraní poté vypadá následovně (Ukázka kódu 15):

```
<androidx.constraintlayout.widget.ConstraintLayout
    <TextView
        android:id="@+id/tvNumber"
        app:layout_constraintTop_toTopOf="parent"
    <Button
        android:id="@+id/btnIncrement"
</androidx.constraintlayout.widget.ConstraintLayout>
```

Ukázka kódu 15: Kód uživatelského rozhraní v XML
(zdroj: vlastní zpracování)

Nejprve je deklarováno, o jaký typ rozložení se jedná a jeho šířku a výšku. Uvnitř tohoto rozložení jsou pak dále deklarovány další komponenty, kterými jsou **TextView**, jednoduché textové pole a **Button**, tlačítko. Oběma těmto komponentům je nutné poskytnout identifikační element (*android:id*) a určit vztahy, které mají s ostatními komponentami/rozložením (např. kód *app:layout_constraintTop_toTopOf="parent"* znamená, že se vrchní strana komponenty „přichycuje“ k vrchní straně svého “rodiče“, což je v tomto příkladě rozložení ConstraintLayout).

S vytvořeným uživatelským rozhraní lze přejít do Kotlin/Java souboru (záleží na preferencích programátora, autor práce využil jazyk Kotlin), ve kterém je následně řízeno chování UI. Tímto souborem může být třída typu Activity (AppCompatActivity), Fragment či ViewModel. Tyto třídy budou popsány v dalších částech práce. Kód je pro popisovaný příklad následující (Ukázka kódu 16):

```
val textViewNumber = findViewById<TextView>(R.id.tvNumber)
val buttonIncrease = findViewById<Button>(R.id.btnIncrement)

buttonIncrease.text = getString(R.string.add_one)
var numberToIncrease = 0
textViewNumber.text = numberToIncrease.toString()

buttonIncrease.setOnClickListener {
        numberToIncrease++
        textViewNumber.text = numberToIncrease.toString()
}
```

Ukázka kódu 16: Řízení UI v XML
(zdroj: vlastní zpracování)

V prvních dvou řádcích se určí, se kterými komponentami UI bude manipulováno – do nezměnitelných proměnných (ukázka je psána v jazyku Kotlin, kde se nezměnitelné proměnné deklarují pomocí **val**) jsou uloženy informace o komponentách pomocí příkazu *findViewById*, pomocí něhož lze příslušné komponenty naleznout (příkaz je nalezne podle jejich identifikačního prvku, a proto je nutné pro každou komponentu deklarovat ID). Komponentami jsou v XML souborech tzv. **Views**, základní stavební prvky uživatelských rozhraní v XML (Google 2022).

V kódu je dále nastaven text, který budou při zapnutí aplikace oba komponenty zobrazovat a v závěru je využita funkce *setOnClickListener*, ve které se určí chování aplikace

po stisknutí tlačítka – v uvedeném příkladě dojde k navýšení proměnné *numberToIncrease* o jedna a následné úpravě textu komponenty *textViewNumber* tak, aby zobrazila správné číslo.

Lze vidět, že v aplikacích, ve kterých se využívají XML soubory k vytvoření uživatelského rozhraní je třeba přesně popsat *JAK* bude UI zobrazeno pro dané hodnoty a změnou hodnot se přímo manipuluje s atributy komponent (v uvedeném případě se s každou inkrementací musí upravit hodnota *textViewNumber.text*).

Obrázek 10: Jednoduchá aplikace vytvořena pomocí XML
(zdroj: vlastní zpracování)

2.2 Jetpack Compose

Pro tvorbu aplikací v Jetpack Compose se využívá pouze jeden jazyk, kterým je Kotlin. Kód je pro příkladovou aplikaci následující (Ukázka kódu 17):

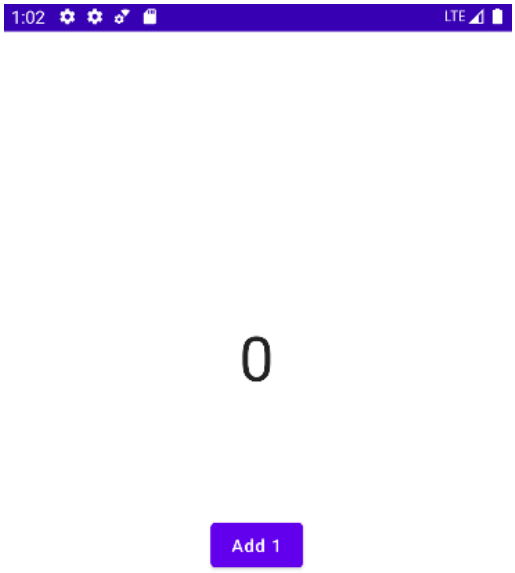
```
fun Counter() {
    var numberToIncrease by remember {mutableStateOf(0)}
    Column(
        modifier = Modifier.fillMaxSize()) {
        Text(text = numberToIncrease.toString()
        Button(onClick = {numberToIncrease++}) {
            Text(text = stringResource(R.string.add_one))}}}
```

Ukázka kódu 17: Kód uživatelského rozhraní v Jetpack Compose
(zdroj: vlastní zpracování)

Nejprve je uvnitř composable funkce deklarována stavová proměnná *numberToIncrease* a následně je vytvořeno UI pomocí prvků Column, Text a Button.

Pomocí tohoto způsobu tvorby uživatelského rozhraní určujeme, jaké UI bude pro danou hodnotu zobrazeno – s každou změnou stavové proměnné se totiž composable funkce, které onu stavovou proměnnou využívají, znovu vykreslí. Composable funkce je znovu vyvolána a dojde k rekompozici.

V tomto případě tak nedochází k průběžné manipulaci s jednotlivými prvky UI a úpravám jejich hodnot, místo toho jsou prvky znovu vytvořeny s novými hodnotami.



Obrázek 11: Jednoduchá aplikace vytvořena pomocí Jetpack Compose
(zdroj: vlastní zpracování)

Největším rozdílem mezi XML a Jetpack Compose je na první pohled délka kódu, který je pro identickou aplikaci nutný – to je v XML přístupu pochopitelně zapříčiněno tím, že jsou využity dva soubory, a také striktním rozložením ConstraintLayout, ve kterém je nutné definovat jednotlivá přichycení UI komponent. To ovšem nezabere příliš mnoho času, neboť ve vývojovém prostředí Android Studio lze intuitivně tato přichycení „naklikat“ v grafickém editoru rozložení aplikace a kód se následně vygeneruje automaticky. I tak je z pohledu autora neustálé přepínání mezi XML souborem pro UI a Java/Kotlin souborem pro řízení UI nepohodlné. Tento problém se dále ještě více prohlubuje u obsáhlých aplikacích, které mohou obsahovat desítky XML souborů.

Na druhou stranu, velkým plusem využívání XML souborů je jasný přehled o tom, jak bude uživatelské rozhraní vypadat, a to díky zmíněnému grafickému editoru rozložení. V Jetpack Compose lze k těmto účelům využít anotaci composable funkcí *@Preview*, která slouží k zobrazení, jak anotovaná composable funkce vypadá, ale tato metoda v současnosti neoplývá přílišnou rychlostí vykreslování. Proto je leckdy výhodnější (a rychlejší) vybudovat přímo celou aplikaci a zobrazit ji v emulátoru/telefonu.

2.3 Další rozdíly mezi XML a Jetpack Compose

Následující část bude věnována popisu dalších rozdílů mezi vývojem uživatelských rozhraní pomocí Jetpack Compose a XML přístupu.

2.3.1 Využívání aktivit a fragmentů

Při vytváření Android aplikací s XML uživatelským rozhraním hrají velkou roli tzv. Aktivity a Fragmenty. Aktivitu lze popsat jako součást aplikace, která poskytuje jednu z obrazovek uživatelského rozhraní Android aplikace (Jenkov 2014). V tomto ohledu se aktivity velmi podobají např. oknům v desktopových aplikacích. Na rozdíl od programovacích paradigmat, v nichž se aplikace spouštějí hlavní metodou **main**, se ovšem v systému Android spouští kód v instanci aktivity voláním specifických „callback“ method (onCreate, onDestroy,...), které odpovídají určitým fázím životního cyklu dané aktivity. Každá aktivita aplikace má přidružený jeden XML soubor, jehož funkcionalitu „spravuje“.

V počátcích Android vývoje byly aktivity hlavní způsobem, jak vytvářet uživatelská rozhraní Android aplikací, což se ovšem změnilo s příchodem verze systému Android, jež nese název Honeycomb (Tutorials Point 2015). V této verzi systému byly představeny tzv. Fragmenty, což jsou dle oficiální Google dokumentace (Google 2021) „*opakovaně použitelné části uživatelského rozhraní aplikace, které si spravují vlastní rozvržení, vstupní události a životní cykly, ale nemůžou žít samostatně, protože musí mít svojí hostitelskou aktivitu nebo jiný fragment*“. Fragmenty lze například využít jako „podobrazovky“ aplikace, které jsou drženy v hlavní, řídící aktivitě. Tato aktivita může poté dále spravovat navigaci mezi jednotlivými fragmenty.

Pomocí „tradičního“ vývoje s XML soubory jsou Android aplikace skládány následujícím způsobem:

- **Aplikace**
    - Aktivita 1 + Aktivita 1 UI XML (UI soubor může např. obsahovat navigační lištu)
        - Fragment 1 + Fragment 1 UI XML
        - Fragment 2 + Fragment 2 UI XML
    - Aktivita 2
        - Fragment 3 + Fragment 3 UI XML
        - …

K tomuto příkladu je nutné podotknout, že jednotlivé fragmenty by také měly mít svůj tzv. ViewModel, speciální třídu ke správě a držení dat pro fragmenty (ale i aktivity), ta bude ovšem popsána v rámci další kapitoly práce.

Tento způsob tvorby aplikací ovšem Jetpack Compose mění, neboť jednotlivými composable funkcemi lze fragmenty nahradit. Aplikace s JC pak mohou mít následující podobu:

- Aplikace
    - Aktivita
        - @Composable funkce ScreenOne
        - @Composable funkce ScreenTwo
        - …

Díky Jetpack Compose a způsobu, kterým řídí navigaci mezi jednotlivými obrazovkami aplikace, lze vytvářet Android aplikace, které obsahují pouze jednu hlavní aktivitu a žádné fragmenty – pouze composable funkce, které představují obrazovky (Masvita 2022). To je dle autora velkým plusem JC.

2.3.2 Design UI component

Poměrně velkým rozdílem mezi Jetpack Compose a XML je dále způsob, jak lze upravovat různé UI komponenty – tlačítka, textová pole atd. Tento rozdíl bude předveden na příkladu vytvoření tlačítka se zakulacenými rohy.

V případě využití XML je třeba vytvořit tzv. **drawable**, typ objektu, který lze vykreslit (Google 2022). Drawable objekty mohou mít mnoho forem: za drawable se považují např.

obrázky ve formátu PNG, či popisy tvarů, které mohou mít UI komponenty. Pro uvedený příklad se bude jednat právě o drawable tvaru, které mají formu XML souboru, ve kterém lze daný tvar popsat následujícím způsobem (Ukázka kódu 18):

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#512DA8" />
    <corners android:radius="300dp"/> </shape>
```

Ukázka kódu 18: Vytvoření drawable tvaru pomocí XML
(zdroj: vlastní zpracování)

Nejprve je deklarováno, že se jedná o popis tvaru (shape) a následně dojde k určení atributů pro daný tvar: v uvedeném případě je použit atribut **solid**, kterým se stanoví barva tvaru a **corners**, kterým se nastaví zakulacení rohů daného tvaru. Dalšími atributy poté mohou být např. gradient, či stroke (přidá tvaru ohraničení).

Pro použití tohoto tvaru je třeba jej nastavit jako pozadí dané UI komponenty, což je v tomto případě tlačítko. Přiřazení tvaru je provedeno následujícím příkazem (*button_shape* je název vytvořeného drawable objektu) (Ukázka kódu 19):

```
android:background="@drawable/button_shape"
```

Ukázka kódu 19: Přiřazení tvaru k tlačítku
(zdroj: vlastní zpracování)

Obrázek 12: Zakulacené tlačítko vytvořené pomocí XML
(zdroj: vlastní zpracování)

Při použití Jetpack Compose je vytvoření takového designu mnohem jednodušší, úpravy vzhledu tlačítka (a dalších composable komponent) se provádí přímo uvnitř dané komponenty (Ukázka kódu 20):

```
@Composable
fun JetpackCompose Button(
    onButtonClick: () -> Unit = {}
) {
    Box(modifier = Modifier.fillMaxSize()){
        Button(
            modifier = Modifier.align(Alignment.Center),
            onClick = onButtonClick,
            shape = RoundedCornerShape(size = 25.dp)
        ) {
            Text(text = "Jetpack Compose Button", fontSize = 20.sp)
        }
    }
}
```

Ukázka kódu 20: Vytvoření zakulaceného tlačítka pomocí Jetpack Compose (zdroj: vlastní zpracování)

Definování tvaru tlačítka se provede poskytnutím třídy **RoundedCornerShape** parametru *shape* (tato třída je součástí Jetpack Compose). V třídě RoundedCornerShape se poté pouze definuje velikost rohového rádiusu, čímž tlačítko získá svůj požadovaný tvar.

Obrázek 13: Zakulacené tlačítko vytvořené pomocí Jetpack Compose (zdroj: vlastní zpracování)

Je ovšem nutno podotknout, že v XML souborech lze také využívat komponenty knihovny Material Design, což ruší nutnost vytvářet drawable pro tvar, neboť např. zaoblení lze řešit přímo atributy komponenty (atribut *app:cornerRadius*).

Pokud by dále bylo v rámci aplikace např. potřeba vytvořit tlačítko s ikonou, použil by se atribut *app:icon*. Ovšem i pro jednotlivé ikony je třeba vytvořit v XML drawable objekt, nejčastěji vektorový drawable. Naopak, v Jetpack Compose je pro vytvoření tlačítka s ikonou nutné pouze přidat composable funkci **Icon** do těla composable funkce **Button**. Do těla tlačítka (a pochopitelně i jiných composable funkcí) lze poté v Jetpack Compose přidávat jakoukoliv composable funkci, což vede k takřka k neomezeným možnostem.

V následujícím příkladu (Ukázka kódu 21) je jednoduchým způsobem v Jetpack Compose vytvořeno tlačítko s ikonou a textovým polem, což by bylo v XML velice složité, ne-li nemožné:

```
Button(
    modifier = Modifier.align(Alignment.Center),
    onClick = onButtonClick,
    shape = RoundedCornerShape(size = 90.dp),
) {
    Icon(imageVector = Icons.Default.Home, contentDescription = "")
    Text(
        modifier = Modifier.padding(20.dp),
        text = "JC Button",
        fontSize = 20.sp
    )
    TextField(value = text.value, onValueChange = { text.value = it })
}
```

Ukázka kódu 21: Vytvoření tlačítka s textovým polem v Jetpack Compose (zdroj: vlastní zpracování)

V praxi nebude mít takové tlačítko pravděpodobně příliš velké využití, nicméně příklad poukazuje na silné stránky využívání Jetpack Compose – jednoduchou modifikaci UI komponent, kterou lze vyřešit v několika řádcích kódu.

2.3.3 Vytváření seznamů

Téměř v jakékoliv mobilní aplikaci se lze setkat s určitou formou seznamu položek – ať už se jedná o nabízené pokrmy restaurace, příspěvky na hlavní stránce sociálních sítí či výpis transakcí v bankovních aplikacích. Způsoby, kterými se v XML a Jetpack Compose takové seznamy vytváří, se od sebe znatelně liší, a proto budou v této části popsány.

Recycler View

V případě využívání XML souborů k tvorbě uživatelského rozhraní lze ke konstrukci seznamů položek použít tzv. Recycler View. Recycler View nese své jméno z důvodu, že při posouvání položek seznamu „neničí“ jednotlivé View komponenty (základní stavební prvky pro komponenty UI), ale znovu je využívá (recykluje) pro položky seznamu, které se nově objeví na obrazovce (Google 2022).

K zobrazení dat v Recycler View je třeba zajistit následující komponenty:

- Data
- Instanci komponenty RecyclerView definovanou v XML souboru obrazovky
- XML soubor s rozložením pro jednu položku seznamu
- ViewHolder – popisuje view položky seznamu a metadata o jeho umístění v RecyclerView
- Adapter – rozhraní, které slouží k poskytnutí dat do viewů, které jsou uvnitř ViewHolderu
- LayoutManager – třída, která se stará o organizaci komponent uvnitř RecyclerView

K lepšímu vysvětlení byl uveden jednoduchý příklad – byl vytvořen seznam textových položek.

Nejprve bylo třeba vytvořit zdroj dat, pro jednoduchost zde bude použit pouze list položek datového typu String (Ukázka kódu 22):

```
val predmety = listOf("Matematika", "Programování", "Počítačové sítě",
"Statistika")
```

Ukázka kódu 22: Vytvoření zdroje dat pro RecyclerView
(zdroj: vlastní zpracování)

Dále byla vytvořena instance RecyclerView uvnitř XML souboru obrazovky (Ukázka kódu 23):

```
<androidx.recyclerview.widget.RecyclerView
android:id="@+id/rv predmety"
android:layout width="match parent"
android:layout height="match parent"
app:layout constraintBottom toBottomOf="parent"
app:layout constraintEnd toEndOf="parent"
app:layout constraintHorizontal bias="0.5"
app:layout constraintStart toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />
```

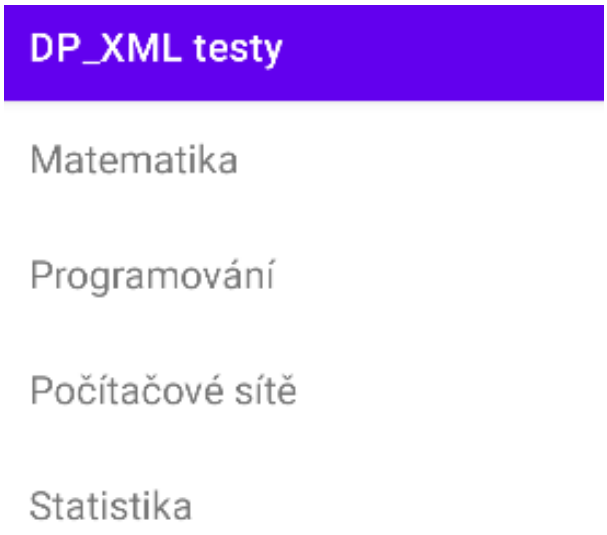
Ukázka kódu 23: Vytvoření instance RecyclerView
(zdroj: vlastní zpracování)

Následně bylo třeba vytvořit třídu typu Adapter, uvnitř které se také vytvoří instance třídy ViewHolder (Ukázka kódu 24):

```
internal class PredmetyAdapter(private var predmetyList: List<String>):
RecyclerView.Adapter<PredmetyAdapter.PredmetViewHolder>() {
    internal inner class PredmetViewHolder(view: View) :
RecyclerView.ViewHolder(view) {
        var predmetTextView: TextView =
view.findViewById(R.id.tv_predmet)
    }
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
PredmetViewHolder {
        val predmetView = LayoutInflater.from(parent.context)
            .inflate(R.layout.rv item, parent, false)
        return PredmetViewHolder(predmetView)
    }
    override fun onBindViewHolder(holder: PredmetViewHolder, position:
Int) {
        val item = predmetyList[position]
        holder.predmetTextView.text = item
    }
}
```

Ukázka kódu 24: Vytvoření Adapteru a ViewHolderu pro RecyclerView
(zdroj: vlastní zpracování)

Posledním krokem byla deklarace třídy *LinearLayoutManager* ve třídě, která řídí UI hlavního rozložení aplikace, kterou je v uvedeném případě *MainActivity*. Výsledný, jednoduchý seznam předmětů poté vypadá následovně (Obrázek 14):



Obrázek 14: Seznam vytvořen pomocí RecyclerView
(zdroj: vlastní zpracování)

Lazy Column

V případě použití Jetpack Compose se ke konstrukci seznamů v uživatelském prostředí aplikace používají composable funkce typu LazyColumn. Implementace této composable funkce bude opět uvedena na jednoduchém příkladu, ve kterém bude sestaven stejný seznam, jako byl vytvořen pomocí XML metody s využitím RecyclerView.

V počátečnímu kroku byla vytvořena composable funkce *PredmetyList*, která v sobě obsahuje pouze composable funkci LazyColumn a na vstupu přijímá argument *predmety* datového typu *List<String>*, což jsou při volání funkce její vstupní data.

```
@Composable
fun PredmetyList(
    modifier: Modifier = Modifier,
    predmety: List<String>) {
    LazyColumn(modifier = modifier.fillMaxSize()){
        items(predmety){ predmet ->
            Text(
                modifier = Modifier
                    .fillMaxWidth()
                text = predmet,
            )
        }
    }
}
```

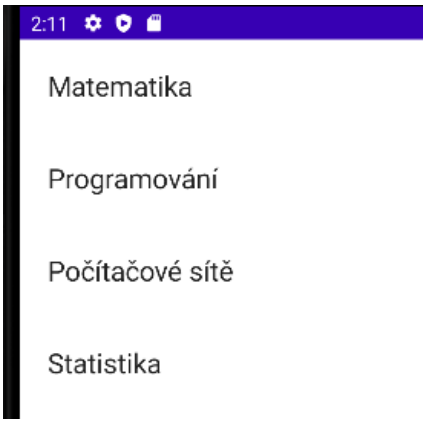
Ukázka kódu 25: Vytvoření composable funkce, která zobrazuje seznam položek (zdroj: vlastní zpracování)

Lze vidět (Ukázka kódu 25), že ve rámci působnosti (scope) funkce LazyColumn je vyvolána funkce *items*. Tato funkce slouží k vložení dat do composable funkce LazyColumn a jako vstupní argument ji je třeba poskytnout seznam hodnot. V tělu této funkce se poté definuje podoba jedné položky seznamu LazyColumn, což je v uvedeném příkladu pouze composable funkce Text.

Následně již pouze zbývalo vyvolat funkci *PredmetyList* a poskytnout ji požadovaná data.

```
Surface( val predmety = listOf("Matematika", "Programování", …)
    PredmetyList(predmety = predmety)}
```

Ukázka kódu 26: Volání composable funkce PredmetyList (zdroj: vlastní zpracování)



Obrázek 14: Výsledná podoba seznamu vytvořeného pomocí Jetpack Compose (zdroj: vlastní zpracování)

Jak lze vidět, vytváření seznamů předmětů je v rámci Jetpack Compose mnohem snazší a intuitivnější. Pro implementaci LazyColumn navíc není třeba vytvářet podpůrné třídy (Adapter, ViewHolder), což zásadně zkracuje délku kódu.

Rozdílů mezi metodami vytváření uživatelských rozhraní pomocí JC a XML je, kromě těch zmíněných pochopitelně mnohem více, vzhledem k tomu, že oba přístupy jsou od sebe diametrálně odlišné. A ačkoliv se může z uvedených příkladů jevit, že využití Jetpack Compose má navrch oproti XML metodě, je třeba stále brát v potaz fakt, že vytváření UI pomocí XML je metodou, která je používána již od počátků vývoje Android aplikací. Je tedy důkladně dokumentována a odladěna.

2.3.4 Výkonnostní rozdíly

Důležitou roli bude pro stávající vývojáře Android aplikací hrát také rozdíl v rychlostech vykreslování mezi Jetpack Compose a XML. Z výsledků testování, které vykonal W. Shelor, je patrné, že v rámci rychlosti vykreslování má XML přístup vývoje UI navrch oproti Jetpack Compose (Shelor 2021). W. Shelor ve svých poznatcích z testování uvádí, že v každém testovacím scénáři bylo vykreslování XML UI oproti JC UI rychlejší až o **33%**, což je znatelný rozdíl (Shelor 2021). Vykreslování Jetpack Compose prvků je dále ještě pomalejší v případě využití tzv. ComposeView komponent (možnost, jak použít composable funkce v rámci XML) uvnitř XML UI souborů.

Z hlediska výkonu je tedy patrné, že vykreslování XML UI je v současné době rychlejší než vykreslování Jetpack Compose. Je však pravděpodobné, že situace se v budoucnu změní, neboť Jetpack Compose podstupuje kontinuálním vývojem.

3 Architektura Android aplikací

Při vytváření aplikací, ať už mobilních, webových, či desktopových, je důležité definovat její aplikační architekturu. Popis aplikační architektury dle A. Bharadwaje (Bharadwaj 2019) je následující: „*Architekturu lze jednoduše popsat jako rozmístění tříd v aplikaci a způsob jejich komunikace. Při seskupování těchto tříd si nakreslíme přehled jejich rolí a odpovědností*“.

Podle oficiální Android dokumentace využívání aplikačně-architektonických přístupů dále umožňuje škálovatelnost aplikace, zvyšuje robustnost aplikace a usnadňuje její testování (Google 2021).

3.1 Zásady Android aplikační architektury

Pro vytváření Android aplikací jsou v oficiální dokumentaci vymezeny dvě hlavní zásady, které by měly být dodrženy v rámci návrhu aplikační architektury.

Oddělení zodpovědností (Separation of Concerns – SoC)

Princip zásady oddělení zodpovědnosti tkví v rozdělení aplikace do samostatných oddílů, tak aby každý oddíl řešil samostatný problém. Celkovým cílem oddělování zodpovědnosti je vytvoření dobře organizovaného systému, kde každá část systému plní smysluplnou a intuitivní roli a zároveň maximalizuje svou schopnost přizpůsobit se změnám (Natesan 2019) (Google 2021).

Z hlediska vývoje Android aplikací znamená využívání principu SoC dělení funkcionalit aplikace do jednotlivých tříd a rozhraní. Pokud by byla např. vytvářena aplikace, která provádí síťové požadavky, bylo by vhodné vytvořit třídu, která zprostředkovává požadavky, třídu, která požadavky zpracuje a třídu, která data obdržená z odpovědí serveru zpracuje pro vrstvu uživatelského rozhraní.

Řízení uživatelského rozhraní z datových modelů

Druhou základní zásadou je řízení uživatelského rozhraní z perzistentních datových modelů, které reprezentují data aplikace. Tyto modely jsou nezávislé na prvcích uživatelského rozhraní a životním cyklu aplikace. Používání perzistentních datových modelů je doporučováno ze dvou hlavních důvodů (Google 2021):

- Uživatelé nepřichází o data, pokud operační systém Android zničí aplikaci, aby uvolnil prostředky.
- Aplikace funguje i v případech, kdy je síťové připojení nestabilní nebo nedostupné.

Na základě těchto dvou zásad lze poté vymezit ideální architekturu Android aplikace. Každá aplikace by měla mít alespoň dvě vrstvy:

- **Vrstvu uživatelského rozhraní**, jejíž účel je zobrazení dat aplikace na obrazovce mobilního zařízení
- **Datovou vrstvu**, která obsahuje logiku, jak ukládat, vytvářet a měnit data aplikace (business logika) a vystavuje data aplikace UI vrstvě (Google 2021).

3.2 Aplikační vrstvy

Vrstva uživatelského rozhraní

Jak již bylo řečeno, hlavním úkolem vrstvy uživatelského je zobrazování dat aplikace. Jakákoliv změna v datech vyvolaná uživatelskou interakcí (např. přidání komentáře) či externím „zásahem“ (např. síťovou odpovědí) by se měla odrážet na stavu a vzhledu uživatelského rozhraní (Google 2021).

Vrstva uživatelského rozhraní se v Android aplikacích skládá ze dvou složek:

- **UI prvky** – prvky, které zobrazují data na obrazovce aplikace (v rámci Jetpack Compose se jedná o composable funkce)
- **Nosiče stavu** – třídy, které pracují s daty z datové vrstvy a vystavují je UI prvkům. V rámci Jetpack Compose především drží stav UI prvků. Jako nosič stavu se v Android aplikacích využívá třída typu ViewModel, která bude popsána podrobněji později v této kapitole.

Obrázek 15: Vrstva uživatelského rozhraní (zdroj: vlastní zpracování)

Datová vrstva

Datová vrstva Android aplikací se skládá z tzv. repositářů (repository) – tříd, které mají za úkol přistupovat ke zdrojům dat, ať již lokálním nebo vzdáleným, a tato data předávat dalším částem aplikace. Obecně by každý zdroj dat měl mít vlastní repositář. Repositáře obsahují tzv. business logiku aplikace a mezi úkoly, které řeší, patří například (Google 2021):

- Předávání data z datových zdrojů dalším částem aplikace
- Centralizování změn dat
- Abstrahování zdrojů dat od zbytku aplikace

Do typické architektury Android aplikací lze poté zařadit takzvanou **doménovou vrstvu**, která je „umístěna“ mezi UI vrstvu a datovou vrstvu. Doménovou vrstvu v Android aplikacích tvoří tzv. use case třídy, přičemž každá use case třída se vždy stará o jednu funkcionalitu aplikace a komunikuje s repositářem. Příkladem takové use case třídy může být např. *GetUserProfileUseCase* (konvence pro pojmenování use case tříd je „<*funkcionalita*>*UseCase*“), jejíž účelem by bylo získání určitého uživatelského profilu (GetUserProfile) z repositáře a předání těchto dat UI vrstvě. Využití doménové vrstvy je v rámci doporučené Android architektury volitelné (Google 2021).

3.3 Android Jetpack

Android Jetpack je sada knihoven, která pomáhá vývojářům Android aplikací postupovat podle osvědčených postupů (best practices), omezovat psaní šablonovitého kódu a psát kód, který funguje konzistentně napříč verzemi systému Android (Google 2022). Nástroje, které Android Jetpack nabízí, jdou tzv. ruku v ruce s oficiálními doporučeními pro architekturu Android aplikací, a tudíž budou některé z nich nyní představeny.

3.3.1 ViewModel

Třída ViewModel je typ třídy, jejíž úkolem je především poskytovat data UI prvkům aplikace. Jedná se o již zmíněný **„nosič stavu“**. Další hlavní funkcí třídy ViewModel je její schopnost udržet data i po změně konfigurace telefonu, např. po rotaci obrazovky (Google 2021).

Implementace třídy ViewModel v Android aplikaci bude dále uvedena na jednoduchém příkladu. V příkladové aplikaci je textové pole, tlačítko a seznam textových objektů. Po stisknutí tlačítka se má do seznamu (předmětů) přidat obsah textového pole. Kód bude vypadat následovně (Ukázka kódu 27):

```
@Composable
fun InformatikaPredmetyScreen() {

    val predmety = remember { mutableStateListOf("Informatika",
"Matematika", "Umělá inteligence","Statistika")}
    var newPredmetState = remember { mutableStateOf("") }

    Column(
        Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        LazyColumn(){
            items(predmety){ predmet ->
                Text(
                    text = predmet,
                    fontSize = 24.sp
                )}}
        Spacer(modifier = Modifier.size(24.dp))
        TextField(value = newPredmetState.value, onValueChange =
{newItemState.value = it})
        Spacer(modifier = Modifier.size(12.dp))
        Button(onClick = { predmety.add(newPredmetState.value) }) {
            Text(text = "Přidej předmět")
        }

    }
}
```

Ukázka kódu 27: Composable funkce bez třídy ViewModel
(zdroj: vlastní zpracování)

Kód v ukázce plní požadovanou funkci (přidání předmětu do seznamu po stisku tlačítka) dle očekávání, problém ovšem nastává ve chvíli, kdy je v textovém poli text a dojde k otočení displeje mobilního zařízení. V tu chvíli dojde k novému vykreslení UI prvku textového pole a text zmizí. V rámci Jetpack Compose lze tento problém vyřešit tak, že se stav textu (v kódu se jedná o *newItemState*) zabalí do funkce **rememberSaveable**, která uchovává stav i po změně konfigurace telefonu. Při konstrukci composable funkcí je ovšem třeba co nejvíce zachovávat pravidla pro bezstavovost composable funkcí, které se dosahuje pomocí již zmíněného state hoistingu.

Pro state hoisting bude tedy využita třída ViewModel, do které se přesunou stavové proměnné, jež jsou používány UI prvky aplikace. Třída ViewModel bude mít následující podobu (Ukázka kódu 28):

```
class InformatikaPredmetyViewModel: ViewModel() {

    private val _predmety = mutableStateListOf("Informatika",
"Matematika", "Umělá inteligence","Statistika")
    val predmety: SnapshotStateList<String> = _predmety

    private val _newItemState = mutableStateOf("")
    val newItemState: State<String> = _newItemState

    fun addPredmet(){
        _predmety.add(_newItemState.value)
    }

    fun setNewItemState(state: String){
        _newItemState.value = state
    }
}
```

Ukázka kódu 28: Přesun stavových proměnných do třídy ViewModel
(zdroj: vlastní zpracování)

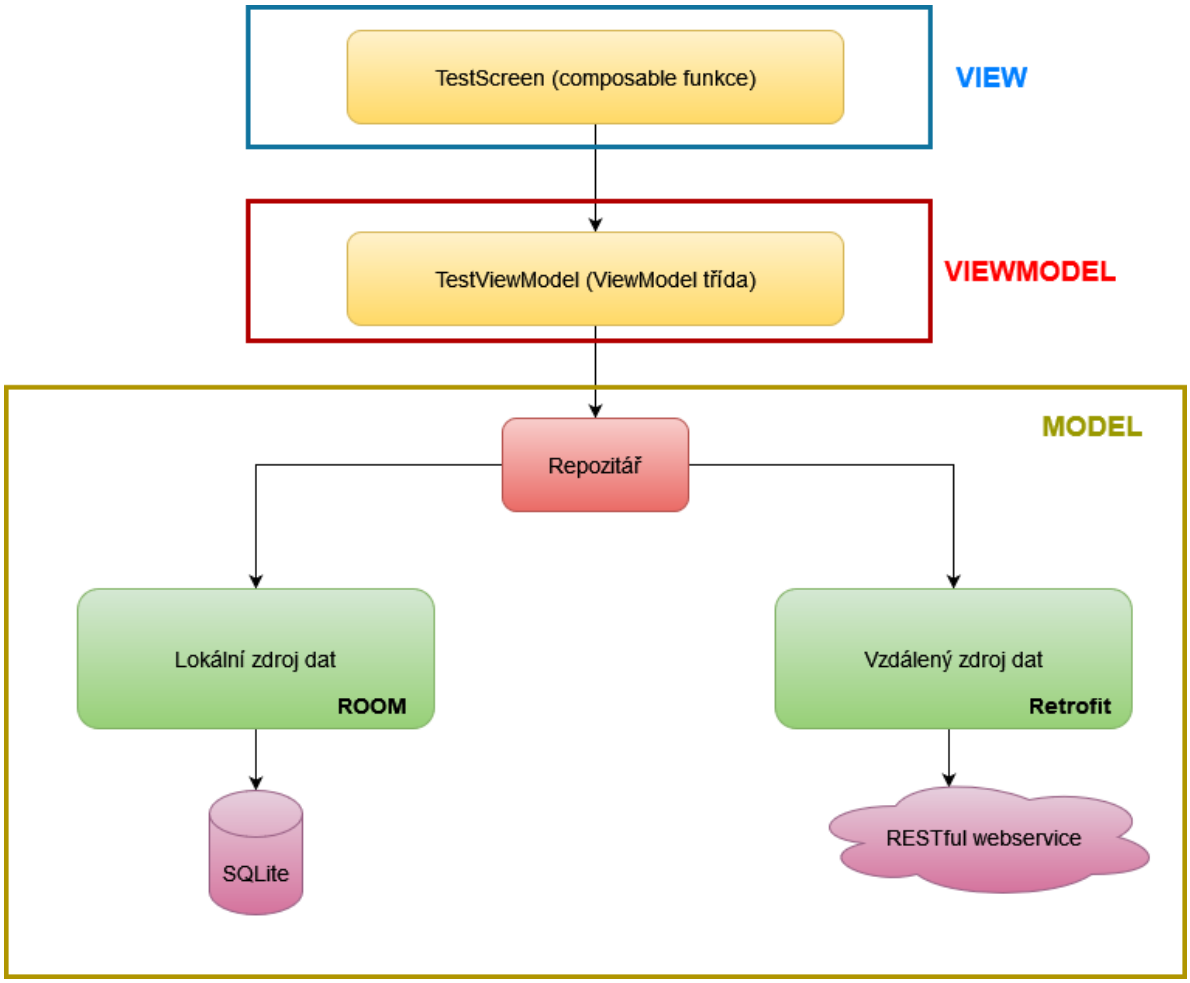
Nejprve si lze všimnout, že nově vytvořena třída InformatikaPredmetViewModel je odvozena z abstraktní třídy **ViewModel**. Uvnitř třídy jsou poté vytvořeny instance stavových proměnných predmety a newItemState – stavové proměnné, které byly přesunuty z composable funkce InformatikaPredmetyScreen (uvnitř ViewModel třídy byly tyto proměnné také zapouzdřeny, aby je bylo možné mimo třídu pouze číst). Uvnitř třídy byly také vytvořeny metody, které proměnné modifikují. Stavové proměnné je poté možné odstranit z composable funkce InformatikaPredmetViewModel a nahradit je instancí vytvořené třídy typu ViewModel (Ukázka kódu 29):

```
val informatikaPredmetyViewModel:InformatikaPredmetyViewModel=viewModel()
```

Ukázka kódu 29: Volání třídy ViewModel
(zdroj: vlastní zpracování)

Ve vytvořené composable funkci poté budou využívány pouze proměnné inicializované ve třídě ViewModel, tudíž jejich stav bude uchován i po změně konfigurace telefonu, a zároveň se zachovají pravidla state hoistingu.

Využívání tříd typu ViewModel poukazuje na návrhový / architektonický vzor, který se často využívá při vývoji Android aplikací. Je jím vzor **MVVM**, tedy *Model-View-ViewModel* (Kumar 2020). Způsob, jakým je tento vzor využíván je zobrazen na následujícím diagramu (Obrázek 16):



Obrázek 16: MVVM diagram
(zdroj: vlastní zpracování dle (Kumar 2020))

Část **View** slouží pouze k vytvoření uživatelského rozhraní a neobsahuje žádnou aplikační logiku. Data UI prvkům předává **ViewModel** (v Jetpack Compose pomocí stavových proměnných), přičemž data poskytují repositáře (které se zdroji dat tvoří **Model**). Repositář poté pomocí specializovaných knihoven získává data z lokální databáze či webových služeb (často pomocí Rest API) (Kumar 2020). Hojně využívanou knihovnou pro vytváření HTTP požadavků na webové služby je Retrofit a knihovnou pro komunikaci s lokální SQL databází je ROOM, která je také z jednou komponent Android Jetpack. Mezi repositáře a třídy ViewModel by poté šlo také přidat třídy typu UseCase (doménová vrstva).

3.3.2 Room

Součástí operačního systému Android je zabudovaná implementace databázového stroje SQLite. Knihovna perzistence ROOM poskytuje abstrakční vrstvu nad SQLite, která umožňuje plynulý přístup k databázi (Google 2022). Mezi funkce knihovny ROOM patří například:

- Ověřování SQL dotazů při kompilaci kódu
- Efektivní způsoby, jak migrovat databázi
- …

3.3.3 Navigační komponenta

Navigace aplikací je zásadním tématem ve vývoji mobilních aplikací. V každé aplikaci je nutné se pohybovat mezi jednotlivými obrazovkami, otevírat různá okna a následně se z nich opět vracet. Pro vývoj Android aplikací slouží k účelům navigace tzv. navigační komponenta. Navigační komponenta se skládá ze tří hlavní částí (Google 2021):

- Navigační graf – navigační graf obsahuje informace o všech navigovatelných lokacích v jedné, centralizované třídě NavGraph.
- Navigační hostitel – v Jetpack Compose se jedná o třídu, která zastřešuje všechny navigovatelné lokace aplikace
- Navigační kontroler – objekt, který zprostředkovává navigaci mezi jednotlivými lokacemi

XML způsob vývoje Android aplikací využíval pro vytváření navigačního grafu aplikace XML soubor, ve kterém byly jednotlivé lokace graficky vizualizovány. V Jetpack Compose se navigační komponenta implementuje odlišně, proto bude její použití předvedeno na příkladu:

V příkladu bude sestavena jednoduchá ukázka navigace mezi dvěma obrazovkami. Nejprve je nutné v aplikaci specifikovat obrazovky, napříč kterými bude docházet k navigaci. Ke specifikaci obrazovek lze v jazyku Kotlin využít např. speciální typ třídy *sealed class*, která představuje omezenou hierarchii tříd (pro takto jednoduchý příklad by šlo využít i tzv. *enum class*, typ sealed class je ovšem flexibilnější).

```
sealed class Screen(val route: String){
    object Home: Screen(route = "home_screen")
    object Settings: Screen(route = "settings_screen")
}
```

Ukázka kódu 30: Definice obrazovek pomocí sealed class

V sealed třídě Screen (Ukázka kódu 30) byly definovány dva objekty, Home a Settings, které představují obrazovky aplikace (domovská obrazovka a obrazovka nastavení). Každá z obrazovek má svojí **route**, cestu, která identifikuje navigační lokaci. Dále je potřeba vytvořit navigačního hostitele a graf. Pro navigačního hostitele je v Jetpack Compose vytvořena specializovaná composable funkce NavHost (Ukázka kódu 31).

```
@Composable
fun NavigationHost() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination =
Screen.Home.route){

        composable(route = Screen.Home.route){
            HomeScreen {
                navController.navigate(Screen.Home.route)
            }
        }
        composable(route = Screen.Settings.route){
            SettingsScreen()
        }}}
```

Ukázka kódu 31: Volání funkce NavHost
(zdroj: vlastní zpracování)

Nejprve se inicializuje navigační kontroler (*navController*), protože je třeba ho předat na vstup funkce **NavHost**. NavHostu je kromě navigačního kontroleru také specifikovat cestu počáteční navigační lokace, což je v uvedeném příkladu cesta domovské obrazovky. V tělu funkce NavHost se poté vytvoří navigační graf z jednotlivých funkcí *composable*, které představují jednotlivé obrazovky aplikace (kterými jsou v uvedeném příkladu obrazovky *HomeScreen* a *SettingsScreen*). Poté již pouze stačí funkci NavigationHost vyvolat, např. z hlavní aktivity aplikace.

Navigace aplikací je pochopitelně mnohem komplexnější téma, velkým tématem v rámci navigace je např. posílání argumentů (dat) mezi jednotlivými obrazovkami, řízení

tzv. backstacku (jednotlivé navigační lokace se ukládají do zásobníku a při zpětné navigace je třeba řídit, jak se bude tento zásobník pročišťovat), či např. integrace navigace se spodní, navigační lištou. Některé z těchto funkcionalit budou ovšem implementovány v aplikaci, která bude součástí praktické části této práce, proto nyní popisovány nebudou.

3.3.4 Dependency Injection

Ačkoliv Dependency Injection není přímou součástí knihovny Android Jetpack, využívání této techniky je doporučováno oficiální Android dokumentací, neboť napomáhá k přehlednosti kódu a zjednodušuje testování aplikace.

Dependency Injection neboli injektáž/vkládání závislostí, lze definovat jako *„soubor principů a vzorů návrhu softwaru, které nám umožňují vyvíjet volně vázaný kód“* (Hodges 2019). Další, méně obecná definice od J. Shora (Shore 2006) poté tvrdí, že injektáž závislostí znamená, že „objekty dostanou své instance proměnných“. Nejlépe lze injektáž ovšem vysvětlit na příkladu:

Třídy aplikací na sebe často potřebují vzájemně odkazovat. V rámci příkladu lze uvažovat o třídě *Přednáška*, která odkazuje na třídu *Učitel*. Třída *Přednáška* je **závislá** na třídě *Učitel* a třídu *Přednáška* nelze inicializovat bez instance třídy *Učitel*.

Třída *Přednáška* má dva základní způsoby, jak zajistit objekt, který potřebuje, kterým je instance třídy *Učitel*:

- Vytvoří si vlastní instanci třídy *Učitel*
- Třídu *Učitel* přijme jako parametr

V prvním způsobu by kód mohl vypadat následovně (Ukázka kódu 32):

```
class Ucitel(val jmeno: String)
class Prednaska(val id: String, val nazev: String){
    init {
        val ucitel = Ucitel(jmeno = "Petr Endrojd")
        println("$id: $nazev - vyučující: ${ucitel.jmeno}")
    }
}
```

Ukázka kódu 32: Vytváření vlastní instance třídy Učitel
(zdroj: vlastní zpracování)

Nejprve je deklarována třída *Učitel*, která přijímá parametr *jmeno*. Poté je deklarována třída *Přednáška*, která po inicializaci vypíše id, název přednášky a jméno učitele. V příkladu lze vidět, že si třída *Přednáška* vytvoří vlastní instanci třídy *Učitel*, kterou využije k výpisu informací o přednášce.

Ačkoliv tento způsob lze využít, v některých případech je použití této metody problematické. Třídy *Učitel* a *Přednáška* jsou totiž pevně spjaty, instance třídy *Přednáška* využívá jednu konkrétní instanci třídy *Učitel* a nemůže jednoduše použít k výpisu informací jinou. Pokud by bylo třeba vytvořit instanci třídy *Přednáška* s jiným učitelem, musela by být implementována jiná verze této třídy s rozdílnou instancí třídy *Učitel*, což by s rostoucím kódem vedlo k nepřehlednosti a komplikacím škálování programu.

Při využití druhého způsobu, tedy způsobu, kdy je třída Učitel předána jako parametr, by vypadal kód následovně (Ukázka kódu 33):

```
class Ucitel(val jmeno: String)

class Prednaska(val id: String, val nazev: String, val ucitel: Ucitel){
    init {
        println("$id: $nazev - vyučující: ${ucitel.jmeno}")
    }
}
```

Ukázka kódu 33: Předání třídy jako parametru
(zdroj: vlastní zpracování)

Kód se v tomto případě změnil tak, že třída Přednáška přijímá jako parametr třídu Učitel. Kromě toho probíhá výpis informací obdobně, jako v předchozím způsobu. Nyní je již ovšem možné vytvářet další instance třídy Přednáška, které umožní výpis rozdílných informací o jménech učitelů (Ukázka kódu 34).

```
fun main(){
    val prMatematika = Prednaska(id = "6ds56a", nazev = "Matematika 1",
ucitel = Ucitel("Petr Endrojd"))
    val prStatistika = Prednaska(id = "6ds57c", nazev = "Statistika 1",
ucitel = Ucitel("Josef Rahcov"))
```

Ukázka kódu 34: Použití třídy Přednáška s parametrem třídy Učitel
(zdroj: vlastní zpracování)

V druhém způsobu dochází k oné injektáži závislostí, třída *Učitel* je poskytnuta třídě *Přednáška*. V tomto případě se jedná o tzv. konstruktorovou injektáž, neboť závislost je injektována skrze konstruktor třídy. Dalším typem injektáže závislosti je tzv. field injection neboli injektáž polem, která se v Android aplikacích využívá v případech, kdy jsou určité třídy inicializovány systémem, a tudíž není možné provést konstruktorovou injektáž (Google 2021).

Mezi hlavní výhody využívání injektáže závislostí v Android aplikacích patří:

- **Znovupoužitelnost tříd a oddělení závislostí** – jak bylo ukázáno na příkladu, je jednodušší změnit instanci třídy v konstruktoru, než znovu vytvářet další třídy s různými závislostmi
- **Zlepšení přehlednosti kódu** – při využití konstruktorové injektáže lze snadněji naleznout použitou instanci třídy než v případě, kdy je instance „ukryta“ uvnitř inicializační metody třídy. To může např. napomáhat k jednoduššímu odhalování chyb v kódu.
- **Usnadnění testování** – během testů kódu lze třídě poskytnout různé instance tříd a otestovat na nich různé situace (Google 2021)

V uvedeném příkladu byla implementována jednoduchá injektáž závislosti, avšak při použití v obsáhlých aplikacích může využívání manuální injektáže vést k opakujícímu se kódu a zvýšené nepřehlednosti. Z těchto důvodů se používají knihovny, které injektáže závislostí obstarávají automaticky. Mezi tyto knihovny patří např. Dagger-Hilt, Koin či Guice. V aplikaci, která bude programována jako součást praktické části této práce, bude injektáž závislostí pomocí některých z uvedených knihoven implementována – jejich princip bude tudíž představen později.

3.3.5 Stránkování

Při vývoji mnoha typů aplikací je vyžadována funkcionalita k načítání mnoha položek – mohou to být předměty, které nabízí e-shop, příspěvky na sociální síti apod. V takových případech ovšem často není vhodné načítat všechny položky najednou, neboť to může vést k dlouhému načítání, či tzv. „sekání“ aplikace. Z těchto důvodů se v aplikacích, jak mobilních, tak i např. webových, aplikuje tzv. stránkování. Stránkování slouží k postupnému načítání položek po menších částech (po stránkách).

Při vývoji Android aplikací se ke stránkování využívá knihovna Paging, která je součástí sady Android Jetpack. Momentální verze knihovny Paging je verze 3 a poskytuje funkce, jako je například:

- Ukládání stránkovaných dat do mezipaměti
- Zabudovaná podpora pro zpracování chyb
- Podpora asynchronního programování pomocí Kotlin Coroutines/Flow či RxJava (Google 2022)

Stránkování pomocí Paging 3 bude implementováno v praktické části práce, proto bude tato knihovna blíže popsána později.

Sada Android Jetpack obsahuje dále mnoho dalších knihoven, mezi něž patří např. Data Binding, LiveData, či WorkManager. Tyto knihovny již detailněji popsány nebudou, neboť nebudou ani využity v praktické části práce.

4 Vývoj aplikace s využitím Jetpack Compose

Touto kapitolou začíná praktická část diplomové práce. Kapitola bude obsahovat uvedení programované aplikace a motivaci k jejímu vývoji, představení technologického „stacku“ aplikace a popis vývoje a implementace všech částí aplikace.

4.1 Uvedení programované aplikace

Aplikace, jejíž vývoj bude popsán v této kapitole, bude sloužit jako jednoduchá, kolejní „sociální síť“ pro studenty liberecké koleje Harcov. Motivací k tvorbě této aplikace byl fakt, že hlavní způsob, kterým se mohou studenti, kteří jsou ubytováni na Harcově, spojit, je skrze facebookové skupiny (tou největší je skupina s názvem „Koleje Harcov, Liberec“). V těchto skupinách často dochází k obecné diskuzi mezi studenty na různá témata, ale i např. nabízení různých předmětů, které pro studenty již nejsou potřebné či svolávání událostí. A právě na poslední dvě činnosti se zaměřuje aplikace, pro kterou bylo zvoleno jméno **Kolejka**.

Na Facebooku se totiž mnoho zmíněných nabídek „ztratí“ mezi spoustou dalších příspěvků a reklam, tudíž si jich mnoho uživatelů ani nevšimne. Facebook ve výchozím nastavení navíc nezobrazuje příspěvky chronologicky, a tak se může stát, že ačkoliv se jedná o např. nabídku předmětu, o který by měl daný uživatel zájem, příspěvek s nabídkou se mu zobrazí až ve chvíli, kdy je předmět prodán/zabrán.

V rámci aplikace Kolejka mohou uživatelé v základu nabízet předměty (či jakékoliv jiné věci) a svolávat mezi sebou jakékoliv události. Příspěvky s nabídkami či událostmi se zobrazí na hlavní obrazovce aplikace, s nejnovějším příspěvkem vždy na začátku stránky.

4.1.1 Funkcionalita a způsoby využití aplikace

K tomu, aby uživatelé mohli vytvářet události či vytvářet nabídky je v aplikaci možnost přidání příspěvku. Při vytvoření příspěvku typu **událost** se určí fotka pro danou událost, název, popis, datum, lokalita události a limit, který slouží ke omezení počtu zájemců, kteří se mohou dané události účastnit. Při vytvoření příspěvku typu **nabídka** se určí stejné informace, ale bez data. Po vytvoření příspěvku se příspěvek zobrazí na hlavní stránce aplikace, kde si jej mohou ostatní účastníci zobrazit a případně se k němu „přidat“. Přidání k příspěvku pro uživatele otevře možnost příspěvek komentovat a domluvit se s majitelem

příspěvku na případných detailech. Při zaplnění kapacity se příspěvek ztratí z hlavní obrazovky.

Pro přiblížení funkcionality aplikace bude na následujícím příkladu popsáno, k jakým účelům by uživatelé mohli aplikaci využívat:

Vytvoření události

Uživatel Petr je studentem prvního ročníku ekonomické fakulty v Liberci a je ubytován na kolejích Harcov. Petr je vášnivým hráčem videohry FIFA, a proto by chtěl s dalšími ubytovanými studenty uskutečnit turnaj v této videohře. Nikoho na kolejích ale nezná, proto v rámci aplikace Kolejka vytvoří událost „FIFA Turnaj“ a vyčká, až se k události přidají další uživatelé. S připojenými uživateli se poté dohodne na detailech události

Vytvoření nabídky

Uživatel Petr se po ukončení prvního ročníku studia stěhuje na léto pryč z kolejí. Na pokoji má ovšem ještě starou kancelářskou židli, kterou nedokáže transportovat domů. Petr v rámci aplikace kolejka vytvoří nabídku „Kancelářská židle za odnos“ s limitem 1, neboť má židli pouze jednu, a počká na zájemce. Se zájemcem se poté v komentářích dohodne na detailech.

Informace o limitu příspěvku byla zahrnuta z důvodu, aby nedocházelo k zahlcení komentářové sekce u jednotlivých příspěvků. Při naplnění limitu u příspěvku bylo také zamýšleno s funkcí, aby se daný příspěvek již nezobrazoval na hlavní stránce.

4.1.2 Technologický stack aplikace

Technologický stack představuje technologie, které aplikace používá na všech jejich vrstvách. Pro vývoj aplikace Kolejka byla využita tradiční třívrstvá architekturu, která obsahuje prezentační vrstvu, aplikační vrstvu a datovou vrstvu (IBM Cloud Education 2020).

Prezentační vrstva

Prezentační vrstvu představuje samotná aplikace, jejíž uživatelské rozhraní je napsáno pomocí frameworku Jetpack Compose v jazyce Kotlin. Aplikace se poté řídí aplikační architekturou, kterou doporučuje vývojář OS Android Google, s architektonickým vzorem MVVM. Aplikace dále používá následující, doplňkové knihovny:

- **Retrofit** – HTTP klient, který slouží k odesílání HTTP požadavků a komunikaci s REST API v aplikační vrstvě.
- **Coroutines** – knihovna, která poskytuje nástroje k asynchronnímu programování
- **Coil** – knihovna k načítání obrázků (např. z URL)
- **Dagger-Hilt** – knihovna, která umožňuje automatické vkládání závislostí, viz předchozí kapitola
- **UCrop** – knihovna k ořezávání obrázků
- **SwipeRefresh** – knihovna, která umožňuje aktualizaci stránky přejetím prstu přes obrazovku
- **Navigation Compose** – knihovna, která poskytuje funkce pro navigaci napříč aplikací
- **Cloudinary** – knihovna pro nahrávání obrázků do cloudové služby Cloudinary
- **MailDroid** – jednoduchý SMTP klient, který slouží k odesílání emailů skrze SMTP server

Aplikační vrstva

Aplikační vrstvu představuje serverová aplikace, vytvořená pomocí frameworku KTor, který je postaven na jazyku Kotlin. V Ktor serverové aplikaci byly dále využity dvě knihovny:

- **KMongo** – knihovna, která poskytuje sadu nástrojů pro práci s databází MongoDB v jazyce Kotlin
- **Koin** – knihovna, která umožňuje automatickou injektáž závislostí

Datová vrstva

Datová vrstva je představována databází, kterou je pro tento projekt NoSQL databáze MongoDB. Konkrétně byla využita MongoDB Atlas, cloudová verze MongoDB.

Podpůrné nástroje

V rámci vývoje byly dále použity následující technologie, které napomohly k usnadnění vývoje:

- **Github** – webová služba, která slouží pro správu verzí softwaru (tzv. Git repositářů) a umožňuje kolektivní práci na daném programu. V rámci práce byla služba Github využívána k zálohování a „transportu“ kódu – vývoj probíhal na dvou počítačích

(jeden počítač na univerzitních kolejích, jeden počítač v místě bydliště) a skrze Github bylo možné na jednom z počítačů importovat změny, které byly provedeny na druhém. Díky Githubu bylo také možné implementovat změny kódu bez obav, že by došlo k „rozbití“ aplikace, a to sice díky možnosti vytváření dalších, vývojářských větví.

- **Wakatime** – plugin, který lze nainstalovat do vývojového prostředí Android Studio, jež slouží k zaznamenávání času, který byl stráven aktivní prací na určitém projektu. Tento plugin byl zakomponován k účelům finálního zhodnocení práce na aplikaci.
- **Figma** – Figma je vektorový grafický nástroj, který v rámci práce sloužil k hrubému grafickému návrhu jednotlivých obrazovek. Výhodou nástroje Figma oproti jiným nástrojům, jako je např. Adobe XD, je fakt, že jej lze využívat zdarma
- **Heroku** – Heroku je webová služba, která umožňuje běh aplikací v poskytnutém, cloudovém prostředí. V rámci práce bylo Heroku použito k běhu serverové aplikaci.

4.2 Návrh UI a implementace

Vývoj aplikace byl rozdělen do tří hlavních částí: návrh UI a implementace, návrh datových modelů, API a jejich implementace, a nakonec propojení UI s aplikační vrstvou. Prvním krokem k vývoji aplikace Kolejka tedy bylo navrhnout uživatelské rozhraní aplikace. Jak již bylo řečeno, k tomuto účelu byl použit grafický nástroj Figma.

Každá aplikace by měla mít své charakteristické barevné schéma. Vzhledem k tomu, že aplikace byla vytvářena se záměrem cílit na studenty Technické univerzity v Liberci, ubytované na univerzitních kolejích Harcov, byla pro aplikaci zvolena primární barva vínová (Pantone 222 C, #6C1D45), což je základní barva TUL dle Manuálu jednotného vizuálního stylu TU v Liberci (Technická univerzita v Liberci 2011). Jako doplňující barvy poté byly zvoleny barvy světle šedá (#C4C4C4), tmavě šedá (#747476) a jemně narůžovělá, bílá barva (#F1EAEE).

4.2.1 Návrh uživatelského rozhraní

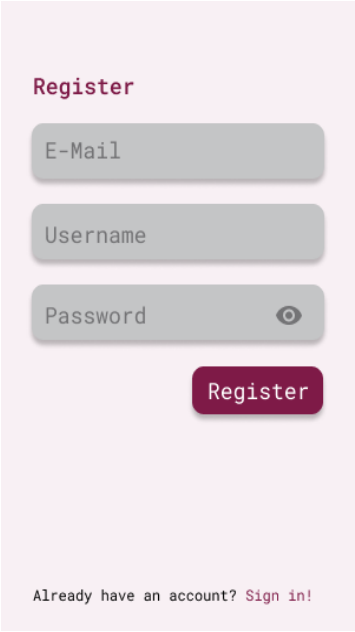
Předtím, než bylo možné vytvořit grafický návrh individuálních obrazovek aplikace, které představují uživatelské rozhraní aplikace, bylo nutné determinovat, jaké obrazovky bude aplikace mít. K tomu, aby bylo možné v aplikaci vykonávat požadované operace, bylo uvažováno s těmito obrazovkami:

- Registrační obrazovka – sloužící k vytvoření uživatelského účtu
- Přihlašovací obrazovka – sloužící k přihlášení registrovaného uživatele
- Hlavní obrazovka s příspěvky – zobrazující příspěvky všech uživatelů aplikace, rozdělená na dvě části: obrazovku s událostmi a obrazovku s nabídkami
- Notifikační obrazovku – sloužící k zobrazení upozornění na novou aktivitu u příspěvku přihlášeného uživatele
- Obrazovku profilu – zobrazující informace o přihlášeném uživateli, jeho příspěvky a příspěvky, ke kterým se přidal
- Obrazovku k editaci profilu – sloužící k zobrazení údajů o uživateli a možnosti úpravy těchto údajů
- Obrazovku k přidání příspěvku – sloužící ke vytvoření nového příspěvku (události či nabídky) s vlastním obrázkem, názvem apod.
- Obrazovku s detailem příspěvku – kterou by si mohli zobrazit uživatelé a pomocí níž by se mohli připojit k danému příspěvku a přidávat komentáře

Následně byl vytvořen hrubý návrh všech uvažovaných obrazovek pomocí nástroje Figma. Profesionalita grafického návrhu byla značně ovlivněna autorovou nízkou znalostí funkcí nástroje Figma, nicméně tímto návrhem byly „postaveny základy“, na kterých se dala aplikace Kolejka dále stavět.

4.2.1.1 Registrační obrazovka

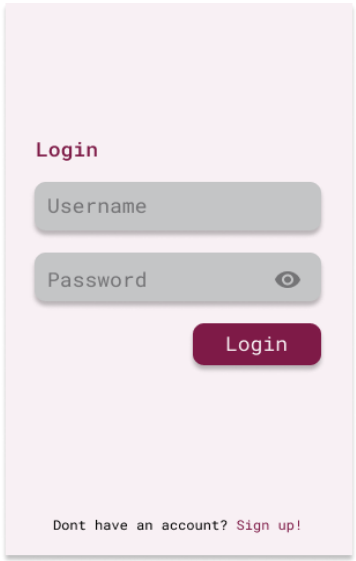
Návrh registrační obrazovky obsahoval tři jednoduchá textová pole pro e-mail, uživatelské jméno a heslo. Pro textové pole s heslem bylo také uvažována funkcionalita pro zobrazení hesla. Dále bylo uvažováno o registračním tlačítku a „klikatelném“ textu, který by již registrovaného uživatele přenesl do přihlašovací obrazovky.



Obrázek 17: Grafický návrh registrační obrazovky
(zdroj: vlastní zpracování)

4.2.1.2 Přihlašovací obrazovka

Návrh přihlašovací obrazovky nese stejný vizuální styl, jako obrazovka pro registraci uživatele s rozdílem, že obsahuje pouze textové pole pro uživatelské jméno a heslo (ve finální verzi bylo textové pole pro uživatelské jméno nahrazeno polem pro vyplnění emailu). Stejně tak, jako registrační obrazovka, obsahuje přihlašovací obrazovka „klikatelný“ text, který uživatele přenese do registrační obrazovky.



Obrázek 18: Grafický návrh přihlašovací obrazovky
(zdroj: vlastní zpracování)

4.2.1.3 Hlavní obrazovka s příspěvky

Při návrhu hlavní obrazovky s příspěvky bylo uvažováno o rozdělení obrazovky na dvě části – část s příspěvky typu událost a část s příspěvky typu nabídka. Mezi těmito dvěma částmi bylo zprostředkování navigace zamýšleno pomocí Android knihovny TabLayout. Hlavní obrazovka měla dle návrhu zobrazovat náhledy jednotlivých příspěvků, které by obsahovaly fotku příspěvku, název, fotku uživatele, který příspěvek přidal a část popisu příspěvku. V návrhu hlavní obrazovky bylo dále uvažováno o plovoucím tlačítku, které by umožňovalo navigaci do obrazovky pro přidání příspěvku. Hlavní obrazovka také měla obsahovat spodní navigační lištu aplikace, pomocí níž by docházelo k navigaci mezi třemi hlavními obrazovkami aplikace – obrazovkou s příspěvky, notifikační obrazovkou a obrazovkou profilu.

Obrázek 19: Grafický návrh hlavní obrazovky s příspěvky
(zdroj: vlastní zpracování)

4.2.1.4 Notifikační obrazovka

Notifikační obrazovka by měla dle návrhu obsahovat pouze jednoduché řádky, které by zobrazovaly informace o nové aktivitě u příspěvků přihlašovaného uživatele. V rámci notifikační obrazovky bylo zamýšleno se dvěma typy upozornění: upozornění na přidání jiného uživatele k příspěvku a upozornění na nový komentář u příspěvku.

Obrázek 20: Grafický návrh notifikační obrazovky
(zdroj: vlastní zpracování)

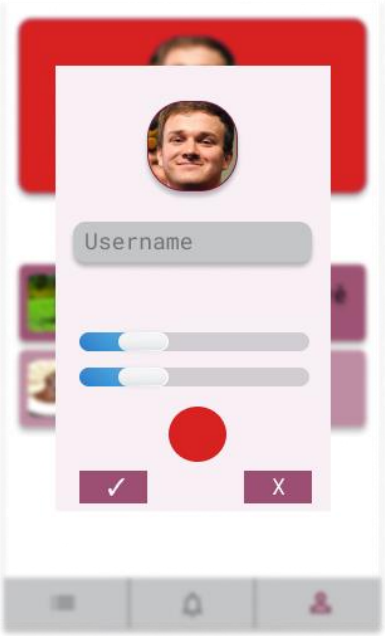
4.2.1.5 Obrazovka profilu

Profilová obrazovka by měla dle návrhu obsahovat „prapor“ (banner) uživatele, na kterém by byla vidět jeho profilová fotka, uživatelské jméno a tlačítko, které by otevřelo obrazovku pro editaci profilu. Dále bylo uvažováno s modifikovatelnou barvou praporu. Kromě praporu uživatele by měla obrazovka profilu dále zobrazovat uživatelem přidané příspěvky a příspěvky, ke kterým se přidal.

Obrázek 21: Grafický návrh obrazovky profilu
(zdroj: vlastní zpracování)

4.2.1.6 Obrazovka k editaci profilu

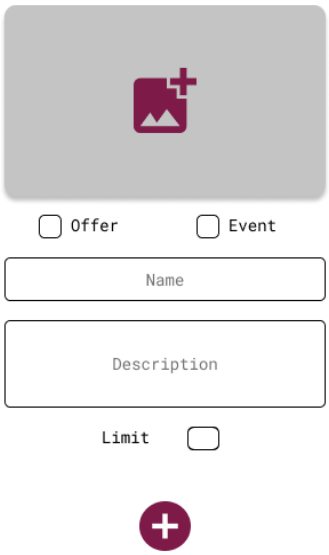
Pro editaci profilu bylo uvažováno nad editačním dialogem, který by se otevřel uvnitř obrazovky profilu. V rámci tohoto dialogu bylo dle návrhu zamýšleno nad možností změny profilového obrázku, uživatelského jména a barvy uživatelského praporu pomocí posuvníků, které by novou barvu zobrazovaly v kruhu, umístěném na spodku dialogu. Editační dialog by měl dále obsahovat tlačítko k potvrzení, či zamítnutí úprav.



Obrázek 22: Grafický návrh editační obrazovky
(zdroj: vlastní zpracování)

4.2.1.7 Obrazovka k přidání příspěvku

Obrazovka k přidávání příspěvků měla dle návrhu původně obsahovat tlačítko k přidání obrázku, tzv. *radio button* k volbě, zda je příspěvek událost či nabídka, a textová pole pro název příspěvku, popis a limit. Při vývoji aplikace byla později přidána možnost určit lokaci a datum (pro události). Nakonec by tato obrazovka měla obsahovat tlačítko k přidání příspěvku.



Obrázek 23: Grafický návrh obrazovky k přidání příspěvku
(zdroj: vlastní zpracování)

4.2.1.8 Obrazovka s detailem příspěvku

Dle grafického návrhu by měly být na obrazovce s detailem příspěvku obsaženy informace o daném příspěvku (fotka, název, popis, …). Dále by na této obrazovce mělo být zobrazeno množství „volných míst“ příspěvku a tlačítko, které by umožňovalo připojení k příspěvku. Nakonec by tato obrazovka měla obsahovat komentářovou sekci, která by byla pro uživatele zpřístupněna po připojení k příspěvku.

Obrázek 24: Grafický návrh obrazovky s detailem příspěvku
(zdroj: vlastní zpracování)

4.2.2 Implementace obrazovek pomocí Jetpack Compose

Dalším krokem ve vývoji aplikace byla samotná implementace jednotlivých obrazovek. K vývoji bylo přistupováno způsobem: vývoj UI -> vývoj serverové aplikace -> propojení UI a serverové aplikace. Tento způsob vývoje byl zvolen z důvodu možnosti testování jednotlivých funkcí serverové aplikace přímo v mobilní aplikaci s možností zobrazení výstupů.

Po vytvoření čistého Jetpack Compose projektu ve vývojovém prostředí Android Studio bylo nejprve třeba specifikovat **styly** pro aplikaci Kolejka. Vytvořením nového Jetpack Compose projektu se automaticky vygeneruje složka s názvem „Theme“ (téma). Uvnitř této složky jsou dále vygenerovány 4 soubory: Color, Shape, Theme, Type, které, jak již bylo zmíněno v teoretické části práce, slouží k popsání vzhledu celé aplikace.

V souboru Color byly nejprve definovány barvy, kde kromě hlavních barev aplikace (vínová, světle růžová, světle šedá a tmavě šedá) byly definovány doplňující barvy, např. barvy pro tlačítka potvrzení (zelená) a zamítnutí (červená).

V souboru Shapes lze dále definovat různě zaoblené tvary jednotlivých komponent aplikace, ta ovšem v rámci Kolejky zůstala nezměněna.

V souboru Type se poté definují různé styly fontu, které bude aplikace využívat – tučné písmo, tenké, polotučné, kurzívové apod. Android Studio ve výchozím nastavení poskytuje pro aplikace font Roboto. Pro aplikaci Kolejka byl zvolen font Roboto Mono.

Soubor Theme následně slouží k inicializaci jednotného stylu barev, tvarů a fontu pro všechny komponenty aplikace.

Po specifikaci stylů bylo dále třeba přidat aplikaci potřebné, doplňující knihovny. Jednotlivé knihovny lze do aplikace přidat pomocí souboru **build.gradle**, do kterého stačí vložit tzv. závislosti (dependencies). Gradle závislosti mají např. následující podobu:

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

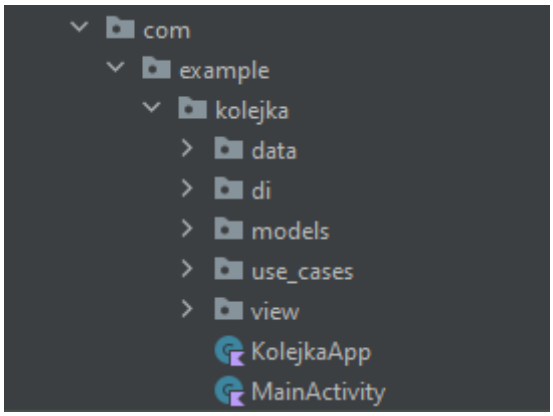
Ukázka kódu 35: Přidání knihovny Retrofit do Android projektu
(zdroj: vlastní zpracování)

V uvedeném příkladu (Ukázka kódu 35) je uvedena závislost, která do projektu aplikace přidá knihovnu Retrofit. Tento řádek kódu se jednoduše vloží do souboru *build.gradle* a provede se synchronizace souboru. Synchronizací je následně knihovna připravena k použití. V rámci aplikace Kolejka byly využity knihovny, které byly zmíněny v části práce, jež se týká technologického stacku aplikace.

Pro přehlednost v souborech projektu byly následně vytvořeny složky pro různé typy souborů.

Jednotlivé složky a některé z jejich podsložek jsou uvedeny v následujícím seznamu:

- **View** – složka, která obsahuje veškeré soubory, které se týkají UI
  - Theme – Androidem vygenerovaná složka pro vzhled UI
  - UI – složka s hlavními UI prvky
  - Screens – složka s jednotlivými obrazovkami a jejich držiteli stavu
  - Components – složitější composable funkce, které jsou využity na obrazovkách
  - Util – dodatečné soubory pro UI (např. navigace)
- **Data** – složka, které obsahuje soubory pro práci se serverovou aplikací (repositáře, …)
- **DI** – složka se soubory, které umožňují implementaci injektáže závislostí
- **Models** – složka obsahující třídy, které popisují hlavní objekty aplikace (např. třída User, Post, …) a které jsou načítány či ukládány z databáze
- **Use-Cases** – složka, která obsahuje třídy pro jednotlivé „use-casy“ aplikace (např. RegisterUseCase, …)



Obrázek 25: Složky v Android projektu aplikace Kolejka (zdroj: vlastní zpracování)

Složky Data, Use-Cases a DI byly vytvořeny až po vytvoření serverové aplikace, neboť obsahují soubory, které umožňují propojení s backendem.

Ačkoliv obsah složky Models představuje jednotlivé objekty, které aplikace ukládá a načítá z databáze, tato složka byla vytvořena ještě před implementací obrazovek aplikace, a to z důvodu, aby mohly být pro UI prvky aplikace poskytnuty tzv. „mock“ objekty, tedy testovací objekty, pomocí nichž bylo možné vytvářet UI bez reálných dat.

Mezi modely (hlavní objekty aplikace) v rámci aplikace Kolejka patří např. datová třída User, Post či Comment. Například třída Post má poté následující podobu (Ukázka kódu 36):

```
data class Post(
        val title: String,
        val id: String = "",
        val userId: String = "",
        val username: String = "", …
)
```

Ukázka kódu 36: Datová třída Post
(zdroj: vlastní zpracování)

Práce s těmito datovými třídami bude dále popsána v dalších částech práce.

V rámci prvotního nastavení projektu byly dále vytvořeny soubory, které popisují navigaci aplikací. Nejprve byla tvořena *sealed* třída (Ukázka kódu 37), která obsahuje objekty, které popisují jednotlivé obrazovky (v ukázce kódu nejsou uvedeny všechny).

```
sealed class Screen(val route: String) {
    object SplashScreen : Screen("splash_screen")
    object LoginScreen : Screen("login_screen")
    object RegisterScreen : Screen("register_screen"),…
}
```

Ukázka kódu 37: Sealed třída s jednotlivými obrazovkami
(zdroj: vlastní zpracování)

Poté již pouze stačilo přidat composable funkci typu NavHost a definovat v ní jednotlivé lokace aplikace (Ukázka kódu 38):

```
fun MainNavHost() {
    val mainNavController = rememberNavController()

    NavHost(navController = mainNavController, startDestination =
Screen.SplashScreen.route) {
        composable(route = Screen.SplashScreen.route) {
            SplashScreen(navController = mainNavController)
        }
    }
}
```

Ukázka kódu 38: NavHost navigace aplikace Kolejka
(zdroj: vlastní zpracování)

Zavedením navigace bylo provedeno základní nastavení projektu a dále již bylo možné se pustit do práce na jednotlivých obrazovkách. První implementovanou obrazovkou byla přihlašovací obrazovka.

4.2.2.1 Implementace přihlašovací obrazovky

Přihlašovací obrazovka měla dle návrhu obsahovat 5 hlavních komponent: text s nápisem „Login“, dvě textová pole pro e-mail a heslo, tlačítko pro přihlášení a „klikatelný“ text pro přechod do registrační obrazovky. Pro všechny z těchto komponent lze využít composable funkce, které jsou poskytnuty v základu Jetpack Compose, ovšem v této části byla vytvořena vlastní composable funkce. Jak si lze totiž v návrhu této obrazovky všimnout, jedno textové pole zobrazuje text normálním způsobem a druhé pole text skrývá (textové pole hesla).

Při použití composable funkce Textfield, kterou poskytuje Jetpack Compose lze nastavit různé parametry – mezi tyto parametry patří např. povolený počet řádků, barvy textového pole, ikona, která se zobrazí na začátku textového pole či typ klávesnice, který se při rozkliknutí určitého textového pole objeví. Pro různé typy textových polí (normální, pole s heslem, …) by ovšem v kódu bylo nutné definovat jednotlivá pole zvlášť, což při použití Jetpack Compose není nutné.

Ve složce *Components* byla vytvořena composable funkce *StandardTextField*, pomocí které lze v kódu jednoduše vytvářet všechny typy textových polí. V tělu této composable funkce je obsaženo textové pole, které poskytuje Jetpack Compose, ale jednotlivé vlastnosti tohoto pole jsou mu předávány skrze argumenty funkce *StandardTextField*. Zde byla využita velmi účinná vlastnost jazyka Kotlin, kterou je možnost poskytnutí výchozích hodnot parametrů

funkce, které funkce využije v případě, že ji není poskytnuta žádná hodnota pro daný parametr. Příklad využití výchozích hodnot v composable funkci StandardTextField vypadá následovně (Ukázka kódu 39):

```
@Composable
fun StandardTextField(
      maxLines: Int = 1,
      leadingIcon: @Composable (() -> Unit)? = null,
      visualTransformation: VisualTransformation =
VisualTransformation.None
)
```

Ukázka kódu 39: Parametry composable funkce StandardTextField
(zdroj: vlastní zpracování)

V uvedeném příkladu jsou vidět 3 z parametrů funkce StandardTextField, které využívají výchozí hodnoty. Např. první uvedený parametr *maxLines* určuje, na kolik řádků lze napsat text v daném textovém poli a je mu určena výchozí hodnota 1. Pomocí dalšího parametru *leadingIcon* lze dále určit ikonu, která bude zobrazena na začátku textového pole a výchozí hodnota tohoto parametru je **null**, tudíž bez explicitního určení ikony nebude použita žádná ikona.

Obrázek 26: Composable funkce StandardTextField
(zdroj: vlastní zpracování)

Po vytvoření composable funkce *StandardTextField* již bylo možné vytvořit uživatelské rozhraní přihlašovací obrazovky. Jednotlivé obrazovky jsou v Jetpack Compose composable funkcemi. Přihlašovací obrazovku tedy představuje composable funkce *LoginScreen*, která ve svém tělu obsahuje composable funkci Column, jež umožňuje skládání dalších composable funkcí vertikálně pod sebe. Vnitřními prvky composable funkce Column je poté např. composable funkce Text (vykresluje nadpis „Login“) nebo Button (tlačítko, jehož stisknutím dojde k přihlášení). Hlavní roli na této obrazovce pochopitelně také hraje vytvořená funkce *StandardTextField*, jejíž volání vypadá následovně:

```
StandardTextField(
    modifier = Modifier.fillMaxWidth(),
    text = emailState.text,
    hint = stringResource(R.string.email),
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email),
    error = when (emailState.error) {
        Errors.EmptyField -> stringResource(id =
R.string.this_field_cant_be_empty)
        else -> ""
    },
    textStyle = MaterialTheme.typography.h2,
    onTextChanged = { viewModel.onEvent(LoginEvent.EnteredEmail(it)) },
    placeholderTextColor = DarkGray,
    placeholderTextStyle = MaterialTheme.typography.h2
)
```

Ukázka kódu 40: Composable funkce StandardTextField přizpůsobena pro vkládání emailů (zdroj: vlastní zpracování)

V uvedeném příkladu (Ukázka kódu 40) je *StandardTextField* přizpůsoben pro vkládání emailů, tudíž je typ klávesnice (*keyboardOptions*) nastaven na *KeyboardType.Email*. Zobrazuje také chyby vstupu pro emailové textové pole. Tato composable funkce je dále použita pro textové pole hesla, kde ji je dále např. specifikován parametr, který upravuje, jak se text zobrazuje (skrytě nebo klasicky).

Na svém spodku má přihlašovací obrazovka composable funkci Text, která je doplněna o modifikátor **clickable**, jež umožňuje na text kliknout a vykonat určitou funkci. V případě aplikace Kolejka je onou funkcí navigace do registrační obrazovky.

4.2.2.2 Implementace registrační obrazovky

Implementace registrační obrazovky probíhala obdobně jako u obrazovky k přihlašování. Jediným rozdílem bylo využití třech textových polí typu *StandardTextfield*.

Jak u přihlašovací, tak i u registrační obrazovky bylo dále potřeba řídit UI události, ke kterým na těchto obrazovkách dochází uživatelskou interakcí. UI událostmi je myšleno např. zadání textu do textového pole email, zadání textu do textového pole heslo, stisknutí tlačítka registrace apod.

K řízení událostí bylo nejprve třeba vytvořit držitele stavu (stavových proměnných) pro jednotlivé obrazovky, kterými jsou v rámci aplikace Kolejka zmíněné třídy typu ViewModel. Např. ve ViewModelu přihlašovací obrazovky (*LoginViewModel*) byla

vytvořena stavová proměnná pro textové pole emailu a hesla. Dále byla vytvořena třída typu *sealed* s názvem *LoginEvent*, která zastřešuje veškeré události přihlašovací obrazovky. Třída LoginEvent má následující podobu (Ukázka kódu 41):

```
sealed class LoginEvent{
    data class EnteredEmail(val email: String): LoginEvent()
    data class EnteredPassword(val password: String): LoginEvent()
    object Login: LoginEvent() …
}
```

Ukázka kódu 41: Sealed třída LoginEvent
(zdroj: vlastní zpracování)

Tato třída je poté využita ve třídě *LoginViewModel*, která obsahuje funkci *onEvent*, jež jako svůj jediný parametr přijímá jednu ze tříd z uzavřené třídy *LoginEvent*. Implementace této funkce poté vypadá následovně (Ukázka kódu 42):

```
fun onEvent(event: LoginEvent) {
    when (event) {
        is LoginEvent.EnteredEmail -> {
            _emailState.value = _emailState.value.copy(…)
        }}
```

Ukázka kódu 42: Ukázka funkce onEvent
(zdroj: vlastní zpracování)

Funkce onEvent je dále volána z composable funkce přihlašovací obrazovky s náležejícími argumenty. Například textové pole pro zadání e-mailu obsahuje parametr *onTextChanged*, který přijímá lambda funkci, která má specifikovat, co se stane při změně textu v poli. Jako argument je pro tento parametr poskytnuta právě funkce *onEvent*, a to sice následujícím způsobem (Ukázka kódu 43):

```
onTextChanged = { viewModel.onEvent(LoginEvent.EnteredEmail(it)) }
```

Ukázka kódu 43: Využití funkce onEvent
(zdroj: vlastní zpracování)

Řízení událostí je u dalších obrazovek je poté prováděno stejným způsobem jako u přihlašovací obrazovky.

4.2.2.3 Implementace hlavní obrazovky s příspěvky

Dalším krokem vývoje uživatelského rozhraní bylo vytvořit obrazovku, která slouží k zobrazování všech příspěvků. Tato obrazovka byla dle návrhu jednou z obrazovek, do které se lze navigovat pomocí spodní navigační lišty (*bottom navigation bar*) aplikace.

Nejprve tedy bylo nutné tuto lištu implementovat a první fází implementace byla konstrukce datové třídy *BottomNavItem*, která slouží k popisu jednotlivých předmětů (lokací) spodní lišty. Tato třída má parametry *name* (jméno lokace), *route* (navigační cestu k dané lokaci), *icon* (ikonu předmětu spodní lišty) a *notificationsCount* (počet notifikací, tento parametr byl použit pouze pro lokaci obrazovky notifikací). Další fází implementace spodní lišty poté bylo vytvoření samotné lišty, což je v rámci Jetpack Compose velice snadné, neboť v základu JC je poskytnuta composable funkce *BottomNavigation*, pomocí které lze lištu jednoduše zkonstruovat. Byla tedy vytvořena composable funkce *BottomNavigationBar*, uvnitř které jsou nejprve definovány jednotlivé lokace spodní lišty jako seznam instancí třídy *BottomNavItem* (Ukázka kódu 44):

```
val bottomBarItems = listOf(
    BottomNavItem(
        name = stringResource(id = R.string.posts),
        route = Screen.PostScreen.route,
        Icons.Outlined.Menu
    ), …
```

Ukázka kódu 44: Seznam lokací spodní lišty aplikace
(zdroj: vlastní zpracování)

a následně byla vyvolána composable funkce *BottomNavigation*. Pro *BottomNavigation* je poté třeba vytvořit jednotlivé lokace, které bude obsahovat, což je možné pomocí composable funkce *BottomNavigationItem*, jež je také poskytnuta v rámci JC. Tato composable funkce se využila v kombinaci s vytvořeným seznamem bottomBarItems následujícím způsobem:

```
bottomBarItems.forEach { item ->
    val selected = item.route == backStackEntry?.destination?.route
    BottomNavigationItem(
        selected = selected,
        label = {
            Text(
                text = item.name,
                style = Typography.caption,
                color = DarkPurple
            )
        } …
}
```

Ukázka kódu 45: Propojení seznamu s lokacemi a composable funkce BottomNavigationItem (zdroj: vlastní zpracování)

Jak lze vidět v ukázce kódu (Ukázka kódu 45), pro každý předmět v seznamu byla pomocí cyklu *forEach* vytvořena composable funkce *BottomNavigationItem*, která se stará o správné zobrazí předmětu (lokace) ve spodní liště.

Pro zobrazení spodní lišty dále bylo třeba vytvořit obrazovku, která měla za úkol „držet" spodní lištu. Tato obrazovka (composable funkce) byla nazvána *AppHolder*. Uvnitř této obrazovky byla vyvolána composable funkce Scaffold, která se stará o správné umístění některých komponent – mezi tyto komponenty patří i spodní navigační lišta. V těle funkce Scaffold poté byla vyvolána vytvořená composable funkce NavHost. Dohromady vypadá konstrukce funkcí Scaffold a NavHost následovně (Ukázka kódu 46):

```
Scaffold(
bottomBar = {
    BottomNavigationBar(
        navController = navController, …}) {
    innerPadding ->
    Box(modifier = Modifier.padding(innerPadding)) {
        NavHost(
            navController = navController
        )}}
```

Ukázka kódu 46: Composable funkce Scaffold obsahující NavHost (zdroj: vlastní zpracování)

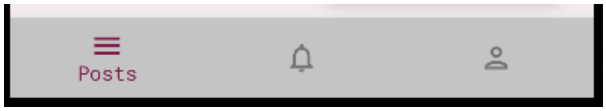
Poslední věcí, kterou bylo nutné vyřešit, bylo zobrazování spodní lišty pouze na obrazovkách, do kterých se lze pomocí lišty navigovat. Těmito obrazovkami jsou obrazovka s příspěvky, notifikační obrazovka a obrazovka profilu. Voláním composable

funkce, která obsahuje spodní navigační lištu způsobem, který byl uveden, se ovšem spodní lišta zobrazí na všech obrazovkách aplikace. Bylo tudíž nutné přidat omezení pro zobrazování, a to sice následujícím způsobem (Ukázka kódu 47):

```
val screensWithBottomBar = listOf(
    Screen.PostScreen.route,
    Screen.NotificationScreen.route,
    Screen.ProfileScreen.route)
val backStackEntry by navController.currentBackStackEntryAsState()
val showBottomBar = backStackEntry?.route in screensWithBottomBar
if (showBottomBar) {
    BottomNavigation(
        modifier = modifier,
```

Ukázka kódu 47: Omezení zobrazení spodní lišty aplikace
(zdroj: vlastní zpracování)

Nejprve byl vytvořen seznam s obrazovkami *screensWithBottomBar*, u kterých se spodní lišta má zobrazit. Poté byla deklarována proměnná *backStackEntry*, která obsahuje informace o momentální navigační lokaci. Dále byla vytvořena proměnná *showBottomBar*, která nabývá hodnoty *true*, pokud je momentální navigační lokace prvkem seznamu *screensWithBottomBar*. Nakonec byla composable funkce BottomNavigation zabalena do podmíněného příkazu *if*, který umožňuje vykreslení spodní lišty pouze v případě, kdy má proměnná *showBottomBar* hodnotu *true*.



Obrázek 27: Spodní navigační lišta aplikace
(zdroj: vlastní zpracování)

Po úspěšné implementaci spodní navigační lišty bylo možné začít vytvářet composable funkce pro obrazovky, které jsou lokacemi této lišty. První z těchto obrazovek byla **hlavní obrazovka s příspěvky**. Tato obrazovka obsahuje posuvný seznam s jednotlivými náhledy příspěvků. Jelikož pro takové náhledy není v základu Jetpack Compose obsažena composable funkce, bylo nutné si ji vytvořit samostatně.

Composable funkce náhledu příspěvku byla pojmenována *PostComposable* a ve svém těle obsahuje základní composable funkce, které jsou obsaženy v Jetpack Compose – tělo náhledu představuje composable funkce Card, u které je specifikováno, že má zaoblené rohy,

je vyvýšená (parametr *elevation*) a okraje má zabarvené podle typu příspěvku (event/offer). Card má dále ve svém těle funkci Column, která umožňuje vertikální skládání jednotlivých komponent. Uvnitř funkce Column jsou již poté obsaženy composable funkce, které zobrazují informace o příspěvku tak, jak bylo navrženo v nástroji Figma. Oproti návrhu byly ovšem v závěru změněny barvy funkce *PostComposable*, neboť šedá barva náhledu, která byla původně zamýšlena nevypadala příliš vkusně a byla zaměněna za bílou barvu s velmi jemným nádechem růžové. Finální podoba *PostComposable* je znázorněna na následujícím obrázku (Obrázek 28):

Obrázek 28: Náhled příspěvku v aplikaci Kolejka
(zdroj: vlastní zpracování)

Zobrazení příspěvků v seznamu bylo následně jednoduše implementováno pomocí composable funkce LazyColumn. Nejprve byla vytvořena obrazovka (composable funkce) pro zobrazování příspěvků, která ve svém těle volá právě funkci LazyColumn, jež načítá jednotlivé příspěvky. Poněvadž při implementaci UI ještě nebyl vytvořen backend aplikace, příspěvky byly k testování načítány ze seznamu „mock“ objektů třídy *Post*. Testovací načítání příspěvků bylo prováděno následovně (Ukázka kódu 48):

```
LazyColumn{items(postList){ post ->PostComposable(post = post,
onPostClick = {})}}
```

Ukázka kódu 48: Testovací načítání příspěvků
(zdroj: vlastní zpracování)

Hlavní obrazovku s příspěvky bylo dále nutné rozdělit na dvě části: na část s příspěvky typu *Event* a část s příspěvky typu *Offer*. Pro přepínání mezi těmito částmi byla tudíž vytvořena

horní navigační lišta, jejíž implementace se podobá implementaci spodní lišty. Prvním krokem bylo vytvoření třídy, která popisuje jednotlivé lokace horní lišty. Pro popis byla využita třída TabItem typu *enum* (Ukázka kódu 49), uvnitř které byly definovány 2 lokace – *Event* a *Offer*.

```
enum class TabItem(val icon: ImageVector, val title: String) {
    Event(Icons.Default.Event, "Event"),
    Offer(Icons.Default.LocalOffer, "Offer")
}
```

Ukázka kódu 49: Enum třída TabItem
(zdroj: vlastní zpracování)

Dále byla vytvořena composable funkce *Tabs*, která obsahuje funkci **TabRow**, jež poskytuje v základu Jetpack Compose a která slouží k vykreslení horní lišty. Uvnitř této funkce je nutné specifikovat jednotlivé lokace, které bude lišta obsahovat, což je podobné jako u spodní navigační lišty – pro každou položku enum třídy *TabItem* se pomocí cyklu *forEach* vytvoří composable funkce *LeadingIconTab*, která představuje lokaci na horní liště.

```
TabItem.values().forEachIndexed { index, tabItem ->
    LeadingIconTab(
        selected = index == selectedTabIndex,
        onClick = { onSelectedTab(tabItem) },
        text = { Text(text = tabItem.title) },…}
```

Ukázka kódu 50: Vytvoření lokací pro horní lištu
(zdroj: vlastní zpracování)

Nakonec byla vytvořena obrazovka (composable funkce) **PostScreen**, která slouží k držení horní lišty. Uvnitř této funkce byla vyvolána funkce Scaffold, která umožní správné umístění horní lišty. V tělu této funkce poté byla vyvolána composable funkce HorizontalPager (Ukázka kódu 51), jež umožňuje posouvání mezi částmi hlavní obrazovky s příspěvky. První část reprezentuje vnořená obrazovka *EventScreen,* která slouží k zobrazování příspěvků typu *Event* a druhou část reprezentuje vnořená obrazovka *OfferScreen*, jež zobrazuje příspěvky typu *Offer*.

```
Scaffold( topBar = {
        Tabs(selectedTabIndex = pagerState.currentPage…){
    HorizontalPager(state = pagerState) { index ->
        when (index) {
            0 -> EventScreen(navController,…
```

Ukázka kódu 51: Volání horní lišty v obrazovce PostScreen
(zdroj: vlastní zpracování)

Obrázek 29: Horní lišta aplikace Kolejka
(zdroj: vlastní zpracování)

4.2.2.4 Implementace notifikační obrazovky

Vytvořením horní lišty byla hlavní obrazovka s příspěvky dokončena a bylo možné začít implementovat **notifikační obrazovku**. Notifikační obrazovka se skládá ze seznamu notifikací (seznam vytvořen opět pomocí LazyColumn), které jsou představovány composable funkcí *NotificationComposable*. Tělo této composable funkce je opět vytvořeno pomocí composable funkce Card, která ve svém těle obsahuje komponenty umožňující zobrazení informací o notifikaci. Data pro jednotlivé notifikace jsou reprezentovány datovou třídou *Notification* (Ukázka kódu 52), která obsahuje následující parametry:

```
data class Notification(
        val parentId: String,
        val username: String,
        val notificationType: NotificationType,
        val formattedTime: String,
)
```

Ukázka kódu 52: Datová třída Notification
(zdroj: vlastní zpracování)

První parametrem je *parentId*, který představuje ID příspěvku, na který notifikace odkazuje. Dalším parametrem je *username*, který slouží k zobrazení jména uživatele, který notifikaci vytvořil. Třetím parametrem je *notificationType*, který určuje, o jaký typ notifikace se jedná. V rámci aplikace Kolejka existují tři typy notifikací:

- notifikace, která oznámí připojení k události

- notifikace, která oznámí, že někdo chce vytvořenou nabídku
- notifikace, že někdo přidal komentář k příspěvku

Parametr *notificationType* je parametr typu *NotificationType*, což je třída typu sealed a má následující podobu (Ukázka kódu 53):

```
sealed class NotificationType(val type: Int) {
    object JoinedEvent: NotificationType(0)
    object WantsOffer: NotificationType(1)
    object CommentedOn: NotificationType(2)
}
```

Ukázka kódu 53: Sealed třída NotificationType
(zdroj: vlastní zpracování)

Každý z objektů třídy NotificationType má svůj identifikátor *type*, který typ notifikace reprezentuje pomocí celého čísla. Identifikátor type byl přidán z důvodů, že databáze nedokáže zpracovat třídu *NotificationType*, tudíž se v databázi typ notifikace ukládá jako celé číslo. Posledním parametrem třídy *Notification* je poté *formattedTime*, který představuje čas, kdy byla notifikace vytvořena. Pro přiblížení využití parametru *notificationType* následuje popis použití (Ukázka kódu 54):

```
val fillerText = when (notification?.notificationType) {
    is NotificationType.JoinedEvent -> stringResource(id =
R.string.joined)
    is NotificationType.WantsOffer -> stringResource(id = R.string.wants)
    is NotificationType.CommentedOn -> stringResource(id =
R.string.commented_on)}

val notificationText = when (notification?.notificationType) {
    is NotificationType.JoinedEvent -> stringResource(id =
R.string.your_event)
    is NotificationType.WantsOffer -> stringResource(id =
R.string.your_offer)
    is NotificationType.CommentedOn -> stringResource(id =
R.string.your_post)
}
```

Ukázka kódu 54: Využití parametru notificationType
(zdroj: vlastní zpracování)

Funkce *NotificationComposable* přijímá jako argument třídu *Notification* a zobrazuje informace o notifikaci pomocí „klikatelných“ a klasických textů. Na začátku funkce *NotificationComposable* jsou poté definovány dvě proměnné.

První z proměnných je *fillerText*, která na základě typu notifikace nabývá určitých textových hodnot (např. pokud se jedná o notifikační typ *CommentedOn*, do proměnné se uloží string hodnota „*commented on“*).

Druhou proměnnou je *notificationText*, která podobně jako *fillerText* nabývá hodnot v závislosti na notifikačním typu (např. pokud se opět jedná o notifikační typ *CommentedOn*, do proměnné se uloží string hodnota *„your post“*). Tyto textové proměnné se poté využívají v composable funkcích Text, které jsou součástí funkce *NotificationComposable*.

Obrázek 30: Composable funkce NotificationComposable
(zdroj: vlastní zpracování)

4.2.2.5 Implementace obrazovky profilu

Poslední obrazovkou, která je lokací spodní navigační lišty, je **obrazovka profilu**. Dle návrhu se tato obrazovka skládá ze dvou částí: prapor uživatele a seznam příspěvků, které uživatel přidal/ke kterým se přidal. Nejprve byl vytvořen uživatelský prapor. Uživatelský prapor zobrazuje informace o přihlášeném uživateli, která jsou reprezentována datovou třídou User (Ukázka kódu 55):

```
data class User(
        val userId: String? = "",
        var username: String,
        val profilePictureUrl: String? = "",
        var bannerR: Float = 255f,
        var bannerG: Float = 255f,
        var bannerB: Float = 255f,
)
```

Ukázka kódu 55: Datová třída User
(zdroj: vlastní zpracování)

Parametr *userId* představuje identifikátor uživatele, *username* uživatelské jméno, *profilePictureUrl* odkaz na profilovou fotku uživatele a další tři parametry určují barvu pozadí uživatelského praporu.

Samotný prapor nese v kódu aplikace název *ProfileBannerComposable* a jedná se o composable funkci, jejíž tělo tvoří composable funkce Card. V těle funkce Card jsou dále composable funkce Column, Text, Image a Icon. Composable funkce Icon je využita dvakrát – jedna ikona reprezentuje tlačítko pro editaci profilu a druhá tlačítko pro odhlášení. Možnost kliknutí na ikonu byla umožněna díky modifikátoru *clickable* (lze také využít composable funkci IconButton).

Kliknutím na editační ikonu se dle návrhu otevře **dialog pro editaci profilu** – pro vytváření dialogových oken jsou v Jetpack Compose poskytnuty dvě vestavěné composable funkce – **Dialog** a **AlertDialog**. Zatímco pomocí funkce Dialog lze vytvořit plně přizpůsobitelné dialogové okno, pro implementaci funkce AlertDialog je nutné zachovat jistá pravidla – mezi tato pravidla patří např. nutnost, aby implementace AlertDialogu obsahovala potvrzovací a zamítací tlačítko. Pro vytvoření editačního dialogu byla využita funkce Dialog.

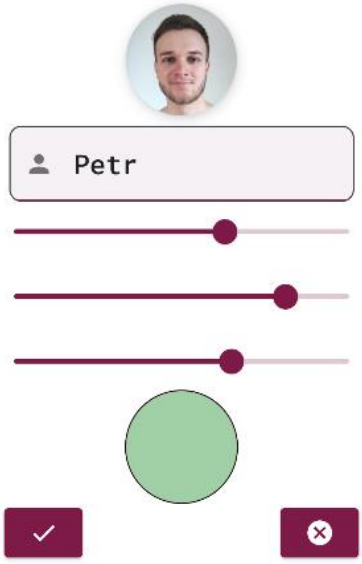
Uvnitř dialogu je nejprve composable funkce Image, která zobrazuje profilovou fotku uživatele. Tato composable funkce je navíc „klikatelná“ a kliknutím se otevře nabídka pro výběr fotky, což je funkcionalita, která byla řešena při fázi propojení backendu a aplikace Kolejka, tudíž bude probrána později v této práci. Dalším UI prvkem je composable funkce StandardTextField, která zobrazuje uživatelské jméno a umožňuje jej přepsat. Po funkci StandardTextField následují tři volání composable funkce Slider (posuvník), která je součástí Jetpack Compose a která umožňuje změnit barvu uživatelského praporu. Barva je složena ze tří barev – červené, zelené a modré (RGB) a každý Slider slouží ke změně hodnoty (0-255) jedné z těchto barev.

```
Slider(
    value = editProfileState.bannerG,
    onValueChange = { viewModel.onEvent(EditProfileEvent.ChangedG(it)) },
    valueRange = 0f..255f
)
```

Ukázka kódu 56: Composable funkce Slider
(zdroj: vlastní zpracování)

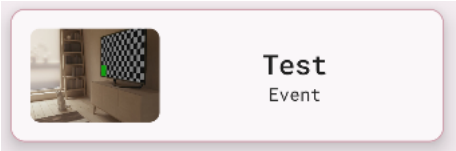
V ukázce kódu (Ukázka kódu 56) je předvedeno, jak composable funkce Slider funguje. Hlavním parametrem této funkce je *valueRange*, pomocí kterého lze specifikovat rozsah čísel, který daný Slider používá. Slidery, které aplikace Kolejka využívá ke změně barev, mají stanovený rozsah od 0 do 255 - hodnota 255 značí maximální intenzitu dané barvy a hodnota 0 znamená, že barva není zastoupena. Dalšími parametry funkce Slider jsou *value* a *onValueChange*. Parametru *value* je poskytnut číselný argument (v uvedeném případě musí mít argument hodnotu 0–255), podle kterého se stanoví pozice posuvníku na své ose. Parametru *onValueChange* je poté poskytnuta funkce, pomocí které lze změnit hodnotu posuvníku (a měnit tak pozici posuvníku na své ose).

Editační dialog následně obsahuje composable funkci Canvas, která slouží k zobrazení barvy uživatelského praporu (a zároveň reaguje na změny hodnot posuvníků). Uvnitř funkce Canvas je využita funkce *drawCircle*. Posledními dvěma UI prvky, které editační dialog obsahuje, jsou tlačítka k potvrzení, či zamítnutí úprav.



Obrázek 31: Dialog k editaci profilu
(zdroj: vlastní zpracování)

Druhou částí obrazovky profilu je seznam příspěvků, ke kterým se uživatel přidal/které uživatel vytvořil. Příspěvky se na obrazovce profilu zobrazují v seznamu pomocí funkce LazyColumn a pro vykreslení jednoho prvku seznamu je využita vytvořená composable funkce PostStrip (Obrázek 32). Tato funkce zobrazuje tři informace o příspěvku: fotku příspěvku, název příspěvku a typ příspěvku.



Obrázek 32: Composable funkce PostStrip
(zdroj: vlastní zpracování)

4.2.2.6 Implementace obrazovky k přidání příspěvku

Dokončením implementace obrazovky profilu zbývalo zkonstruovat dvě poslední obrazovky aplikace. První ze dvou zbývajících obrazovek, která byla vytvořena, byla **obrazovka k přidání příspěvku**.

Tato obrazovka nejprve obsahuje „klikatelný“ box, který po kliknutí otevře nabídku pro volbu obrázku příspěvku. Po boxu následují dvě composable funkce **RadioButton** (tlačítka), které jsou součástí Jetpack Compose. Těmito tlačítky může uživatel zvolit jaký typ příspěvku bude přidávat – *Event* či *Offer*. Pod tlačítky typu RadioButton se poté nachází 3 textová pole typu *StandardTextField*. Do prvního textového pole se vyplňuje název příspěvku, do druhého pole popis příspěvku a do třetího se napíše lokace události / nabídky (resp. kam si mají zájemci pro nabízený předmět přijít).

Při přidávání příspěvku typu *Offer* se na obrazovce nakonec objevuje poslední textové pole (opět typu *StandardTextfield*), pomocí kterého lze stanovit limit příspěvku. Při přidávání příspěvku typu *Event* je obrazovka rozšířena dále rozšířena o „klikatelný“ text, který po kliknutí otevře dialog, uvnitř kterého lze zvolit datum události. Dialog je composable funkce typu AlertDialog a ve svém těle obsahuje kalendář, který je vytvořen s pomocí dosud nezmíněné composable funkce **AndroidView** a který slouží ke zvolení data události. Vzhledem k tomu, že Jetpack Compose je nový framework, tak ve svém základu neobsahuje všechny UI prvky, které může vývojář Android aplikací požadovat. Jednou z UI komponent, kterou Jetpack Compose neobsahuje, je právě kalendářová komponenta. Autor práce měl tudíž na výběr dvě možnosti, jak kalendář v aplikaci implementovat:

- Vytvořit vlastní composable funkci, která by poskytovala všechny nutné funkce – zobrazení možných dat, přepínání mezi měsíci a možnost zvolení data
- Využít composable funkci AndroidView, pomocí které lze využít UI komponenty, které poskytuje XML způsob tvorby uživatelských rozhraní Android aplikací

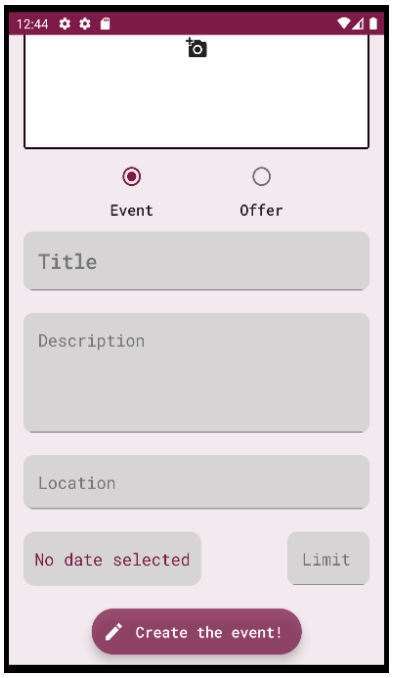
V práci byl zvolena druhá možnost implementace. Využití AndroidView v kódu vypadá následovně (Ukázka kódu 57):

```
AndroidView(
{CalendarView(android.view.ContextThemeWrapper()
    ).apply {}
update = { views ->
    views.setOnDateChangeListener { _, year, month, dayOfMonth ->
```

Ukázka kódu 57: Použití composable funkce AndroidView
(zdroj: vlastní zpracování)

Uvnitř funkce AndroidView je nejprve specifikována XML UI komponenta (View komponenta), která bude využita, což je v případě aplikace Kolejka komponenta **CalendarView**. Komponenta CalendarView poskytuje všechny funkce, které byly autorem požadovány – zobrazení možných dat, přepínání mezi měsíci a možnost zvolení data. Dále je ve funkci AndroidView uveden modifikátor, který slouží k specifikaci velikosti UI komponenty, a nakonec je ve funkci obsažen parametr update, který slouží k aktualizaci (překreslení) komponenty po změně stavu. V uvedeném příkladu se CalendarView komponenta aktualizuje při volbě jiného data. Kromě použití AndroidView v Jetpack Compose UI lze také používat composable funkce v XML vývoji UI, a to sice v rámci tzv. **ComposeView** komponent.

Na obrazovce nakonec nesmí chybět tlačítko k přidání příspěvku.



Obrázek 33: Obrazovka k přidávání příspěvků
(zdroj: vlastní zpracování)

K tomu, aby se dalo do obrazovky k přidání příspěvku navigovat, bylo třeba dle návrhu vytvořit na hlavní obrazovce s příspěvky plovoucí navigační tlačítko. Toto tlačítko bylo vytvořeno pomocí composable funkce **ExtendedFloatingActionButton**, která je poskytnuta v základu Jetpack Compose. Toto tlačítko bylo zabaleno do vytvořené composable funkce *FloatingAddPostButton*, která má parametr showButton (typ Boolean), jenž řídí viditelnost tlačítka.

```
@Composable
fun FloatingAddPostButton(
    showButton: Boolean,
    …
) {
    if(showButton) {
        ExtendedFloatingActionButton(
            text = {
                Text(text = buttonText, style =
MaterialTheme.typography.subtitle2)
            } …
    }
}
```

Ukázka kódu 58: Composable funkce FloatingAddPostButton
(zdroj: vlastní zpracování)

Tato composable funkce je následně volána z composable funkce (obrazovky) AppHolder. Stejně jako u spodní navigační lišty je umístění tlačítka *FloatingAddPostButton* řízeno composable funkcí Scaffold. Pro limitaci zobrazení tlačítka jen na jedné z obrazovek (obrazovka pro přidání příspěvku) je aplikována stejná logika, jako u spodní navigační lišty – tlačítko se zobrazuje jen pro obrazovky, které jsou obsaženy v seznamu (Ukázka kódu 59):

```
val screensWithAddPostButton = listOf(
    Screen.PostScreen.route)
Scaffold(
    bottomBar = {
        BottomNavigationBar(
            navController = navController,…)},
    floatingActionButton = {
        FloatingAddPostButton(
            showButton = navBackStackEntry?.destination?.route in
screensWithAddPostButton,…)))
```

Ukázka kódu 59: Řízení zobrazení tlačítka FloatingAddPostButton (zdroj: vlastní zpracování)

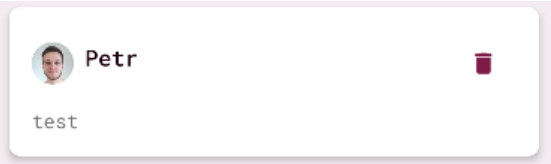
Obrázek 34: Grafická podoba composable funkce FloatingAddPostButton (zdroj: vlastní zpracování)

4.2.2.7 Implementace obrazovky s detailem příspěvku

Poslední vytvářenou obrazovkou byla **obrazovka s detailem příspěvku**. Na této obrazovce jsou nejprve zobrazeny detaily o příspěvku, které jsou reprezentovány datovou třídou Post – fotka příspěvku, název, popis, lokace, limit příspěvku, volná místa příspěvku, fotka uživatele, který fotku přidal a v případě příspěvku typu *Event* také datum. Dále je na obrazovce zobrazena informace o volných místech u příspěvku, která je uvedena ve formátu *<volná místa> / <limit příspěvku>* (např. *5/10*). Ve stejné úrovni informace o volných místech je na druhé straně obrazovky dále tlačítko, které umožňuje přidání k příspěvku. Pokud detail příspěvku otevře uživatel, který jej přidal, místo tlačítka pro přidání má k dispozici tlačítko pro smazání příspěvku.

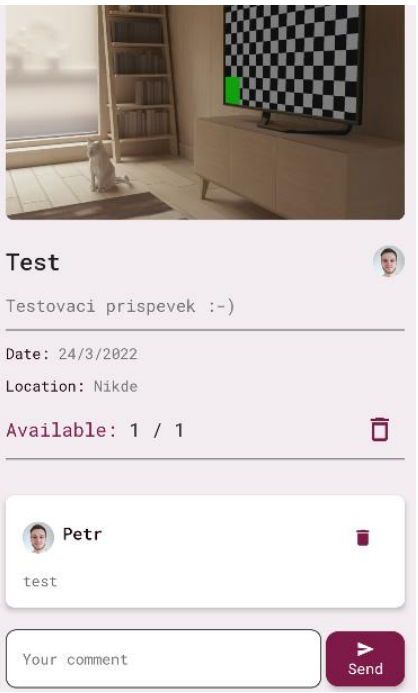
Pokud se uživatel přidá k příspěvku, zobrazí se mu také možnost příspěvek komentovat. Komentář v UI představuje composable funkce *CommentComposable*. Tělo této composable

funkce tvoří funkce Card, která má ve svém těle composable funkci Image, jež zobrazuje fotku uživatele, jenž komentář přidal, composable funkci Text, která zobrazuje jméno komentujícího uživatele, a nakonec druhou funkci Text, která zobrazuje text komentáře. Uživatel, který příspěvek vytvořil, má také možnost svůj komentář smazat pomocí tlačítka pro smazání. Komentáře se zobrazují v seznamu pomocí composable funkce LazyColumn.



Obrázek 35: Grafická podoba composable funkce CommentComposable (zdroj: vlastní zpracování)

Pro napsání komentáře je na spodku obrazovky umístěno textové pole typu *StandardTextField* a vytvořená composable funkce *SendCommentComposable*, která má podobu tlačítka s ikonou a textem, vertikálně poskládanými pod sebe.



Obrázek 36: Obrazovka s detailem příspěvku (zdroj: vlastní zpracování)

Dokončením obrazovky s detailem příspěvku (Obrázek 36) byla dokončena fáze návrhu UI a implementace. Další fází vývoje bylo vytvoření serverové aplikace neboli backendu aplikace.

4.3 Vývoj serverové aplikace

Jak již bylo naznačeno v předchozí kapitole, aplikační vrstvu aplikace Kolejka tvoří serverová aplikace vybudována pomocí frameworku **KTor**, který je postaven stejně jako Jetpack Compose na programovacím jazyku Kotlin. Komunikace mezi aplikací Kolejka a serverovou KTor aplikací probíhá pomocí aplikačního rozhraní, které je vybudováno podle architektury REST.

KTor aplikace lze programovat ve vývojářském prostředí **IntelliJ IDEA**, které vyvíjí firma Jetbrains. KTor projekt ovšem nelze vytvořit v neplacené verzi programu IntelliJ IDEA, ale je třeba využít webové rozhraní s URL *www.start.ktor.io*. V tomto rozhraní lze poté jednoduše zvolit všechny parametry projektu, včetně tzv. pluginů, což jsou dodatečné funkcionality, které projekt bude využívat. Mezi pluginy patří např. autentikační pluginy, serializační pluginy, či websockets. Po přidání pluginů se v projektu automaticky vygeneruje kód, který umožní funkce pluginů využívat.

4.3.1 Nastavení serverové aplikace

Po vytvoření čistého KTor projektu bylo nejprve třeba vytvořit jednotlivé datové třídy, které aplikace Kolejka využívá. Datové třídy v KTor projektu jsou stejné jako v aplikaci Kolejka – jedná se o třídy *User* (uživatel), *Post* (příspěvek), *Comment* (komentář) a *Notification* (notifikace). Tyto objekty se ukládají do databáze, načítají se z ní a slouží k zobrazení požadovaných informací v mobilní aplikaci.

Dalším krokem k vytvoření serverové aplikace bylo propojení aplikace s datovou vrstvou, kterou reprezentuje databáze MongoDB Atlas. MongoDB Atlas je tzv. **DBaaS**, což je zkratka termínu „DataBase as a Service“, tedy „databáze jako služba“. Jedná se o cloudovou verzi databáze MongoDB, která umožňuje přístup k vytvořené databázi přes internet, bez nutnosti hostingu databáze na vlastním serveru. Platforma MongoDB Atlas umožňuje vytvoření bezplatného účtu a omezené cloudové databáze – mezi omezení patří např. velikost úložiště. V bezplatné verzi databáze je poskytnuto 512 MB volného prostoru, což je ovšem pro základní testování aplikace více než dostačující (do databáze se ukládají pouze textové informace, které zabírají velmi málo místa).

Pro propojení MongoDB Atlas databáze a KTor projektu bylo nejprve nutné vygenerovat odkaz na databázi. To lze učinit pomocí MongoDB Atlas velmi snadno, a to sice kliknutím

na tlačítko *Connect* v rozhraní Atlasu a zkopírováním poskytnutého odkazu. V odkazu je ovšem třeba doplnit některé informace, mezi něž patří heslo k účtu, který databázi spravuje a název požadované databáze. V KTor projektu je poté nutné vytvořit proměnnou prostředí, do které se uloží odkaz na databázi. Pomocí této proměnné lze poté přistupovat k databázi.

Následným krokem k prvotnímu nastavení serverové aplikace bylo přidání knihoven **Koin** a **KMongo** do projektu. Přidávání knihoven funguje stejně, jako ve vývojovém prostředí Android Studio - přidáním jednotlivých závislostí do souboru *build.gradle*. Po „začlenění" knihoven do projektu byla také vytvořena první injektáž závislosti pomocí knihovny Koin. Pro využití injektáže bylo nejprve vytvořit tzv. **modul**, což je funkce, ve které lze inicializovat komponenty, které budou vkládány jako závislosti do jiných tříd (Ukázka kódu 60). Jelikož jednou z takových komponent je i databázový objekt, uvnitř modulu byla vytvořena jeho instance. Instance databázového objektu je dále používána jako závislost pro další třídy. Instance databázového objektu byla inicializována jako tzv. „singleton", tudíž se za běhu serverové aplikace vytvoří pouze jedna, unikátní instance tohoto objektu.

```
val mainModule = module {
    single {
        val client =
KMongo.createClient(System.getenv("MONGO_URI")).coroutine
        client.getDatabase(DATABASE_NAME)
    }
}
```

Ukázka kódu 60: Funkce modul k injektáži závislostí
(zdroj: vlastní zpracování)

Tímto byla prvotní konfigurace projektu dokončena a dále již byla vyvíjena business logika celé serverové aplikace. Business logika serverové aplikace byla založena na schématu *Repository – RepositoryImpl – Service – Route*. Toto schéma specifikuje, jak se pracuje s jednotlivými datovými třídami.

Na příklad pro datovou třídu User byly vytvořeny následující třídy:

- *UserRepository* – třída typu Interface, ve které jsou definovány jednotlivé operace, které je třeba provádět s datovou třídou User. Ve třídě Interface se definují abstraktní funkce, které může další třída implementovat.

- *UserRepositoryImpl* – třída, která implementuje funkce z UserRepository a komunikuje přímo s databází. Díky využití interface třídy UserRepository lze poté implementovat další „verze“ UserRepositoryImpl, které budou např. sloužit k testování funkcí na testovací databázi
- *UserService* – třída, která slouží jako doplňující vrstva k UserRepositoryImpl
- *UserRoutes* – soubor, ve kterém jsou definovány jednotlivé uživatelské endpointy aplikačního rozhraní, které je vytvořeno podle architektury REST (některé uživatelské operace jsou řešeny v souboru AuthRoutes, který taky obsahuje endpointy)

4.3.2 Registrace nového uživatele

Registrace nového uživatele se řídí schématem, které bylo specifikováno v předchozí podkapitole. Nejprve byla vytvořena funkce *createUser* v souboru UserRepository (Ukázka kódu 61):

```
interface UserRepository {
    suspend fun createUser(user: User)}
```

Ukázka kódu 61: Funkce createUser v UserRepository
(zdroj: vlastní zpracování)

Funkce *createUser* má jeden parametr, kterým je *user* (datového typu *User*). Tuto funkci dále implementuje třída *UserRepositoryImpl* (Ukázka kódu 62):

```
class UserRepositoryImpl(
    db: CoroutineDatabase
) : UserRepository {
    private val users = db.getCollection<User>()

    override suspend fun createUser(user: User) {
        users.insertOne(user)}}
```

Ukázka kódu 62: Implementace funkce createUser v UserRepositoryImpl
(zdroj: vlastní zpracování)

Třídě *UserRepositoryImpl* je nejprve injektován databázový objekt *db* typu CoroutineDatabase. Uvnitř třídy je poté vytvořena proměnná *users*, která odkazuje

na kolekci objektů typu *User* v databázi. **Kolekce** je v MongoDB skupina objektů určitého typu (ekvivalent tabulky v SQL databázi). Ve třídě je dále implementována funkce *createUser* a v jejím těle je specifikováno, že použitím této metody se do kolekce *users* vloží nový objekt typu *User* (*insertOne(user)*).

Ve třídě *UserService* je dále využita implementace funkce *createUser* z *UserRepositoryImpl*, která je vyvolána z funkce stejného jména, která ovšem jako argument přijímá instanci třídy *RegisterAccountRequest* (Ukázka kódu 63), jež představuje tělo HTTP požadavku, jenž uživatel pošle z aplikace Kolejka při pokusu o registraci uživatele.

```
data class RegisterAccountRequest(
    val email: String,
    val username: String,
    val password: String
)
```

Ukázka kódu 63: Třída RegisterAccountRequest
(zdroj: vlastní zpracování)

V těle funkce *createUser* ve třídě *UserService* se poté vytvoří instance třídy *User*, která je uložena do databáze.

```
suspend fun createUser(request: RegisterAccountRequest) {
    userRepository.createUser(
        User(
            email = request.email,
            username = request.username,
            password = getHashWithSalt(request.password),
        ))}
```

Ukázka kódu 64: Vytvoření instance třídy User
(zdroj: vlastní zpracování)

Instance třídy User přijímá argumenty pro parametry *email*, *username*, *password* a *profilePictureURL* (Ukázka kódu 64). Argumenty pro *email* a *username* jsou poskytnuty vstupním argumentem funkce createUser *request*. Argument pro parametr *password* pochází také z argumentu *request*, je ovšem zašifrován pomocí hashovací funkce, která bude uvedena později v této práci. Třída User má poté ještě 4 další parametry: *profilePictureURL*, který

obsahuje odkaz na profilovou fotku uživatele a *bannerR*, *bannerG*, *bannerB*, které určují barvu uživatelského praporu. Těmto parametrům je ovšem nastavena výchozí hodnota (Ukázka kódu 65), proto ji není třeba specifikovat při vytvoření instance třídy.

```
val profilePictureURL: String = Constants.DEFAULT_AVATAR_URL,
val bannerR: Float = 255f, val bannerG: Float = 255, …
```

Ukázka kódu 65: Parametry třídy User s výchozí hodnotou
(zdroj: vlastní zpracování)

Výchozím argumentem pro *profilePictureURL* je text s odkazem na základní profilovou fotku, kterou mají všichni nově zaregistrovaní uživatelé. Výchozím argumentem pro všechny parametry z trojice *bannerR*, *bannerG* a *bannerB* je hodnota 255 - každý nově zaregistrovaný uživatel má tak hodnotu barvy praporu (255,255,255), která reprezentuje bílou barvu.

Nakonec se v souboru *AuthRoutes* vytvoří rozšiřovací funkce třídy Route, kterou poskytuje framework KTor a která slouží k definování endpointu aplikačního rozhraní, se kterým bude komunikovat mobilní aplikace Kolejka. Dále se ve funkci určí typ HTTP požadavku, v případě vytváření uživatele se jedná o metodu *POST*. V této funkci také dojde k validaci, zda např. není tělo uživatelské požadavku prázdné a v případě, že je vše v pořádku, dojde k vytvoření nového uživatele v databázi.

```
fun Route.registerUser(
    userService: UserService
) {

    route(<CESTA K VYTVOŘENÍ POŽADAVKU>) {
        post {
            val request = call.receiveOrNull<RegisterAccountRequest>() ?:
kotlin.run {
                call.respond(HttpStatusCode.BadRequest)
                return@post
            }
        }}
```

Ukázka kódu 66: Definování endpointu k registraci uživatele
(zdroj: vlastní zpracování)

V příkladu (Ukázka kódu 66) lze vidět, že se definuje *route*, tedy cesta, pomocí které lze přistoupit k registrační funkci serverové aplikace. Dále je definováno, že se jedná o požadavek typu POST, což KTor umožňuje voláním funkce s názvem *post*. V těle této funkce poté dochází k validaci požadavku apod.

Tímto stručně popsaným příkladem bylo popsáno, jak dochází k vytváření nových uživatelů a jejich zapsání do databáze pomocí serverové aplikace. S ostatními datovými třídami je poté práce často velmi podobná, a proto nebude veškerá funkcionalita serverové aplikace popsána dopodrobna. V následující části kapitoly budou ovšem popsány esenciální funkce serverové aplikace, u kterých autor usoudil, že by měly být zmíněny.

4.3.3 Přihlášení uživatele

Poté, co si uživatel vytvoří účet, otevře se mu možnost přihlášení. Komunikace mezi klientem a serverem je opět založena na metodice *request-response*, tedy požadavek-odpověď. Nejprve byly ve třídě *UserRepository* vytvořeny abstraktní funkce *getUserByEmail* a *doesPasswordForUserMatch.* Tyto funkce byly následně implementovány ve třídě *UserRepositoryImpl* následujícím způsobem (Ukázka kódu 67):

```
override suspend fun getUserByEmail(email: String): User? {}
override suspend fun doesPasswordForUserMatch(email: String, password:
String): Boolean {val user = getUserByEmail(email) ?: return false
    return checkHashForPassword(password, user.password)}
```

Ukázka kódu 67: Implementace funkcí getUserByEmail a doesPasswordForUserMatch (zdroj: vlastní zpracování)

Funkce *getUserByEmail* přijímá argument typu String pro parametr *email*. Pokud je v databázové kolekci *users* nalezen uživatel s emailem, který je stejný jako poskytnutý argument funkce, funkce navrátí objekt třídy User s informacemi o požadovaném uživateli. Pokud uživatel se stejným emailem nalezen není, funkce vrací hodnotu *null*.

Funkce *doesPasswordForUserMatch* poté přijímá argumenty typu String pro parametry email a password. Uvnitř funkce pak dojde k vyvolání funkce *getUserByEmail* a pokud tato funkce nenalezne uživatele, funkce *doesPasswordForUserMatch* vrací hodnotu *false*. V případě, že je uživatel nalezen, funkce vrací hodnotu *true*, pokud se hash poskytnutého

hesla rovná hashi, který má uživatel uložen v databázi. Pokud se nerovnají, funkce vrací hodnotu *false*.

Tyto funkce nevyžadují doplňující informace, tudíž se ve třídě *UserService* vytvoří funkce se stejným jménem, které pouze volají funkce z *UserRepository*.

Nakonec je v souboru *AuthRoutes* vytvořena rozšiřující funkce třídy Route, která nese název *loginUser*. Aby mohla být tato funkce implementována správně, byla nejprve vytvořena třída *LoginAccountRequest* (Ukázka kódu 68), která představuje tělo uživatelského požadavku na server:

```
data class LoginAccountRequest(
    val email: String,
    val password: String
)
```

Ukázka kódu 68: Třída LoginAccountRequest
(zdroj: vlastní zpracování)

V tělu funkce *loginUser* je poté specifikována cesta k přihlašovací funkcionalitě (route) a vyvolána funkce *post*, jelikož se jedná o požadavek typu POST. Uvnitř funkce post již poté dochází k přijetí požadavku a deserializaci JSON dat do třídy *LoginAccountRequest* (deserializace je umožněna díky zabudovanému pluginu Gson), validaci požadavku (kontrola, zda nemá požadavek prázdná pole) a nakonec k ověření, zda poskytnuté heslo odpovídá heslu, které má uživatel uložené jako hash v databázi.

```
when (userService.validateLoginRequest(loginRequest)) {
    is ValidationEvent.EmptyFieldError -> {
        call.respond(BasicApiResponse<Unit> …
```

Ukázka kódu 69: Validace přihlašovacího požadavku
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 69) lze vidět způsob, jak probíhá kontrola, zda má požadavek prázdná pole. Kontrola je prováděna ve funkci *validateLoginRequest*, která je umístěna ve třídě *UserService*. Tato funkce přijímá argument typu *LoginAccountRequest* a vrací jeden z objektů sealed třídy *ValidationEvent*, kterými jsou *EmptyFieldError* nebo *Success*.

V příkladu lze vidět, že pokud funkce *validateLoginRequest* vrátí objekt *EmptyFieldError*, serverová aplikace vytvoří odpověď (response), kterou reprezentuje třída *BasicApiResponse*. Tato třída byla vytvořena autorem a vypadá následovně (Ukázka kódu 70):

```
data class BasicApiResponse<T>(
    val message: String? = null,
    val successful: Boolean,
    val data: T? = null)
```

Ukázka kódu 70: Třída BasicApiResponse
(zdroj: vlastní zpracování)

Tato třída představuje odpověď serveru a má parametr *message*, který může obsahovat libovolnou zprávu, která se zobrazí tvůrci požadavku, parametr *successful*, který značí, zda byl požadavek úspěšný či ne a parametr *data*, který může, ale nemusí obsahovat data spjata s odpovědí serveru. V uvedeném příkladu kontroly, zda má přihlašovací požadavek prázdná pole, serverová aplikaci při vrácení objektu *EmptyFieldError* odpoví instancí třídy *BasicApiResponse* s informací, že byl požadavek neúspěšný a se zprávou, že pole požadavku nesmí být prázdná. S odpovědí zároveň nejsou uživateli poslána žádná data.

V případě, že požadavek projde veškerou validací, dojde k přihlašovacímu mechanismu – serverovou aplikací je vygenerován tzv. JWT, což je zkratka pro Json Web Token a tento token je poslán uživateli zpět v datech odpovědi typu *BasicApiResponse*. Json Web Token je dle definice „*kompaktní a bezpečný prostředek pro výměnu informací, které mají být převedeny mezi dvěma stranami*“ (Jones a kol. 2015). Jedno z častých využití JWT je k autorizaci uživatele v určitém systému. V rámci serverové aplikace k aplikaci Kolejka je JWT používán k omezení přístupu ke zdrojům, které jsou uloženy v databázi, pouze pro přihlášené uživatele. KTor poskytuje podporu pro JWT a při vytváření nového KToru projektu lze do projektu zahrnout plugin, který se stará o konfiguraci JWT. Generování Json Web Tokenu v kódu vypadá následovně (Ukázka kódu 71):

```
val token =
JWT.create().withClaim("userId",user.id).withIssuer(jwtIssuer).withExpire
sAt(Date(System.currentTimeMillis() + expiresIn))…
```

Ukázka kódu 71: Generování JWT
(zdroj: vlastní zpracování)

V rámci JWT je nejprve třeba specifikovat tzv. **claims** – tvrzení, která obsahují informace o uživateli a další dodatečná data. První tvrzení, které je v tokenu vytvořen je tvrzení, které obsahuje ID uživatele, jenž se přihlašuje. Dalšími tvrzeními jsou poté tzv. registrovaná tvrzení – **issuer** (obsahuje informace vydavateli tokenu**), expiration** (obsahuje informace o délce platnosti tokenu) a **audience** (obsahuje informace o příjemci tokenu). Posledním krokem pro vytvoření tokenu je vytvoření digitálního podpisu. Podepsání tokenu slouží k ověření, že s daným tokenem nebylo manipulováno a že se nezměnil.

Vygenerovaný token může mít ve finále následující podobu:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiUGV0ciBIb3LDocSNZWsiLCJt
ZXNzYWdlIjoiWmRyYXbDrW0gdG9obywga2RvIHByw6F2xJsgxI10ZSBtb2rDrSBkaXBsb21vd
m91IHByw6FjaSEgOi0pIn0.5vOAbUmIAag2MRihnlc2_sY2TEWAgO_wJwwaJ0aDPok
```

Ukázka kódu 72: Json Web Token
(zdroj: vlastní zpracování)

První, červená část (**hlavička**) obsahuje informace o typu tokenu a použitém šifrovacím algoritmu, zakódované pomocí Base64. Druhá, zelená část (**náklad**) obsahuje tvrzení a také je zakódována pomocí Base64. V poslední, modré části (**podpis**) se poté vezme zakódovaná hlavička, náklad a tajný klíč, který reprezentuje jakákoliv kombinace znaků (v příkladu je použit jednoduchý tajný klíč „secret“, v reálném využití je pak ovšem vhodné použít komplexnější kombinaci znaků). Všechny tyto tři části se spojí dohromady a vytvoří se z nich hash pomocí algoritmu, který se specifikován v hlavičce.

Vygenerovaný token poté posílá klient v hlavičce HTTP požadavku. Ve chvíli, kdy aplikační server obdrží HTTP požadavek s tokenem v hlavičce, může si ověřit jeho platnost, neboť tajný klíč, se kterým byl token podepsán, je uložen na serveru. Server tak může vytvořit hash z hlavičky, nákladu a uloženého tajného klíče a porovnat jej s podpisem tokenu.

4.3.4 Přidávání příspěvků

Další důležitou funkcí aplikace Kolejka je přidávání příspěvků. Pro vytvoření této funkcionality v serverové aplikaci bylo nejprve potřeba vytvořit třídu *PostRepository*, která definuje databázové operace s příspěvky a *PostRepositoryImpl*, která operace implementuje. Pro vytvoření příspěvku byla ve třídě *PostRepository* nejprve vytvořena funkce *createPost* (Ukázka kódu 73):

```
interface PostRepository {
    suspend fun createPost(post: Post): Boolean
    …
}
```

Ukázka kódu 73: Funkce createPost v PostRepository
(zdroj: vlastní zpracování)

Tato funkce byla dále implementována ve třídě PostRepositoryImpl (Ukázka kódu 74):

```
override suspend fun createPost(post: Post): Boolean {
    return posts.insertOne(post).wasAcknowledged()
}
```

Ukázka kódu 74: Implementace funkce createPost
(zdroj: vlastní zpracování)

Lze vidět, že funkce přijímá jako argument instanci třídy *Post* a návratová hodnota této funkce je typu Boolean. Uvnitř funkce se jednoduše provede zápis do databáze (do kolekce *posts*) příkazem *insertOne*. Metoda *wasAcknowledged* následně vrací hodnotu *true* v případě, že byl zápis do databáze zaznamenán, nebo *false* v případě opačném.

Dále byla vytvořena třída *NewPostRequest* (Ukázka kódu 75), která reprezentuje uživatelský požadavek pro vytvoření příspěvku.

```
  data class NewPostRequest(
    val title: String,
    val description: String,
    val limit: Int,
    val type: Int,
    val date: String,…)
```

Ukázka kódu 75: Třída NewPostRequest
(zdroj: vlastní zpracování)

Tato třída reprezentuje data, která uživatel specifikuje při tvorbě příspěvku. V případě, že by pro požadavky byla použita samotná třída *Post*, uživatel by pro vytvoření příspěvku musel specifikovat argumenty i pro ostatní parametry této třídy, např. seznam členů, své uživatelské ID či URL své profilové fotky, což by nebylo možné.

Ve třídě PostService následně dochází k volání funkce *createPost* a samotnému vytvoření instance třídy *Post*, která se uloží do databáze (Ukázka kódu 76):

```
suspend fun  createNewPost(request: NewPostRequest, userId: String,
username: String, profilePictureUrl: String): Boolean{
    return postRepository.createPost(
        Post(
            userId = userId,
            title = request.title,…)}
```

Ukázka kódu 76: Volání funkce createPost v PostService
(zdroj: vlastní zpracování)

Funkce *createNewPost* v třídě *PostService* přijímá jako argumenty instanci třídy *NewPostRequest* (uživatelský požadavek) a poté *userId, username* a *profilePictureUrl* jako typ String. Uvnitř funkce poté dochází k volání funkce *createPost* z třídy *PostRepository* a dojde k vytvoření příspěvku. Některá data pro nový příspěvek jsou poskytnuta skrze uživatelský požadavek, zbytek dat zajistí serverová aplikace v dalším kroku, kterým je vytvoření endpointu aplikačního rozhraní.

K vytvoření endpointu byla ve vytvořeném souboru *PostRoutes* vybudována rozšiřující funkce třídy Route, která nese název *createNewPost*. Kromě specifikování cesty k přidání příspěvků a určení typu požadavku (v tomto případě se opět jedná o požadavek typu post) je veškerá logika přidávání příspěvku také zabalena do funkce *authenticate*, která umožní přidání příspěvku pouze pro autorizované uživatele. Autorizovanými uživateli jsou ti, kteří vyšlou požadavek s platným JWT v hlavičce HTTP požadavku.

```
authenticate {
    route(<CESTA K PŘIDÁNÍ PŘÍSPĚVKU>) {
        post {
            val request = call.receiveOrNull<NewPostRequest>() ?:
kotlin.run {
                call.respond(HttpStatusCode.BadRequest)
                return@post
            }}}
```

Ukázka kódu 77: Endpoint k přidávání příspěvků
(zdroj: vlastní zpracování)

Uvnitř funkce post (Ukázka kódu 77) nejprve dojde k přijetí uživatelského požadavku, který se uloží do proměnné *request* (v případě, že by měl požadavek hodnotu null, server odpoví chybovým kódem 400 - *Bad Request*).

```
val username = userService.getUsernameById(call.userId) ?: kotlin.run {
    call.respond(HttpStatusCode.NotFound)
    return@post
}
val profilePictureUrl = userService.getUserProfileUrl(call.userId) ?:
kotlin.run {
    call.respond(HttpStatusCode.NotFound)
    return@post
}
val newPost = postService.createNewPost(
    request = request,
    userId = call.userId,…)
```

Ukázka kódu 78: Tělo funkce post u přidávání příspěvků
(zdroj: vlastní zpracování)

Dále ve funkci (Ukázka kódu 78) dojde k deklarování proměnné *username*, která načte hodnotu pomocí třídy *UserService*, uvnitř které je funkce *getUsernameById*. Tato funkce vrací jméno uživatele podle poskytnutého ID. V ukázce kódu je této funkci poskytnut argument *call.userId*, pomocí kterého lze získat ID uživatele, které je uloženo v „claimu" jeho JWT. Tímto způsobem lze jednoduše získat ID uživatele, který se pokouší o přidání příspěvku. Podobně poté funguje funkce *getUserProfileUrl*, pomocí které se načte do proměnné *profilePictureUrl* URL k profilové fotce daného uživatele.

Pomocí proměnných *request*, *username* a *profilePictureUrl* lze pak volat funkci *createNewPost*, do které se dosadí proměnné jako argumenty funkce. Jelikož funkce *createNewPost* vrací hodnotu typu Boolean, lze poté snadno vyhodnotit výsledek přidávání příspěvku a poslat odpovídající odpověď (Ukázka kódu 79).

```
if(newPost){
    call.respond(
        HttpStatusCode.OK,
        BasicApiResponse<Unit>(
            successful = true))} else {…}
```

Ukázka kódu 79: Vyhodnocení výsledku přidávání příspěvku
(zdroj: vlastní zpracování)

4.3.5 Získávání příspěvků

K tomu, aby aplikace mohla načítat příspěvky bylo třeba tuto funkcionalitu vytvořit na aplikačním serveru. Funkcionalita načítání byla opět implementována pomocí schéma *Repository-RepositoryImpl-Service-Route*. Ve třídě PostRepository byla nejprve vytvořena funkce *getPostsByAll* s návratovým typem List <Post>, tedy seznam příspěvků. Tato funkce byla implementována ve třídě *PostRepositoryImpl* následujícím způsobem (Ukázka kódu 80):

```
 override suspend fun getPostsByAll(page: Int, pageSize: Int): List<Post>
{
    return posts.find(Post::available gt 0).skip(page *
pageSize).limit(pageSize).toList()
}
```

Ukázka kódu 80: Implementace funkce getPostsByAll
(zdroj: vlastní zpracování)

Funkce *getPostsByAll* přijímá argumenty pro parametry *page* a *pageSize*. Page přijímá hodnoty typu Int (integer – celé číslo) a specifikuje stránku příspěvků k načtení. PageSize také přijímá hodnoty typu Int a značí velikost jedné stránky (množství příspěvků, které jedna stránka obsahuje).

V tělu funkce se poté vrací seznam příspěvků, který odpovídá parametrům, které odpovídají databázovému dotazu. Nejprve jsou vyfiltrovány příspěvky, které mají hodnotu volných míst větší než 0 (*find (Post::available gt 0)*). Poté je „přeskočen“ počet příspěvků, který odpovídá násobku argumentů *page* * *pageSize* (nejprve je přeskočeno 0 příspěvků, potom množství, které odpovídá velikosti argumentu pageSize, poté pageSize * 2, …). Množství příspěvků je dále limitováno podle argumentu pageSize (*limit(pageSize)*) a převedeno na datový typ List (seznam).

Ve třídě *PostService* dále není s funkcí nijak manipulováno, a nakonec dochází k vytvoření požadovaného endpointu. Endpoint je vytvořen v souboru *PostRoutes*, uvnitř kterého je vybudována rozšiřující funkce třídy Route s názvem *getPostsByAll*. V tělu funkce se nejprve opět volá funkce *authenticate* k autorizování přístupu a je specifikována cesta k příspěvkům. Požadavek pro získání příspěvků je tentokrát typu GET (pro který KTor poskytuje příslušnou funkci).

Uvnitř funkce *get* jsou vytvořeny proměnné page a pageSize, do kterých se načtou hodnoty parametrů, které jsou klientem specifikovány při vytvoření požadavku (Ukázka kódu 81). Pokud není seznam s příspěvky prázdný, vrací se v těle odpovědi serveru.

```
val page = call.parameters[QueryParameters.PARAM_PAGE]?.toIntOrNull() ?:
0
val pageSize =
call.parameters[QueryParameters.PARAM_PAGE_SIZE]?.toIntOrNull() ?:
Constants.POSTS_PAGE_SIZE

    val allPosts = postService.getPostsByAll(page, pageSize)
    if(allPosts.isNotEmpty()){
        call.respond(
            HttpStatusCode.OK, allPosts
        )
    }
}
```

Ukázka kódu 81: Tělo funkce get u získávání příspěvků
(zdroj: vlastní zpracování)

4.3.6 Aktualizace uživatelských informací

K vytvoření funkcionality pro aktualizaci uživatelských informací byla nejprve vytvořena třída *UpdateUserRequest*, která představuje uživatelský požadavek o aktualizaci informací (Ukázka kódu 82):

```
data class UpdateUserRequest(
    val username: String,
    val bannerR: Float,
    val bannerG: Float,
    val bannerB: Float,
    val profilePictureURL: String?
)
```

Ukázka kódu 82: Třída UpdateUserRequest
(zdroj: vlastní zpracování)

Tato třída je poté využita uvnitř *UserRepository*, kde byla vytvořena funkce *updateUserInfo,* jež přijímá instanci třídy *UpdateUserRequest* jako argument, společně s ID uživatele. Tato funkce vrací hodnotu typu Boolean (Ukázka kódu 83):

```
suspend fun updateUserInfo(userId: String, updateProfileRequest:
UpdateUserRequest): Boolean
```

Ukázka kódu 83: Funkce updateUserInfo
(zdroj: vlastní zpracování)

Funkce *updateUserInfo* je dále implementována v třídě *UserRepositoryImpl*, kde dochází k samotné aktualizaci uživatelských dat (Ukázka kódu 84):

```
override suspend fun updateUserInfo(userId: String, updateProfileRequest:
UpdateUserRequest): Boolean {

    val user = getUserById(userId) ?: return false
    return users.updateOneById(
        id = user.id,
        update = User(
            email = user.email,
            username = updateProfileRequest.username,
            password = user.password,
            profilePictureURL = updateProfileRequest.profilePictureURL ?:
user.profilePictureURL
            …
        )
    ).wasAcknowledged()
}
```

Ukázka kódu 84: Implementace funkce updateUserInfo
(zdroj: vlastní zpracování)

Uvnitř funkce nejprve dojde verifikaci, zda uživatel s poskytnutým ID existuje (pokud ne, funkce vrací hodnotu *false*). Poté již dochází k aktualizaci dat uživatele pomocí databázového dotazu *updateOneById*, kterému je nejprve třeba specifikovat ID uživatele a poté způsob, jakým bude uživatel aktualizován. Při aktualizaci dojde ke změně celého zápisu uživatele v databázi za nový zápis. Vytvoří se instance třídy *User*, která nahradí stávajícího uživatele s tím, že některé informace zůstanou stejné – ID, email a heslo. Pomocí poskytnutého argumentu *updateProfileRequest* poté může uživatel změnit své uživatelské jméno, barvu uživatelského praporu a profilovou fotku (pomocí jiné funkce na serverové aplikaci poté může uživatel změnit i heslo).

Ve třídě *UserService* není s funkcí dále manipulováno. Nakonec je v souboru *UserRoutes* vytvořen endpoint, který umožňuje aktualizaci uživatelských informací. Endpoint je

reprezentován funkcí *updateUserInfo*. Uvnitř této funkce je opět vyvolána funkce *authenticate,* specifikovaná cesta ke změně uživatele a specializovanou funkcí specifikován typ požadavku – tím je v případě aktualizace uživatelských informací typ požadavku PUT, který by se měl v rámci architektury aplikačního rozhraní REST využívat k úpravám.

```
put {
    val request = call.receiveOrNull<UpdateUserRequest>() ?: kotlin.run {
        call.respond(BadRequest)
        return@put
    }
    val updateProfile = userService.updateUserInfo(
        userId = call.userId,
        updateUserRequest = request
    ) …
}
```

Ukázka kódu 85: Tělo funkce put u aktualizace uživatelských informací (zdroj: vlastní zpracování)

Ve funkci *put* (Ukázka kódu 85) byla nejprve vytvořena proměnná *request*, která načte data z požadavku, který je na endpoint aplikačního rozhraní odeslán (v případě, že by měl požadavek hodnotu *null*, server odpoví chybovým kódem 400 - *Bad Request* a ukončí funkci). Do proměnné *updateProfile* se poté načte výsledek funkce *updateUserInfo*, která je ve třídě *UserService*. Jako argument pro parametr *userId* je opět využito *call.userId* a pro parametr *updateUserRequest* se použije vytvořená proměnná *request*. Proměnná *updateProfile* obdrží výsledek funkce k aktualizaci uživatelských informací jako hodnotu typu Boolean. Díky tomu lze jednoduše snadno vyhodnotit výsledek aktualizace a poslat odpovídající odpověď.

```
if(updateProfile){
    postService.updatePostsProfilePic(call.userId)
    commentService.updateCommentInfo(call.userId)
    call.respond(OK,BasicApiResponse<Unit>(successful = true))
```

Ukázka kódu 86: Vyhodnocení výsledku aktualizace uživatelských informací (zdroj: vlastní zpracování)

Lze také vidět (Ukázka kódu 86), že pokud *updateProfile* nabude hodnoty *true*, uvnitř podmíněného výrazu se vykonají další funkce, např. funkce *updatePostsProfilePic*, která u všech příspěvků od uživatele, který požadavek vykonal, změní profilovou fotku, která se

v detailu příspěvku zobrazuje (v případě, že si uživatel změnil požadavkem svoji profilovou fotku).

Více funkcí serverové aplikace v rámci práce popisováno nebude, neboť je práce primárně zaměřena na vývoj Android aplikace. Mezi nepopsané funkce patří např:

- vytváření komentářů
- vymazání příspěvků
- vytvoření notifikací
- připojení uživatelů k příspěvku
- …

Všechny tyto funkcionality byly v rámci KTor serverové aplikace vyvíjeny podle schématu, které bylo uvedeno dříve v této podkapitole. Jejich implementaci si lze prohlédnout ve zdrojovém kódu serverové aplikace, která je přiložena k této diplomové práci.

Další podkapitola se bude zabývat nejdůležitější částí vývoje aplikace, a to sice propojením uživatelského rozhraní aplikace se serverovou aplikací.

4.4 Propojení serverové aplikace s uživatelským rozhraním

K propojení serverové aplikace a uživatelského rozhraní aplikace Kolejka byly esenciální dvě externí knihovny, které byly přidány do projektu aplikace: **Retrofit** a **Dagger-Hilt**. Retrofit slouží jako HTTP klient aplikace, díky kterému je umožněno posílat HTTP požadavky na serverovou aplikaci. Dagger-Hilt slouží k injektáži závislostí a usnadňuje využívání knihovny Retrofit, což bude později uvedeno na příkladu.

Základní schéma pro vytváření požadavků na serverovou aplikaci a načítání náležitých dat do UI je následující:

- API třída typu Interface, ve které se specifikují jednotlivé požadavky, které budou posílány serverové aplikaci
- Implementace předešlé třídy typu Interface
- Třída s názvem *Repository*, která je typu Interface a ve které jsou specifikovány jednotlivé funkce, které slouží ke komunikaci se serverovou aplikací
- Třída s názvem *RepositoryImpl*, která implementuje funkce z třídy Repository

- Třída s názvem *<Funkce>UseCase*, kterou je prováděno volání jednotlivých funkcí serverové aplikace pomocí implementace třídy Repository
- Třída typu *ViewModel*, která komunikuje s UseCase třídou a načítá data pro uživatelské rozhraní
- Data z ViewModelu jsou zobrazeny na již vytvořených obrazovkách aplikace

Toto schéma bude dále popsáno na příkladu, který reprezentuje registraci uživatele do aplikace.

4.4.1 Implementace registrace uživatele

Dle schématu byla nejprve vytvořena API třída typu Interface, která nese jméno *AuthApi*. Uvnitř této třídy byla definována funkce *registerAccount* (Ukázka kódu 87), která byla anotovaná textem *@Post (<API cesta>)*. Pomocí takových anotací jsou v rámci knihovny Retrofit určeny endpointy aplikačního rozhraní serverové aplikace, se kterou bude komunikováno.

```
interface AuthApi {
    @POST("<API cesta>")
    suspend fun registerAccount(@Body request: RegisterAccountRequest): )
    BasicApiResponse<Unit>
}
```

Ukázka kódu 87: Funkce registerAccount
(zdroj: vlastní zpracování)

Jak si lze povšimnout, v ukázce kódu funkce *registerAccount* přijímá parametr *request*, který je typu *RegisterAccountRequest*. Třída *RegisterAccountRequest* je stejná jako třída stejného jména, jež byla vytvořena v serverové aplikaci. Tento parametr je anotován pomocí *@Body*, díky čemuž knihovna Retrofit rozpozná, že se jedná o **tělo** HTTP požadavku. Funkce *registerAccount* má dále návratový typ *BasicApiResponse*, což je stejná třída, která byla vytvořena v serverové aplikaci.

Dále byla interface třída *AuthApi* implementována. K implementaci byla využita knihovna Dagger-Hilt k injektáži závislostí. Nejprve byla vytvořena třída s názvem *AuthModule*, která je typu Object. Tato třída funguje stejně, jako modul v serverové aplikaci – uvnitř této třídy se inicializují třídy, jejichž instance se využívají jako závislost v jiných třídách. V serverové

aplikaci k injektáži závislostí slouží knihovna Koin, kterou lze využít i při vytváření Android aplikací, autorovi diplomové práce však byla bližší práce s Dagger-Hilt, proto byla tato knihovna zvolena (Dagger-Hilt ovšem nelze použít v rámci frameworku KTor, proto serverová aplikace využívá Koin).

Třídě *AuthModule* byly dodány dvě anotace: *@Module*, která specifikuje, že je daná třída modulem a *@InstallIn(SingletonComponent:class)*, která specifikuje, že daný modul bude nainstalován v tzv. singleton komponentě, což značí, že instance modulu „žije“ od zapnutí aplikace po její ukončení.

Uvnitř třídy *AuthModule* poté byla vytvořena funkce *provideAuthApi* (Ukázka kódu 88), pomocí které je vytvořena instance třídy Retrofit.

```
@Module
@InstallIn(SingletonComponent::class)
object AuthModule {
    @Provides
    @Singleton
    fun provideAuthApi(client: OkHttpClient): AuthApi {
        return Retrofit.Builder()
            .baseUrl(BASE URL)
            .client(client)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(AuthApi::class.java)
    }
}
```

Ukázka kódu 88: Funkce provideAuthApi
(zdroj: vlastní zpracování)

Tato funkce je také doplněna o anotace. První anotací je *@Provides*, která specifikuje knihovně Hilt způsob jak poskytovat instance třídy *AuthApi*, která je implementována pomocí třídy Retrofit. Další anotací je *@Singleton*, která informuje Hilt o tom, že tato instance bude vytvořena pouze jednou. Uvnitř třídy je specifikováno, jak třídu *AuthApi* implementovat, a to sice pomocí knihovny Retrofit. Nejprve se specifikuje základní URL k serverové aplikaci, poskytne se HTTP klient (Retrofit využívá pro vytváření požadavků/přijímání odpovědí knihovnu **OkHttp**), specifikuje se knihovna pro serializaci/deserializaci dat (v případě aplikace Kolejka se jedná o knihovnu **Gson**, lze ovšem využít také např. Jackson či Kotlinx) a nakonec se metodou *create* vytvoří instance *AuthApi*.

Dalším krokem bylo vytvoření třídy *Repository* (repositář). K zachování čistého kódu byly pro práci s různými daty (datovými třídami) vytvořeny samostatné repositáře. Pro registraci uživatelů byla vytvořena třída *AuthRepository*. Ještě předtím, než byla započata práce na *AuthRepository*, byla vytvořena pomocná třída typu *sealed* s názvem *Resource* (Ukázka kódu 89). Tato třída slouží ke zpracování dat o úspěšné odpovědi serveru nebo k zpracování chybových zpráv.

```
typealias SimpleResource = Resource<Unit>
sealed class Resource<T>(val data: T? = null, val uiText: UiText? =
null){   class Success<T>(data: T?): Resource<T>(data), class Error <T> …
```

Ukázka kódu 89: Pomocná třída Resource
(zdroj: vlastní zpracování)

Do této třídy se „zabalí“ odpověď serverové aplikace, což usnadňuje práci s jejím vyhodnocením. Uvnitř této třídy byl vytvořen tzv. **typealias** s názvem SimpleResource – typealias poskytuje alternativní jméno pro některý z již existujících datových typů. V tomto případě se jedná o typ Resource <Unit>, který představuje odpověď serveru, která neobsahuje žádná data (*Unit* v jazyce Kotlin představuje prázdný datový typ, který nic nevrací).

Po vytvoření třídy Resource již bylo možné vytvořit funkci *registerAccount* uvnitř *AuthRepository* (Ukázka kódu 90):

```
interface AuthRepository {
    suspend fun registerAccount(
        email: String,
        username: String,
        password: String
    ): SimpleResource}
```

Ukázka kódu 90: Funkce registerAccount uvnitř AuthRepository
(zdroj: vlastní zpracování)

Tato funkce přijímá argumenty parametry *email*, *username* a *password* a její návratový typ je *SimpleResource*, tedy Resource <User>. Funkce *registerAccount* byla dále implementována uvnitř *AuthRepositoryImpl* (Ukázka kódu 91):

```
override suspend fun registerAccount(
    email: String,): SimpleResource {val request =
RegisterAccountRequest(email, username, password)
    return try {
        val response = authApi.registerAccount(request)
        if (response.successful) {Resource.Success(Unit)} else {}
```

Ukázka kódu 91: Implementace funkce registerAccount
(zdroj: vlastní zpracování)

V ukázce kódu lze vidět, že uvnitř funkce *registerAccount* je vytvořena instance třídy *RegisterAccountEmail* s hodnotami, které jí jsou předány parametry funkce. Následuje blok *try-catch*, ve kterém se nejprve do proměnné response, která značí odpověď serveru, načte výsledek funkce *registerAccount* z třídy *AuthApi*. Tato proměnná je typu *BasicApiResponse*. V závislosti na tom, jaká je odpověď serverové aplikace, se určí návratová hodnota funkce *registerAccount*: v případě, že dojde ke správné komunikaci se serverem, funkce *registerAccount* vrátí *Resource.Success* (tato hodnota značí úspěch). Pokud ke správné komunikaci nedojde, funkce vrací hodnotu *Resource.Error*, společně s chybovou zprávou, kterou serverová aplikace odeslala.

Dále byla vytvořena třída *RegisterUseCase*, která primárně slouží k validaci uživatelského požadavku o registraci: v této třídě se kontroluje, zda je správně vyplněn email, zda nejsou jsou v požadavku prázdná pole, zda je heslo správné délky apod. Pokud je požadavek v pořádku, proběhne volání funkce *registerAccount* ze třídy *AuthRepositoryImpl*. Třída *RegisterUseCase* poté navrací odpověď serverové aplikace.

```
private fun validateUsername(username: String): Errors?{
    val trimUsername = username.trim()
    if(trimUsername.isBlank()){
        return Errors.EmptyField}
    if(trimUsername.length < Constants.MIN USERNAME_LENGTH){
        return Errors.InputTooShort} return null}
```

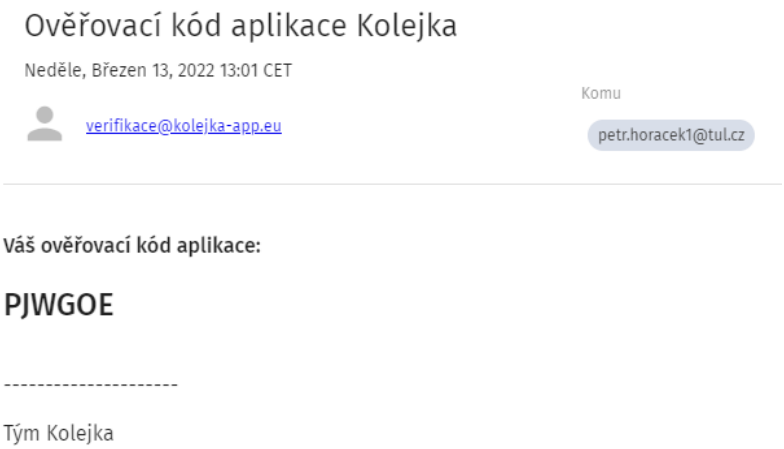
Ukázka kódu 92: Funkce pro validaci uživatelského jména
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 92) je vidět funkce *validateUsername*, která je umístěna uvnitř třídy *RegisterUseCase* a slouží k validaci uživatelského jména v požadavku.

Předposledním krokem implementace celého registračního řetězce bylo vytvoření třídy ViewModel, ve které dochází ke zpracování dat, která jsou odesílána serverové aplikaci a ke zpracování (a zobrazení) odpovědi serveru. Byla vytvořena třída *RegisterScreenViewModel*, která obsahuje stavové proměnné, jež využívá obrazovka uživatelského rozhraní *RegisterScreen*. Těmito proměnnými jsou např. *emailState*, *usernameState* či *passwordState*, které reprezentují stavy jednotlivých textových polí v *RegisterScreen*. Tato obrazovka byla ovšem oproti původní implementaci doplněna o ještě jedno textové pole, do kterého uživatel vkládá verifikační kód, který mu je poslán na email – tímto způsobem je ověřeno, zda se jedná o opravdový email.

Hlavní funkcí třídy *RegisterScreenViewModel* je funkce *register*, ve které dochází k odeslání požadavku na server s daty, které uživatel vypíše do textových polí. Registrace selže, pokud jsou uživatelem vyplněny špatné údaje (např. příliš krátké heslo) nebo pokud vyplní nesprávný ověřovací kód.

K odesílání emailů s ověřovacím kódem byla použita knihovna **MailDroid**, jež využívá k odesílání poskytnutou adresu SMTP serveru. Pro tyto účely bylo využito emailového hostingu, jenž poskytuje česká firma *MyDreams*. Ověřovací kód, který je uživatelům odesílán, sestává ze 6 náhodně vygenerovaných znaků. Email s ověřovacím kód poté vypadá následovně:



Obrázek 37: Ukázka emailu s ověřovacím kódem aplikace (zdroj: vlastní zpracování)

Posledním krokem pro implementaci registrace uživatelů bylo propojení třídy *RegisterScreenViewModel* a composable funkce *RegisterScreen*. K propojení bylo nejprve třeba anotovat třídu *RegisterScreenViewModel* anotací *@HiltViewModel* a konstruktor této

třídy anotovat *@Inject*. Díky těmto anotacím získá knihovna Hilt informaci o tom, jak injektovat tuto třídu jako závislost. Ve funkci *RegisterScreen* již poté stačilo jako parametr funkce poskytnout *RegisterScreenViewModel* následujícím způsobem (Ukázka kódu 93):

```
viewModel: RegisterScreenViewModel = hiltViewModel()
```

Ukázka kódu 93: Poskytnutí třídy ViewModel registrační obrazovce (zdroj: vlastní zpracování)

Poskytnutím třídy *RegisterScreenViewModel* bylo poté možné využívat všechny funkce, které poskytuje. V *onClick* funkci registračního tlačítka bylo tak možné vyvolat funkci *register* (Ukázka kódu 94), která se nachází ve ViewModelu (uvnitř ViewModelu je vyvolána skrze funkci *onEvent*, která byla popsána v předchozí části práce, která se zabývá implementací obrazovek).

```
Button(
    onClick = {
        viewModel.onEvent(RegisterEvent.Register)
}
```

Ukázka kódu 94: Volání registrační funkce z composable funkce registrační obrazovky (zdroj: vlastní zpracování)

V případě odeslání nesprávného požadavku se chybové hlášky zobrazují pomocí tzv. **Snackbaru**, což je composable funkce, kterou poskytuje Jetpack Compose. Snackbar má podobu lišty, která se zobrazuje na spodní straně obrazovky.

Obrázek 38: Chybová hláška zobrazena pomocí composable funkce Snackbar (zdroj: vlastní zpracování)

Podle schématu, které bylo vymezeno na začátku této kapitoly a jehož využití bylo předvedeno na příkladu registrace uživatele, se v aplikaci Kolejka následně řídí všechny funkce, při kterých dochází k volání serverové aplikace. V další části kapitoly bude popsáno propojení funkcí, které byly předvedeny v kapitole, jež se zabývala konstruováním serverové

aplikace, s uživatelským rozhraním aplikace Kolejka. Mezi tyto funkce patří přihlášení uživatele, vytvoření příspěvku, získání příspěvků a aktualizace uživatelských informací.

4.4.2 Implementace přihlášení uživatele

Implementace přihlašování v aplikaci probíhalo podobně, jako při implementaci registrace. Nejprve byla v interface třídě *AuthApi* vytvořena funkce *loginAccount*, která představuje endpoint aplikačního rozhraní, se kterým aplikace komunikuje. Funkce *loginAccount* přijímá argumenty pro parametr *request* (Ukázka kódu 95), který je typu *LoginAccountRequest*. Parametr *request* představuje uživatelský požadavek, který je poslán v jeho těle.

```
@POST(<CESTA API>)
suspend fun loginAccount(@Body request: LoginAccountRequest):
BasicApiResponse<AuthResponse>
```

Ukázka kódu 95: Funkce loginAccount
(zdroj: vlastní zpracování)

Tato funkce vrací *BasicApiResponse* typu *AuthResponse*, což je jednoduchá datová třída, jež reprezentuje odpověď serverové aplikace a obsahuje jediný parametr – *token*, který je typu String.

```
data class AuthResponse(
    val token: String
)
```

Ukázka kódu 96: Třída AuthResponse
(zdroj: vlastní zpracování)

Dále byla vytvořena funkce se stejným názvem (*loginAccount*) ve třídě *AuthRepository,* která byla následně implementována ve třídě *AuthRepositoryImpl*. Implementace se podobá implementaci registrační funkce – nejprve je vytvořena instance třídy *LoginAccountRequest* a v *try-catch* bloku se poté odesílá požadavek na server (Ukázka kódu 97). Rozdíl u *loginAccount* funkce je oproti registrační funkci ovšem v odpovědi serveru – při úspěšném přihlášení (správné přihlašovací údaje) se s odpovědí, která značí úspěch, odešle také JWT token. Tento token se poté uloží do tzv. **sharedPreferences**, což je malé úložiště, které se

vytvoří lokálně v datových složkách aplikace. Data se v sharedPreferences ukládají ve formě klíč-hodnota a samotný sharedPreferences soubor je uložen ve formátu XML.

```
if (response.successful) {sharedPreferences.edit().putString(JWT_TOKEN,
response.data?.token).apply()
```

Ukázka kódu 97: Zpracování odpovědi serverové aplikace u přihlašovací funkce (zdroj: vlastní zpracování)

Ukládání JWT do úložiště poté slouží k tomu, aby uživatel zůstal v aplikaci přihlášen po dobu platnosti tokenu. Ve chvíli, kdy uživatel zapne aplikaci, aplikace zkontroluje, zda je v úložišti sharedPreferences uložen platný JWT. V případě, že JWT existuje a je platný, uživatel je přesměrován do hlavní obrazovky s příspěvky, v opačném případě je přesměrován do přihlašovací obrazovky.

K tomu, aby byl uživatel autorizován k přístupu k mnoha funkcím serverové aplikace, je třeba, aby jeho odeslaný HTTP požadavek obsahoval JWT v hlavičce požadavku. O vytváření požadavků se stará knihovna Retrofit společně s knihovnou OkHttp. Pomocí OkHttp lze poté také modifikovat hlavičky požadavků:

```
@Provides
@Singleton
fun provideOkHttpClient(sharedPreferences: SharedPreferences):
OkHttpClient {
    return OkHttpClient.Builder()
        .addInterceptor {
            val token = sharedPreferences.getString(JWT_TOKEN, "") ?: ""
            val modifiedRequest = it.request().newBuilder()
                .addHeader("Authorization", "Bearer $token")
                .build()
            it.proceed(modifiedRequest)
        }
        .build()
}
```

Ukázka kódu 98: Modifikování hlavičky pomocí OkHttp (zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 98) lze vidět, jak je vytvořena instance OkHttp klientu, která je poskytována Retrofit instancím. Důležitý kód se v ukázce nachází v těle lambda funkce *addInterceptor*. Nejprve se zde vytvoří proměnná *token*, která načte JWT z úložiště

sharedPreferences nebo načte prázdný string, pokud JWT v úložišti neexistuje. Proměnná *modifierRequest* poté představuje upravenou podobu HTTP požadavku, která je doplněna o hlavičku s tokenem.

Co se týká implementace přihlašovací funkce, po vytvoření funkce v *AuthRepositoryImpl* byla vytvořena třída *LoginUseCase*, která stejně jako při implementaci registrace slouží k validaci přihlašovacích údajů.

S třídou *LoginUseCase* poté komunikuje třída *LoginScreenViewModel*, uvnitř které dochází k volání třídy *LoginUseCase* (která volá repositář *AuthRepositoryImpl*, jež volá třídu *AuthApi*, díky které dochází ke komunikaci se serverovou aplikací). Volání *LoginUseCase* ve třídě *LoginScreenViewModel* probíhá ve funkci *login*. V závislosti na odpovědi serveru poté dojde k další operaci – navigaci do hlavní obrazovky s příspěvky / zobrazení chybové hlášky. K navigaci / zobrazení chyby byla vytvořena třída typu sealed *UiEvent*, která má následující podobu (Ukázka kódu 99):

```
sealed class UiEvent {
    data class ShowSnackbar(val uiText: UiText) : UiEvent()
    data class Navigate(val route: String): UiEvent()
    data class ShowSnackbarAndNavigate(val uiText: UiText, val route:
String): UiEvent()
}
```

Ukázka kódu 99: Třída UiEvent
(zdroj: vlastní zpracování)

Ve třídě *LoginScreenViewModel* je poté vytvořena proměnná *eventFlow* typu **MutableSharedFlow**, která slouží k emitování hodnot, které jí jsou poskytnuty. Využití MutableSharedFlow je také vhodné z důvodů, že emitace hodnoty proběhne pouze jednou. Emitace hodnot v LoginScreenViewModelu probíhá následovně (Ukázka kódu 100):

```
when (loginResult.result) {
                is Resource.Success -> {
                    _eventFlow.emit(
                        UiEvent.Navigate(Screen.AppHolderScreen.route)
                    )
                    _emailState.value = StandardTextfieldState()
                    _passwordState.value = PasswordTextfieldState()
                }
                is Resource.Error -> {
                    _eventFlow.emit(
                        UiEvent.ShowSnackbar(uiText =
loginResult.result.uiText ?: UiText.unknownError())
                    )
                }
```

Ukázka kódu 100: Emitování hodnot typu UiEvent
(zdroj: vlastní zpracování)

Pokud je požadavek na serverovou aplikaci úspěšný, *eventFlow* emituje navigační událost (*UiEvent.Navigate*) s cestou, kam je třeba uživatele navigovat. V případě neúspěšného požadavku je naopak emitována událost *UiEvent.ShowSnackbar*, která v sobě drží chybovou hlášku. Emitované události se poté „posbírají“ v composable funkci *LoginScreen* následujícím způsobem (Ukázka kódu 101):

```
   LaunchedEffect(key1 = true) {
        viewModel.eventFlow.collectLatest { event ->
            when (event) {
                is UiEvent.ShowSnackbar -> {
                    scaffoldState.snackbarHostState.showSnackbar(
                        message = event.uiText.asString(localContext),
                        duration = SnackbarDuration.Short
                    )
                } …
            }
        }
    }
```

Ukázka kódu 101: Sbírání emitovaných událostí
(zdroj: vlastní zpracování)

Sbírání emitované události proběhne pomocí funkce *collectLatest*. Uvnitř této funkce se poté vyhodnotí, která událost byla emitována, a v závislosti na tom se uskuteční např. zobrazení Snackbaru (Ukázka kódu 101). Sbírání emitovaných hodnot je dále obaleno v composable funkci **LaunchedEffect**, což je funkce poskytnutá v základu Jetpack Compose. Tato funkce

zajistí, že ve chvíli, kdy dojde k neúspěšnému požadavku na serverovou aplikaci a zobrazí se chybová hláška, při změně orientace displeje se chybová hláška nezobrazí znova.

Ve zbytku přihlašovací obrazovky se poté data z ViewModelu zobrazují stejně jako v registrační obrazovce.

4.4.3 Vytváření příspěvků v aplikaci

Vytváření příspěvků využívá stejné schéma, jako registrace a přihlašování uživatelů. Nejprve se ve třídě *PostApi* (pro práci s jednotlivými datovými třídami byly vytvořeny oddělené API třídy ke zlepšení přehlednosti kódu) vytvořila funkce *createNewPost*, která představuje endpoint aplikačního rozhraní serverové aplikace. Funkce se stejným názvem je poté vytvořena uvnitř třídy *PostRepository* a implementována ve třídě *PostRepositoryImpl*. Návratový typ této funkce je opět *SimpleResource*, neboť po vytvoření příspěvku není serverovou aplikací odeslána žádná odpověď. Funkce *createNewPost* (Ukázka kódu 102) dále přijímá argumenty pro parametry *title* (název příspěvku), *description* (popis příspěvku), *limit* (počet volných míst u příspěvku), *type* (typ příspěvku), *date* (datum – pouze u událostí), *location* (lokace události / nabídky), *postImageURL* (URL fotografie příspěvku).

```
override suspend fun createNewPost(title: String, description: String,
limit: Int?,…
```

Ukázka kódu 102: Funkce createNewPost
(zdroj: vlastní zpracování)

Dále byla vytvořena třída *NewPostUseCase*, ve které se opět validuje uživatelský požadavek – probíhá zde kontrola, zda jsou vyplněna všechna pole u nového příspěvku. V případě, že nejsou, aplikace ukáže chybovou hlášku.

Nakonec se ve třídě *NewPostScreenViewModel* vytváří nový příspěvek s použitím dat ze stavových proměnných. Data, která jsou ve stavových proměnných uložena, vytváří uživatel vyplněním textových polí (typu *StandardTextField*) v obrazovce *NewPostScreen*. Stavovou proměnnou je např. *titleState*, která reprezentuje stav textového pole, do kterého se vypisuje popisek. Vytvoření požadavku k přidání příspěvku poté vypadá následovně (Ukázka kódu 103):

```
fun createPost() {
    viewModelScope.launch {
         _isLoading.value = true
         val postResult = newPostUseCase(
            title = titleState.value.text,
            description = descriptionState.value.text,
            limit = limitState.value.text.toIntOrNull(),
            type = when (optionsRadioState.value.eventEnabled) {
                true -> PostType.Event.type
                else -> PostType.Offer.type
            },…)
        _isLoading.value = false
```

Ukázka kódu 103: Vytváření požadavku uvnitř třídy ViewModel
(zdroj: vlastní zpracování)

Posledním úkolem, který bylo k vytváření příspěvků třeba vyřešit, bylo získání URL obrázku. Kliknutím na pole pro vybrání obrázku v *NewPostScreen* se otevře galerie obrázků v telefonu (pomocí funkce *rememberLauncherForActivityResult*, která je obsažena v Jetpack Compose). Vybráním obrázku se získá jeho URI (identifikátor obrázku), jež se dále předá do dalšího volání funkce *rememberLauncherForActivityResult,* která umožní oříznutí obrázku do správných dimenzí (ořezávání zprostředkuje externí knihovna **UCrop**). Výstupem této funkce je opět URI, tentokrát ořízntého obrázku. Získané URI bylo dále třeba transformovat do veřejné URL, ze které může aplikace obrázek načíst.

Tento problém byl v aplikaci vyřešen využitím služby **Cloudinary**, což je služba, která umožňuje nahrávání obrázků do cloudového úložiště. Tato služba má bezplatnou verzi, která poskytuje 25 GB prostoru pro nahrávání médií (s možností placených rozšíření). Další výhodou služby Cloudinary je, že lze využít Cloudinary knihovnu pro Android, která umožňuje jednoduché nahrávání obrázků do cloudu přímo z kódu. Tato možnost byla v rámci aplikace využita a použití vypadá následovně:

```
MediaManager.get().upload(uri).preprocess(ImagePreprocessChain().saveWith
(BitmapEncoder(BitmapEncoder.Format.JPEG, 80))).callback(object :
UploadCallback {
    override fun onStart(requestId: String?) {isLoading.value = true}
    override fun onSuccess(resultData: MutableMap<Any?, Any?>?) {
        _imageUrl.value = resultData?.getValue("secure_url").toString()
        createPost()}
```

Ukázka kódu 104: Použití knihovny Cloudinary
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 104) je zavolán singleton objekt *MediaManager* s metodou *get*, která slouží jako vstupní bod pro komunikaci se službou Cloudinary. Další volanou metodou je poté *upload*, které je poskytnuto URI ořízntutého obrázku. Metoda *upload* umožní nahrání obrázku do Cloudinary. Další metodou je *preprocess*, pomocí které lze nahraný obrázek komprimovat – v případě aplikace Kolejka se nahraný obrázek uloží ve formátu JPEG s 85% kvalitou. Poslední metodou je poté *callback*, která volá funkce v závislosti na průběhu nahrávání obrázku. Na začátku nahrávání obrázku se zavolá funkce *onStart*, uvnitř které se změní hodnota proměnné *isLoading* (která slouží k zobrazení načítacího indikátoru) na *true*. V případě úspěšného nahrání se zvolá funkce *onSuccess* a do stavové proměnné *imageUrl* se načte URL nahraného obrázku. Po uložení URL hodnoty se dále zvolá funkce *createPost*, která se stará o vytvoření příspěvku skrze serverovou aplikaci.

```
if (viewModel.isLoading.value) {
    CircularProgressIndicator(
        modifier = Modifier.align(Alignment.CenterHorizontally),
        color = DarkPurple
    )
} else {
     Row(
        modifier = Modifier.fillMaxSize(),
        …
```

Ukázka kódu 105: Použití proměnné isLoading
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 105) je zobrazeno, jak je v *NewPostScreen* použita proměnná *isLoading*. Pokud má proměnná *isLoading* hodnotu *true*, na obrazovce se místo tlačítka pro přidání příspěvku zobrazí composable funkce **CircularProgressIndicator**. Tato composable funkce je poskytnuta v základu Jetpack Compose a má podobu načítacího kolečka. Pokud je hodnota proměnné *isLoading* false, na obrazovce se zobrazí tlačítko pro přidávání. Použitím indikátoru načítání je uživatel informován, že aplikace pracuje a zároveň je uživateli znemožněno klikat na přidávací tlačítko i během procesu vytváření příspěvku. Composable funkce CircularProgressIndicator je tímto způsobem využita napříč celou aplikací.

Obrázek 39: Composable funkce CircularProgressIndicator
(zdroj: vlastní zpracování)

4.4.4 Zobrazování příspěvků v aplikaci

Pro získávání příspěvků byla využita knihovna Paging 3, knihovna, která je součástí sady Android Jetpack. První krok implementace získávání příspěvků byl stejný jako pro předchozí funkcionality – v API třídě (v případě získávání souborů ve třídě *PostApi*) byla vytvořena funkce, která reprezentuje endpoint aplikačního rozhraní.

```
@GET(<CESTA API>)
suspend fun getPostsByAll(
    @Query("page") page: Int,
    @Query("pageSize") pageSize: Int
): List<Post>
```

Ukázka kódu 106: Funkce getPostsByAll v aplikaci Kolejka
(zdroj: vlastní zpracování)

Funkce *getPostsByAll* (Ukázka kódu 106), která vrací všechny příspěvky z databáze, má dva parametry – *page* a *pageSize*. Tyto parametry jsou anotovány pomocí *@Query*, což značí, že parametry funkce se použijí jako parametry GET požadavku (pomocí kterého se získávají ze serverové aplikace příspěvky). Parametry *page* a *pageSize* byly popsány v části, která se věnuje vytvoření serverové aplikace.

Ve třídě *PostRepository* se poté vytvořila proměnná *posts*, která představuje načtené příspěvky. Proměnná *posts* je typu Flow<PagingData<Post>>. Flow je datový typ, který je specifický pro jazyk Kotlin, a slouží k asynchronnímu, sekvenčnímu emitování hodnot (více, než jedné). V případě aplikace Kolejka jsou emitovány objekty typu PagingData<Post>. PagingData je typ, který je součástí knihovny Paging 3 a představuje data, která budou stránkovány. Pro PagingData se specifikuje typ objektů, které jsou v něm uloženy – v uvedeném příkladu se jedná o objekty typu *Post*.

Proměnná *posts* je dále implementována uvnitř třídy PostRepositoryImpl následujícím způsobem (Ukázka kódu 107):

```
posts: Flow<PagingData<Post>>get() = Pager(PagingConfig(pageSize =
Constants.DEFAULT_PAGE_SIZE)) {AllPostsSource(postApi)}.flow
```

Ukázka kódu 107: Implementace proměnné posts
(zdroj: vlastní zpracování)

Při implementaci se do proměnné *posts* načte instance třídy Pager. Tato třída slouží k načítání dat ze zdroje, kterým je v uvedeném případě instance třídy *AllPostsSource*. *AllPostsSource* je třída, která implementuje abstraktní třídu PagingSource. PagingSource definuje zdroj dat a způsob získávání dat z tohoto zdroje – zdrojem dat může být např. lokální, či vzdálená databáze. Třída PagingSource přijímá dva argumenty – klíč (*key*) a hodnotu (*value*). Hodnota definuje typ dat a klíčem se např. specifikuje číslo stránky dat, která má být načtena. PagingSource také obsahuje funkci *load*, uvnitř které se implementuje načítání stránkovaných dat.

```
override suspend fun load(params: PagingSource.LoadParams<Int>):
    return try { val nextPage = params.key ?: currentPage
        val posts = api.getPostsByAll(...
            data = posts,…
```

Ukázka kódu 108: Stránkovací funkce load
(zdroj: vlastní zpracování)

Uvnitř funkce *load* (Ukázka kódu 108) je nejprve definována proměnná *nextPage*, která drží informaci o číslu stránky, která je načítána. Dále se do proměnné *posts* načítají data z API serverové aplikace. Načtená data se nakonec uloží do třídy *LoadResult.Page*, která také spravuje inkrementaci/dekrementaci čísel stránek.

Dále byla vytvořena třída *GetAllPostsUseCase*, která pouze přebírá načtené příspěvky z *PostRepositoryImpl* a posílá je do ViewModel tříd hlavních obrazovek s příspěvky – *EventScreen* a *OfferScreen*.

Ve ViewModel třídách se poté načtou příspěvky z *GetAllPostsUseCase* jedním řádkem kódu (Ukázka kódu 109):

```
var posts = getAllPostsUseCase().cachedIn(viewModelScope)
```

Ukázka kódu 109: Načtení příspěvků z GetAllPostsUseCase
(zdroj: vlastní zpracování)

Načtené příspěvky se poté v composable funkci *EventScreen* (či *OfferScreen*), která představuje obrazovku s příspěvky, „posbírají“ z flow pomocí metody *collectAsLazyPagingItems* (Ukázka kódu 110):

```
val posts = viewModel.posts.collectAsLazyPagingItems()
```

Ukázka kódu 110: Použití metody collectAsLazyPagingItems
(zdroj: vlastní zpracování)

Příspěvky, které byly „posbírány“, jsou poté zobrazeny na obrazovce využitím composable funkce LazyColumn.

4.4.5 Implementace aktualizace uživatelských informací

Pro implementaci funkce, která umožňuje změnu uživatelských informací, byla nejprve ve třídě *UserApi* vytvořena funkce *updateUserInfo*, která představuje endpoint aplikačního rozhraní serverové aplikace (Ukázka kódu 111).

```
suspend fun updateUserInfo(@Body updateUserRequest: UpdateUserRequest):
BasicApiResponse<Unit>
```

Ukázka kódu 111: Funkce updateUserInfo
(zdroj: vlastní zpracování)

Postup je poté stejný, jako byl u předešlých funkcionalit aplikace. Ve třídě UserRepository byla vytvořena funkce *updateUserInfo*, která byla implementována uvnitř třídy *UserRepositoryImpl*. V implementaci této funkce dochází k vytvoření požadavku na serverovou aplikaci a vyhodnocuje se odpověď, která je serverovou aplikací odeslána.

Dále byla vytvořena třída *UpdateProfileUseCase*, která slouží především ke kontrole, zda při změně uživatelských informací nedošlo k chybě. Chybou v tomto případě může být

například změna uživatelského jména na prázdný text, což pochopitelně není povoleno. Z *UpdateProfileUseCase* se také volá funkce *updateUserInfo* ze třídy *UserRepositoryImpl*, která uskuteční změnu uživatelských informací.

Instanci třídy *UpdateProfileUseCase* poté volá třída *EditProfileViewModel*, která je typu ViewModel a která je držitelem stavu pro obrazovku *EditProfileDialog*. Uvnitř této obrazovky dochází ke specifikaci nových uživatelských informací. Pomocí třídy *EditProfileViewModel* se poté vytváří požadavek na serverovou aplikaci (požadavek je vytvořen ve ViewModel třídě, poté je validován v *UpdateProfileUseCase* a následně odeslán do *UserRepositoryImpl*, kde dochází k volání serverové aplikace) s daty, které vytvoří uživatel a které jsou uloženy ve stavových proměnných uvnitř ViewModelu. Nahrávání vlastní profilové fotky zde funguje stejně jako u přidávání příspěvků, a to sice pomocí služby Cloudinary.

Kromě uvedených funkcí byly pochopitelně dále implementovány i další funkcionality aplikace, mezi něž patří např. komentování, změna hesla, odhlášení či mazání příspěvků. Tyto funkcionality již ovšem v práci popsány nebudou. Jejich implementace byla ovšem programována podle schématu, který byl předveden u popsaných implementací. Naprogramováním všech potřebných funkcí tak byla vytvořena první verze aplikace Kolejka. Po dokončení první verze byla serverová aplikace nahrána na PaaS službu Heroku.

Je nutné podotknout, že první verze aplikace Kolejka obsahuje některé další obrazovky, které nebyly pokryty v počátečním návrhu uživatelského rozhraní. Jednou z takových obrazovek je např. obrazovka pro změnu hesla.

4.5 Nasazení serverové aplikace a zabezpečení dat

K tomu, aby k funkcím aplikace Kolejka mohli uživatelé přistupovat odkudkoliv, bylo nutné nasadit serverovou aplikaci na veřejně přístupný server. Po úvaze nad různými možnostmi hostingu byla k běhu serverové aplikace nakonec zvolena platforma Heroku. Heroku je tzv. cloudová PaaS (Platform as a Service) založená na kontejnerech. Kontejnery jsou balíčky, které obsahují nejen serverovou aplikaci, ale také všechny závislosti potřebné k jejímu spuštění (Shapland 2021). V rámci služby Heroku se používají tzv. **dynos**, což jsou virtualizované linuxové kontejnery, které jsou určeny ke spouštění kódu na základě uživatelem zadaného příkazu (Heroku 2022).

Volba služby Heroku byla také vhodná z důvodu, že framework KTor umožňuje jednoduché nasazení serverové aplikace na platformu Heroku přímo z vývojářského prostředí IntelliJ. Po nasazení pouze zbývalo kontejneru serverové aplikace v Heroku poskytnout odkaz na cloudovou databázi a vše bylo připraveno k běhu aplikace.

Platforma Heroku, stejně jako MongoDB Atlas, umožňuje použití bezplatné verze cloudové služby k provozu aplikací. Bezplatná verze poskytuje 512 MB RAM, možnost nasazování aplikace pomocí Gitu či například využití vlastní domény. U bezplatné verze ovšem dochází ke spánku serverové aplikace po 30 minutách neaktivity. Aplikace se uspává z důvodů, že bezplatná verze poskytuje 550-1000 dyno hodin měsíčně, a proto ve chvíli, kdy s aplikací není komunikováno, přenechává výpočetní zdroje ostatním aplikacím. Pro účely testování aplikace je však bezplatná verze dostačující.

K tomu, aby byli uživatelé aplikace Kolejka lépe chráněni, bylo před nasazením aplikace ještě nutné zabezpečit citlivá data. Do této chvíle totiž docházelo k ukládání hesel do databáze ve formě jednoduché textu, což je z hlediska bezpečnosti nepřípustné. V serverové aplikaci byl tedy vytvořen soubor *PasswordUtility*, který obsahuje dvě funkce: *getHashWithSalt* a *checkHashForPassword*. Funkce *getHashWithSalt* funguje jako hashovací funkce, která vytvoří otisk z poskytnutého hesla, ke kterému přidá kryptografickou sůl. Hash se solí se ukládá do databáze.

Funkce *checkHashForPassword* poté kontroluje, zda vstupní heslo uživatele (při přihlašování) odpovídá heslu v databázi. Ze vstupní hesla se vytvoří hash, který se porovnává s hashem hesla v databázi. Pokud jsou si hashe rovny, vstupní heslo je správné.

Vzhledem k tomu, že serverová aplikace běží za pomoci cloudových platforem Heroku a MongoDB Atlas, spousta bezpečnostních funkcí nebylo třeba řešit samostatně. Heroku využívá protokol SSL, tudíž komunikace mezi klientem a serverovou aplikací probíhá přes HTTPS. V MongoDB Atlas poté databáze běží ve virtuálních, privátních cloudech (VPC) s dedikovanými firewally, kterým je třeba poskytnout seznam IP adres, které mohou k databázím přistupovat. MongoDB Atlas dále používá pro zabezpečení síťového provozu protokol TLS.

5 Uživatelské testování aplikace

K tomu, aby byl posouzen uživatelský zážitek a použitelnost aplikace Kolejka, bylo provedeno uživatelské testování. V rámci testování byl 10 studentům ubytovaných na kolejích Harcov rozeslán instalační soubor aplikace, společně s dotazníkem, který sestával ze dvou sekcí otázek:

- Používání aplikace
- Zpětná vazba k aplikaci

Používání aplikace

V této sekci dotazníku byli respondenti dotazování na složitost jednotlivých operací v rámci aplikace. V hlavičce otázky byla vždy specifikována operace, kterou měli respondenti vykonat, a v tělu otázky poté respondenti hodnotili složitost od 1-5, kde hodnota 1 značila „žádné problémy s vykonáním operace“ a hodnota 5 „operace nešla vykonat“.

Operace, o které byli respondenti požádáni, byly následující:

- Registrace
- Přihlášení
- Změna uživatelské fotky a barvy uživatelského praporu
- Přidání příspěvku
- Připojení se ke sledování příspěvku
- Napsání komentáře
- Smazání příspěvku

Ačkoliv se jedná o velmi jednoduché operace, bylo třeba zjistit, zda je pohyb skrz aplikací dostatečně jasný a přehledný.

Zpětná vazba k aplikaci

V druhé sekci dotazníku byli respondenti požádáni o zodpovězení otázek, které se týkaly jejich uživatelského prožitku z používání aplikace. V dotazníku byly položeny následující otázky:

- Jak byste na stupnici 1-5 ohodnotili design aplikace? (hodnocení jako ve škole)

    - Volitelné – popište, co byste na designu aplikace změnili.
- Jak byste na stupnici 1-5 ohodnotili funkcionalitu aplikace? (hodnocení jako ve škole)
    - Volitelné – napište, které funkce byste v aplikaci změnili.
    - Volitelné – napište, jaké funkce podle Vás aplikace postrádá.
- Narazili jste při testování aplikace na nějaké problémy (padání aplikace, neočekávané chování, ...)? Pokud ano, vypište je prosím.
- Pokud by byla aplikace Kolejka vydána oficiálně, používali byste ji?

5.1 Vyhodnocení dotazníků

Používání aplikace

Výsledky dotazníku v sekci Používání aplikace jsou obsaženy v následující tabulce:

Tabulka 1: Hodnocení aplikace

Operace	Průměrné hodnocení složitosti (n = 10)
Registrace	1,1
Přihlášení	1,2
Změna uživatelských údajů	1,4
Přidání příspěvku	1
Připojení ke sledování příspěvku	1
Napsání komentáře	1
Smazání příspěvku	1

(zdroj: vlastní zpracování)

Vzhledem k jednoduchosti operací byly průměrné výsledky hodnocení složitosti v rozmezí 1-1,5 u všech otázek. Nejhorší hodnocení zde obdržela operace „Změna uživatelských údajů“, a to sice 1,4. Po diskuzi s respondenty bylo zjištěno, že některým v této pasáži vadilo

neprojevení se změny profilového obrázku v editačním dialogu. Ve chvíli, kdy si tito uživatelé změnili fotku profilu, v dialogu byla stále vidět stará fotografie a až po potvrzení editace se jim v profilu zobrazila správná, aktualizovaná fotografie. Tento problém byl na základě připomínky vyřešen.

Zpětná vazba k aplikaci

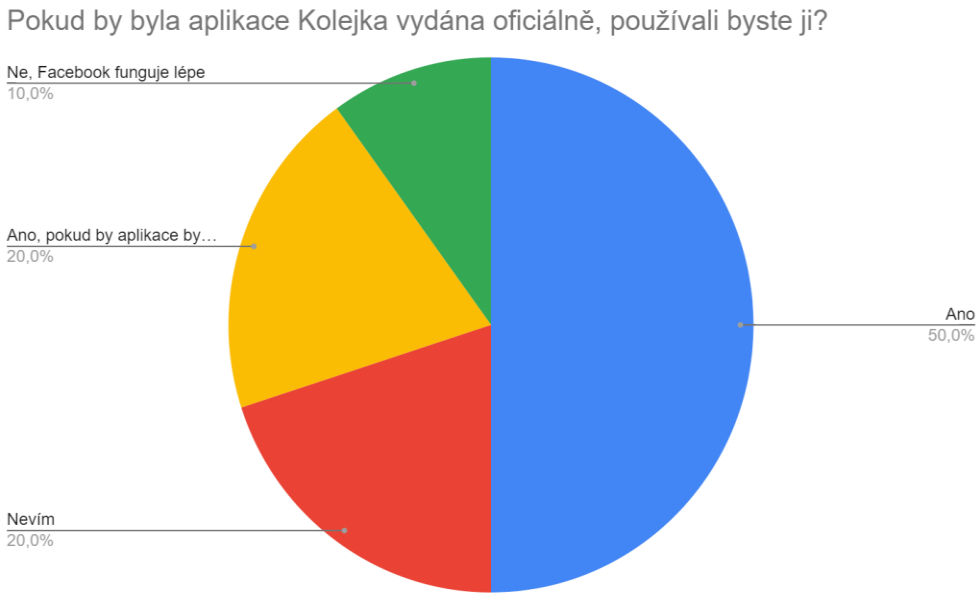
Výsledky dotazníku v sekci „Zpětná vazba k aplikaci“ budou rozebrány otázku po otázce:

- Jak byste na stupnici 1-5 ohodnotili design aplikace?
    - Průměrné hodnocení u této otázky bylo **1,6**. Volitelná možnost dopsání připomínky k této otázce nebyla využita všemi respondenty, avšak jeden z respondentů vyjádřil nespokojenost kvůli chybějící podpoře pro tmavý režim telefonu. Tento problém byl částečně vyřešen správnou implementací barev v aplikaci pomocí Material themingu (který při inicializaci projektu nebyl využit).
- Jak byste na stupnici 1-5 ohodnotili funkcionalitu aplikace? (hodnocení jako ve škole)
    - Průměrné hodnocení u této otázky bylo **1,9**. U volitelných otázek, které byly vázány k této otázce, byly získány odpovědi, které obsahovaly dobré připomínky k funkcím aplikace.
    - U otázky, které funkce by uživatelé změnili, se např. dvakrát objevila odpověď, že by vhodné přidat kolonku s cenou při přidávání nabídky. Další připomínkou bylo, že při změně barev uživatelského praporu se nezmění barva ikon na praporu. Tudíž při změně barvy praporu na černou nejsou vidět jednotlivé ikony.
    - U otázky, jaké funkce podle uživatelů aplikace postrádá, byla například navrhnuta možnost přidat do aplikace obrazovku, která by obsahovala informace o kolejních novinkách (změny cen kolejného apod.). Ve 4 odpovědích poté bylo navrhnuto lepší informování o notifikacích v aplikaci a smazání starých notifikací. Dále bylo např. doporučeno, že by bylo vhodné doplnit aplikaci o možnost změny údajů příspěvku.
    - Doplnění některých z těchto funkcí bude popsáno v další části této kapitoly
- Narazili jste při testování aplikace na nějaké problémy (padání aplikace, neočekávané chování, ...)? Pokud ano, vypište je prosím.

- U této otázky se vyskytly připomínky k občasnému dlouhému načítání aplikace. Tyto prodlevy jsou způsobeny využitím bezplatné verze platformy Heroku, která, jak již bylo zmíněno, aplikaci uspává při neaktivitě. Tudíž tento problém by byl vyřešen povýšením na placenou verzi Heroku. Někteří respondenti dále upozornili na nesprávné chování zpětného tlačítka v aplikaci – při stisknutí tlačítka zpět se aplikace místo vypnutí navigovala na předchozí navštívenou obrazovku. Tento problém byl vyřešen opravou navigační komponenty aplikace.

- Pokud by byla aplikace Kolejka vydána oficiálně, používali byste ji?

  - Tato otázka byla položena formou výběrové otázky s otevřenou odpovědí. Respondenti měli možnost výběru 1 z 4 předpřipravených odpovědí (Ano | Ano, pokud by aplikace byla doplněna o navrhované funkce | Nevím | Ne) nebo napsání vlastní odpovědi. Výsledky této odpovědi jsou obsaženy v následujícím grafu:



Obrázek 40: Odpovědi respondentů zobrazeny v grafu (zdroj: vlastní zpracování)

  - Jak lze vidět, 70 % respondentů odpovědělo odpovědí „Ano“ nebo „Ano, pokud by aplikace byla doplněna o navrhované funkce“. Dalších 20 %

odpovědělo odpovědí „Nevím“ a pouze jeden z 10 odpověděl odpovědí „Ne“, doplněnou o připomínku, že sociální síť Facebook v rámci nabízených funkcí funguje lépe. Lze tedy vidět, že v případě doplnění aplikace Kolejka o další funkce by byl o aplikaci poměrné vysoký zájem. Tento fakt naznačil autorovi práce, že má cenu aplikaci dále zlepšovat.

5.2 Rozšíření aplikace na základě uživatelských doporučení

Následující část bude věnována implementaci funkcí, které byly respondenty navrhnuty v odpovědích dotazníku.

5.2.1 Dynamická změna barev textu uživatelského praporu

První navrženou změnou aplikace bylo měnění barvy textu uživatelského praporu v závislosti na barvě pozadí praporu. Jak již bylo zmíněno, ve chvíli, kdy si uživatel v aktuální verzi aplikace změní barvu praporu na černou, i barva textu (a ikon) v praporu zůstane černá. V aplikaci tedy byla vytvořena funkce *bannerTextColor*, která navrací bílou nebo černou barvu v závislosti na parametrech funkce, kterými jsou *red*, *green* a *blue*. Tyto parametry představují jednotlivé složky barvy pozadí uživatelského praporu. Funkce *bannerTextColor* se v kódu využívá následovně (Ukázka kódu 112):

```
Text(
    text = user.username,
    style = Typography.body1,
    color = bannerTextColor(red = user.bannerR, green = user.bannerG,
blue = user.bannerB)
)
```

Ukázka kódu 112: Použití funkce bannerTextColor
(zdroj: vlastní zpracování)

V ukázce kódu lze vidět, že composable funkce Text přijímá jako argument pro parametr color vytvořenou funkci *bannerTextColor*. Funkce *bannerTextColor* poté přijímá jako argumenty pro parametry *red*, *green* a *blue* barevné složky uživatelského praporu, které se načítají z databáze. Na základě barevných složek uživatelského praporu se jednoduše počítá, zda funkce vrátí černou či bílou barvu. Vrácená barva je poté barva textu a ikon uživatelského praporu. Text i ikonky jsou tedy stále viditelné.

5.2.2 Cena nabídky

Dalším z návrhů byla možnost přidání ceny nabídky. K implementaci tohoto návrhu musela být pozměněna třída datová Post, které byl přidán parametr *price*, jenž je datového typu Int. Třída *Post* musela být pozměněna i v serverové aplikaci. Dále byla pozměněna funkce k přidávání příspěvků, která byla obohacena o možnost poslat v serverovém požadavku informaci o ceně.

Po úpravě backend funkcí bylo dále nutné zahrnout možnost přidání ceny i do UI aplikace. Obrazovka pro přidávání příspěvku typu *Offer* byla doplněna o textové pole, do kterého lze vypsat cenu nabídky. Cena nabídky je nepovinný údaj, který může zůstat nevyplněn. V uživatelském rozhraní obrazovky pro přidávání příspěvků byla také vytvořena možnost otevření/zavření textového pole pro přidání ceny.

Obrázek 41: Textové pole pro přidání ceny s možností zavření/otevření (zdroj: vlastní zpracování)

Nakonec bylo třeba přidat informaci o ceně do obrazovky s detailem příspěvku.

5.2.3 Upozornění na notifikace a smazání notifikací

Notifikace se ve verzi, která byla odeslána k testování, pouze zobrazují v obrazovce notifikací, ale není na ně žádným způsobem upozorněno. Jak již bylo zmíněno, několika respondenty dotazníku bylo navrhnuto, že by bylo vhodné dodat do aplikace upozornění na příchozí notifikace. V dotazníku byla také navržena možnost smazání notifikací.

Do aplikace tedy byla přidána funkce, která zobrazuje počet nových notifikací ve spodní liště aplikace. K implementaci této funkce byla v serverové aplikaci nejprve vytvořena datová třída *NotificationCount*, která obsahuje informaci o ID uživatele a počtu nepřečtených notifikací. Ve chvíli, kdy je pro uživatele vytvořena notifikace (např. v okamžiku, kdy se někdo připojí k jeho příspěvku), počet nepřečtených notifikací se ve třídě *NotificationCount* navýší o 1 (jedná se o instanci třídy *NotificationCount*, která je spjata s daným uživatelem skrze uživatelské ID, které instance této třídy obsahuje).

V serverové aplikaci poté byla vytvořena funkce, která navrací počet notifikací pro daného uživatele. V aplikaci byla tato funkce implementována pomocí datového typu **flow**, který umožňuje asynchronní, sekvenční emitování hodnot. Implementace vypadá v kódu následovně:

```
override fun getNotificationsCount(): Flow<Int> {
    return flow {
        while (true){
            emit(api.getNotificationsCount())
            delay(NOTIFICATION_UPDATE_DELAY)
        }
    }
}
```

Ukázka kódu 113: Použití flow k navrácení počtu notifikací
(zdroj: vlastní zpracování)

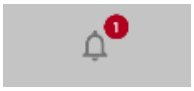
V ukázce kódu (Ukázka kódu 113) lze vidět, že funkce *getNotificationsCount* vrací typ Flow<Int>. Uvnitř funkce se poté vrací instance flow. Emitování hodnot počtu notifikací poté probíhá neustále (díky bloku *while(true)*), vždy každých 5 sekund (hodnota 5 sekund je uložena v proměnné *NOTIFICATION_UPDATE_DELAY*).

Sbírání hodnot z flow se poté provádí ve ViewModel třídě composable funkce AppHolder, která zobrazuje spodní lištu aplikace a drží ostatní obrazovky. Sbírání hodnot probíhá pomocí funkce *collectLatest*. Počet notifikací se poté zobrazuje ve spodní liště obrazovky u ikony notifikací. Zobrazení počtu notifikací je umožněno pomocí composable funkcí **BadgedBox** a **Badge** (Ukázka kódu 114).

```
BadgedBox(badge = {
    if (notificationsCount > 0) {
        Badge {Text(
                text = notificationsCount.toString())}}}) {
    Column(horizontalAlignment = CenterHorizontally) {
        Icon(imageVector = item.icon, contentDescription = item.name)}}
```

Ukázka kódu 114: Použití composable funkcí BadgedBox a Badge
(zdroj: vlastní zpracování)

Pomocí těchto composable funkcí lze „obalit“ composable funkci a přidat jim dekorativní „odznak“, který může zobrazovat různé informace – v případě aplikace Kolejka se jedná o počet notifikací.



Obrázek 42: Ikona notifikací ve spodní liště, doplněna o počet nepřečtených notifikací (zdroj: vlastní zpracování)

Rozkliknutím obrazovky s notifikacemi poté dojde k volání serverové aplikace a vynuluje se počet nepřečtených notifikací.

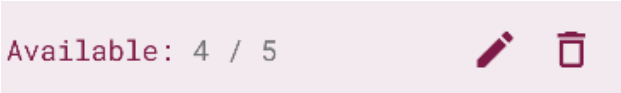
Ke smazání notifikací poté byla vytvořena funkce v serverové aplikaci, která maže veškeré notifikace pro daného uživatele. V uživatelském rozhraní notifikační obrazovky aplikace bylo přidáno otevírací tlačítko k mazání notifikací pomocí composable funkce ExtendedFloatingActionButton. Do těla tohoto tlačítka byly také přidány dvě composable funkce IconButton, které slouží jako tlačítka k potvrzení/zamítnutí smazání notifikací. Tlačítko bylo následně propojeno s mazací funkcí serverové aplikace.

Obrázek 43: Tlačítko ke smazání notifikací po otevření (zdroj: vlastní zpracování)

5.2.4 Možnost úpravy příspěvku

Další navrženou změnou aplikace byla možnost úpravy příspěvku. Do aplikace byla tato funkce přidána podobným způsobem, jako byla implementována pro změnu uživatelských informací. Nejprve byly vytvořeny potřebné funkce a cesty v serverové aplikaci. V aplikaci Kolejka byla poté přidána obrazovka *EditPostScreen*, společně s ViewModelem. Vzhled obrazovky k úpravě příspěvků byl použit stejný, jako byl využit u obrazovky pro přidání příspěvků.

Do obrazovky s detailem příspěvků poté bylo přidáno tlačítko, které umožňuje změnu informací o příspěvku. Toto tlačítko se pochopitelně zobrazuje pouze uživateli, který příspěvek vytvořil.



Obrázek 44: PostDetailScreen, doplněný o editační tlačítko (zdroj: vlastní zpracování)

Kliknutím na editační tlačítko dojde k navigaci do obrazovky *EditPostScreen*, ve které lze upravit příspěvek.

5.2.5 Kolejní novinky

Jedním ze zajímavých návrhů, který jeden z respondentů dotazníku uvedl, bylo přidání obrazovky s kolejními novinkami. Těmito novinkami respondent myslel informace pro ubytované na kolejích – změny v cenách ubytování apod.

Pro implementaci této funkcionality byla přidána datová třída *News*, která reprezentuje článek o novinkách na koleji. Pro tuto datovou třídu byly v serverové aplikaci přidány potřebné funkce: *createNews*, která přidá instanci třídy News do databáze, *deleteNews*, která smaže instanci třídy News, *getNews*, která z databáze načte všechny kolejní novinky a *getNewsById*, která načte informace o jedné z novinek.

Instance třídy *News* by do databáze pochopitelně nemohl přidávat každý z registrovaných uživatelů. Proto byla cesta k přidávání (a mazání) příspěvků zabezpečena, aby tyto funkce mohli provádět jen určití uživatelé (např. autor práce). Zabezpečení bylo umožněno přidáním parametru *accessRights* datové třídě User. Parametr *accessRights* je datového typu Int. V serverové aplikaci poté byla vytvořena enum třída *AccessRights* (Ukázka kódu 115), která definuje dva typy uživatelů: *Regular* (běžný uživatel) a *Admin* (administrátor).

```
enum class AccessRights(val type: Int){
    Regular(type = 0),
    Admin(type = 1)
}
```

Ukázka kódu 115: Enum třída AccessRights (zdroj: vlastní zpracování)

Ověření, zda uživatel, který se pokouší o přidání/smazání příspěvku, poté probíhá následujícím způsobem:

```
if(user.accessRights != AccessRights.Admin.type){
    call.respond(
        HttpStatusCode.Unauthorized
    )
    return@post
}
```

Ukázka kódu 116: Ověření uživatele u přidání instance třídy News
(zdroj: vlastní zpracování)

V ukázce kódu (Ukázka kódu 116) (kód pro přidávání instancí třídy *News*) lze vidět, že ve chvíli, kdy se *accessRights*, které má uživatel uloženy v databázi, nerovnají typu *AccessRights.Admin*, serverová aplikace vrátí chybový kód *Unauthorized*.

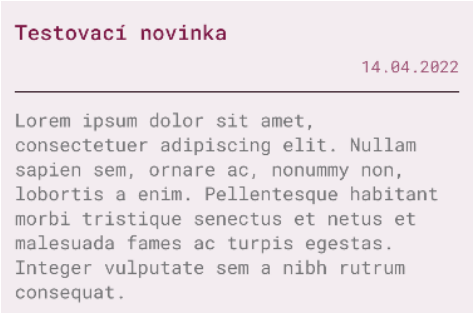
Uvnitř aplikace Kolejka byly implementovány funkce k získání všech novinek (*getNews*) a získání detailu o novince (*getNewsById*). Dále byly vytvořeny obrazovky k zobrazení novinek: zobrazení všech novinek má na starost obrazovka *NewsScreen*, zobrazení detailu o novince obrazovka *NewsDetailScreen*. Na spodní lištu byla poté přidána další navigační lokace, a to sice lokace obrazovky *NewsScreen*.

Obrázek 45: Spodní lišta aplikace s News lokací
(zdroj: vlastní zpracování)

Uvnitř obrazovky *NewsScreen* se zobrazují náhledy jednotlivých novinek. Náhledy představuje composable funkce *NewsComposable*, která zobrazuje titulek novinky, část textu popisku a datum přidání.

Obrázek 46: Composable funkce NewsComposable
(zdroj: vlastní zpracování)

Náhledy lze rozkliknout, čímž dojde k navigaci do obrazovky *NewsDetailScreen*. Na této obrazovce se poté zobrazí detaily o dané kolejní novince.



Obrázek 47: Detail o novince v obrazovce NewsDetailScreen (zdroj: vlastní zpracování)

Novinky nelze přidávat skrze aplikaci, ale je k tomu nutné využít software k odesílání API požadavků – autorem byl použit software **Postman**.

5.3 Další možnosti rozšíření

Do aplikace by pochopitelně bylo možné přidat další funkce, které by zlepšily využitelnost aplikace Kolejka. Jednou z takových funkcí by bylo zakomponování tzv. push notifikací, oznámeních o nové aktivitě v aplikaci. Push notifikace se v operačním systému Android ukazují v horní, systémové liště.

Obrázek 48: Příklad Push notifikace (zdroj: vlastní zpracování)

Pro odesílání push notifikací uživateli je třeba použít službu Firebase Cloud Messaging, kterou poskytuje Google, nebo např. službu OneSignal (která „pod kapotou“ také využívá Firebase Cloud Messaging).

Další funkce, která by mohla být zakomponována do aplikace Kolejka, by byla chatovací funkce. Chatování by bylo možné umožnit s použitím protokolu websockets. Websockets lze konfigurovat pomocí pluginu, který poskytuje KTor, tudíž není třeba žádné služby třetí strany. Dále by bylo vhodné vytvořit např. webové rozhraní pro přidávání novinek, aby nemusel být používán software Postman. Tyto funkce však nebyly do aplikace zahrnuty z časových a znalostních důvodů.

Momentálně nelze aplikaci stáhnout z veřejných zdrojů. K tomu, aby byla aplikace Kolejka veřejně dostupná, bylo by nezbytné publikovat aplikaci skrze oficiální obchod s Android aplikacemi Google Play Store. K publikaci aplikace na Google Play Store je pro autory aplikací nutné si vytvořit vývojářský účet v rámci platformy Google Play Developer Console, která poskytuje všechny nezbytné funkce - vytvoření samotné aplikace pro Google Play Store (což také vyžaduje vygenerování APK souboru v IDE Android Studio, který je nutný pro inicializaci aplikace v Google Play Storu), nastavení plateb (v případě, že aplikace obsahuje tzv. in-app nákupy, tedy nákupy uvnitř aplikace) či deklarování ceny aplikace v rámci Google Play Store.

K tomu, aby byla aplikace Kolejka připravena k publikování, bylo by dále třeba především pečlivě otestovat všechny funkce aplikace (nejen uživatelským testování, ale také jednotkovým testováním), zajistit bezpečnost APK souboru, který by byl publikací aplikace veřejně dostupný (např. pomocí obfuskačních nástrojů, mezi něž patří Proguard či R8) a také zajistit plynulý chod serverové aplikace, čehož by mohlo být dosaženo využitím placených verzí Heroku a MongoDB. Placené měsíční plány těchto platforem budou krátce probrány v další kapitole této práce, která bude věnována zhodnocení vývoje aplikace Kolejka.

V neposlední řadě je třeba podotknout, že pro oficiální vydání aplikace Kolejka by bylo vhodné zajistit možnost používání aplikace i na mobilních telefonech s operačním systémem iOS. Nejlogičtější možností by v případě aplikace Kolejka, která byla vyvinutá jako nativní aplikace pro operační systém Android, bylo vyvinutí druhé nativní verze aplikace Kolejka, avšak pro systém iOS. Takovou aplikaci by bylo možné vytvořit použitím programovacího jazyka Swift.

Druhou možností by bylo využití jednoho z hybridních frameworků pro vývoj mobilních aplikací. Nejpoužívanějšími hybridními frameworky k vývoji mobilních aplikací jsou v dnešní době frameworky Flutter a React Native. Aplikace vytvořené pomocí hybridního frameworku lze poté spustit na mobilních telefonech s OS Android i iOS. Hybridní vývoj má však i své nevýhody, mezi něž patří např. obtížné propojení s funkcemi, jako je Bluetooth či GPS. Hybridní frameworky jsou tudíž vhodné především pro jednodušší aplikace, které nevyžadují komplexní funkcionalitu.

V případě, že by nová verze aplikace Kolejka byla vytvořena pomocí jednoho z hybridních frameworků, a to z důvodů zajištění dostupnosti aplikace pro OS Android i iOS, nativní

vývoj aplikace Kolejka, jenž byl popsán v této diplomové práci, by přišel vniveč. Autor se tudíž domnívá, že v případě rozšíření aplikace Kolejka pro operační systém iOS by bylo vhodnější vyvinout druhou verzi aplikace pomocí jazyka Swift, jak již bylo zmíněno výše.

6 Zhodnocení vývoje

V této části práce bude vývoj aplikace zhodnocen z ekonomického hlediska. Součástí kapitoly bude také hodnocení vývoje pomocí Jetpack Compose a subjektivní srovnání s XML vývojem.

6.1 Ekonomické zhodnocení

Tato část bude věnována ekonomickému aspektu vývoje mobilní aplikace – konkrétně bude věnována nákladům spojených s vývojem mobilní aplikace.

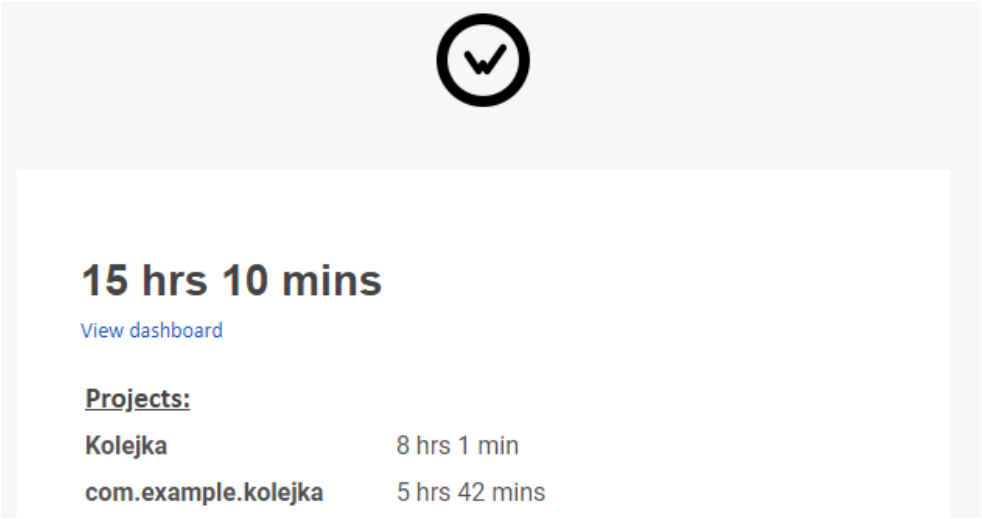
Práci na mobilní aplikaci Kolejka lze rozdělit do 4 fází:

- Návrh a design UI
- Vývoj aplikace pomocí Jetpack Compose
- Vývoj backendu pomocí frameworku KTor
- Testování aplikace

V rámci diplomové práce bude zvolena taxa na jednu člověkohodinu práce na mobilní aplikaci ve výši **400 Kč**. Tato částka byla stanovena po internetovém průzkumu nabídek práce na pozici vývojáře Android aplikací v České republice.

Fáze návrhu a designu UI byla časově nejméně náročná fáze vývoje. Celkově zabrala 2,5 člověkohodiny práce. Jedná se také o hrubý čas, protože alespoň 1 člověkohodina byla věnována prozkoumávání funkcí nástroje Figma. Do výpočtu tedy bude zahrnuto 1,5 člověkohodin práce na vývoji UI. Celkové mzdové náklady na tuto fázi činí **600 Kč**.

Vývoj aplikace pomocí Jetpack Compose zahrnuje implementaci uživatelského rozhraní a propojení aplikace s backendem aplikace. K monitorování hodin, které byly stráveny na vývoji aplikace Kolejka, byl použit plugin Wakatime. Tento plugin byl zmíněn v části 4.1.2 a funguje tak, že zaznamenává aktivní čas práce na projektu. Plugin periodicky zasílá emailové zprávy s reporty o předchozím týdnu práce.



Obrázek 49: Ukázka reportu z pluginu Wakatime
(zdroj: vlastní zpracování)

Čistý čas, který byl stráven na vývoji aplikace, činí 180 hodin. Celkové mzdové náklady na fázi „Vývoj aplikace pomocí Jetpack Compose“ tedy činí **72 000 Kč**. Do tohoto času není započítán čas, kterým byl dále stráven učením se o Jetpack Compose, neboť vývoj pomocí tohoto frameworku byl pro autora zcela nový. Do celkové, odhadované částky je ovšem vhodnější započítávat čistý čas.

Vývoj backendu pomocí frameworku KTor zahrnoval vývoj serverové aplikace. Čas práce byl opět zaznamenáván pomocí pluginu Wakatime. Čistý, celkový čas, který byl stráven na vývoji serverové aplikace, činí 42 hodin. Celkové mzdové náklady na tuto fázi jsou tedy **16 800 Kč**.

Poslední fází vývoje bylo uživatelské testování. Zde autor počítá s taxou na jednu člověkohodinu testování ve výši 200 Kč. Testování aplikace, společně s vyplněním dotazníku, zabralo každému z respondentů přibližně 1 hodinu čistého času. Celkové mzdové náklady na fázi testování tedy činí **2 000 Kč** (10 respondentů).

V rámci vývoje se počítá s tím, že vývojář má všechno potřebné vybavení k dispozici, tudíž z tohoto hlediska se nepřičítají další náklady. Do finální, odhadnuté ceny také nebudou započítány ceny energií.

Dalšími nákladovými položkami by v odhadu byly náklady za provoz serverové aplikace a databáze. Momentálně jsou k běhu aplikace používána bezplatné verze platforem Heroku a

MongoDB Atlas, každopádně při produkčnímu běhu aplikace by s nárůstem uživatelů bylo vhodné investovat do zlepšení infrastruktury.

Z hlediska platformy Heroku by pro produkční běh aplikace byl zvolen platební plán „Standard 2X“, který nabízí server s 1 GB RAM, konstantní běh aplikace (bez uspávání), notifikace při plnění limitu plánu či jednoduchou možnost dalšího škálování aplikace. Tento plán vychází na 50 USD měsíčně, což činí **1 120 Kč** (ke dni, kdy byla tato část zpracována).

MongoDB poté nabízí „Dedicated“ platební plán, který disponuje 10GB místa v úložišti, 2 GB RAM či odladěná, přístupová nastavení. Měsíční taxa za tuto službu je podobná ceně Heroku Standard 2X a činí 57 USD měsíčně, tedy **1 277 Kč**.

Další nákladovou položkou představuje poplatek za mail server. V rámci aplikace Kolejka byl použit e-mail hosting od firmy MyDreams, který poskytuje 10 GB prostoru pro emaily. Měsíční taxa za tento hosting činí 35 Kč. Pro účely aplikace Kolejka je tento hosting dostačující. Poslední položkou nákladů je poté poplatek za vlastní doménu – pro aplikaci Kolejka byla zaregistrována doména *kolejka-app.eu,* a to sice z důvodu, aby verifikační emaily aplikace přicházely uživatelům z oficiálně vypadajícího emailu (např. *verifikace@kolejka-app.eu*). K zaregistrování domény byl použit hosting firmy Wedos, kde domény s koncovkou *.eu* stojí 5,5 Kč za měsíc.

V závěru by bylo také vhodné počítat s náklady na údržbu aplikace. V rámci odhadu ceny bude ve finální ceně aplikace započítána technická podpora při řešení problémů s aplikací. V případě, že by zákazník požadoval další funkce aplikace, taxa za provedené úkony by byla odvozena z množství člověkohodin, které by byly stráveny nad dodatečnou prací na aplikaci.

Celková, odhadnutá cena aplikace je uvedena v následující tabulce:

Tabulka 2: Zhodnocení aplikace Kolejka

Položka	**Cena [Kč]**
Mzdy	90 800
Provoz Heroku	1 120
Provoz MongoDB	1 277
Email hosting	35
Doména	5,5
Celkem	**93 237,5**

(zdroj: vlastní zpracování)

Základní částka za aplikaci by tedy činila **93 237,5 Kč**, s měsíční taxou **2 437,5 Kč** za provoz aplikace. Jedná se o čisté, nezdaněné ceny a do celkových nákladů také není započítána amortizace počítače, na kterém byla aplikace programována.

Při výpočtu ceny aplikace je ovšem nutné podotknout, že zkušený programátor by stejnou aplikaci vyvíjel se značně redukovaným, celkovým časem vývoje. Autor odhaduje, že při dobré znalosti Jetpack Compose a KTor by vývoj zabral až o 50-60 % méně času. Zkušený programátor by ovšem také jistě požadoval vyšší taxu za jednu člověkohodinu.

V poslední kroku této části práce došlo porovnání stanovené ceny 93 237,5 Kč s cenami vývoje aplikací firmami na území České republiky. Z průzkumu několika webových stránek českých vývojářských firem bylo zjištěno, že firmy často rozdělují vyvíjené aplikace do tří skupin:

Tabulka 3: Klasifikace typů aplikací

Jednoduché aplikace	Obecně se jedná o aplikace se statickým obsahem bez vlastního backendu. Cena takových aplikací se většinou pohybuje v rozmezí mezi 150 000 - 200 000 Kč.
Středně pokročilé aplikace	Aplikace střední pokročilosti mají k dispozici např. vlastní backend, možnost registrace a přihlašování, lokální caching informací či push notifikace. Ceny aplikací se střední pokročilostí se pohybují v rozmezí mezi 250 000 - 500 000 Kč.
Komplexní aplikace	Komplexní aplikace poté umožňují např. možnost přehrávání médií, zasílání fotografií mezi uživateli, využití augmentované reality apod. U aplikací, které se považují za komplexní, se poté ceny odvíjí od složitosti požadovaných funkcí aplikace. Ceny za takové aplikace přesahují 500 000 Kč.

(zdroj: vlastní zpracování)

Aplikaci Kolejka by bylo možné klasifikovat jako jednoduchou, až středně pokročilou aplikaci. Vzhledem k tomu, že aplikace Kolejka je dostupná pouze pro operační systém Android, částka 93 237,5 Kč za aplikaci Kolejka by tudíž hrubě odpovídala cenám, které stanovují české vývojářské firmy (ceny, které jsou uvedeny v tabulce, jsou za vývoj aplikace v OS Android i iOS).

6.2 Subjektivní hodnocení vývoje

Tato část práce bude věnována subjektivnímu hodnocení vývoje pomocí Jetpack Compose, v porovnání s XML přístupem. Ačkoliv Kapitola 2 byla věnována srovnání těchto dvou přístupů pro vývoj uživatelského prostředí Android aplikací, uvedené příklady byly velice triviální. Pro tuto část práce bylo tudíž vytvořeno UI hlavní obrazovky s příspěvky, které obsahuje aplikace Kolejka, pomocí XML přístupu vývoje uživatelských rozhraní.

Hlavní obrazovka s příspěvky se skládá ze 3 částí: spodní navigační lišty, seznamu příspěvků a horní navigační lišty. Vzhledem k tomu, že se tato část týká pouze srovnání přístupů tvoření

UI, aplikace vytvořená pomocí XML nebyla propojena se serverovou aplikací a byla použita testovací data.

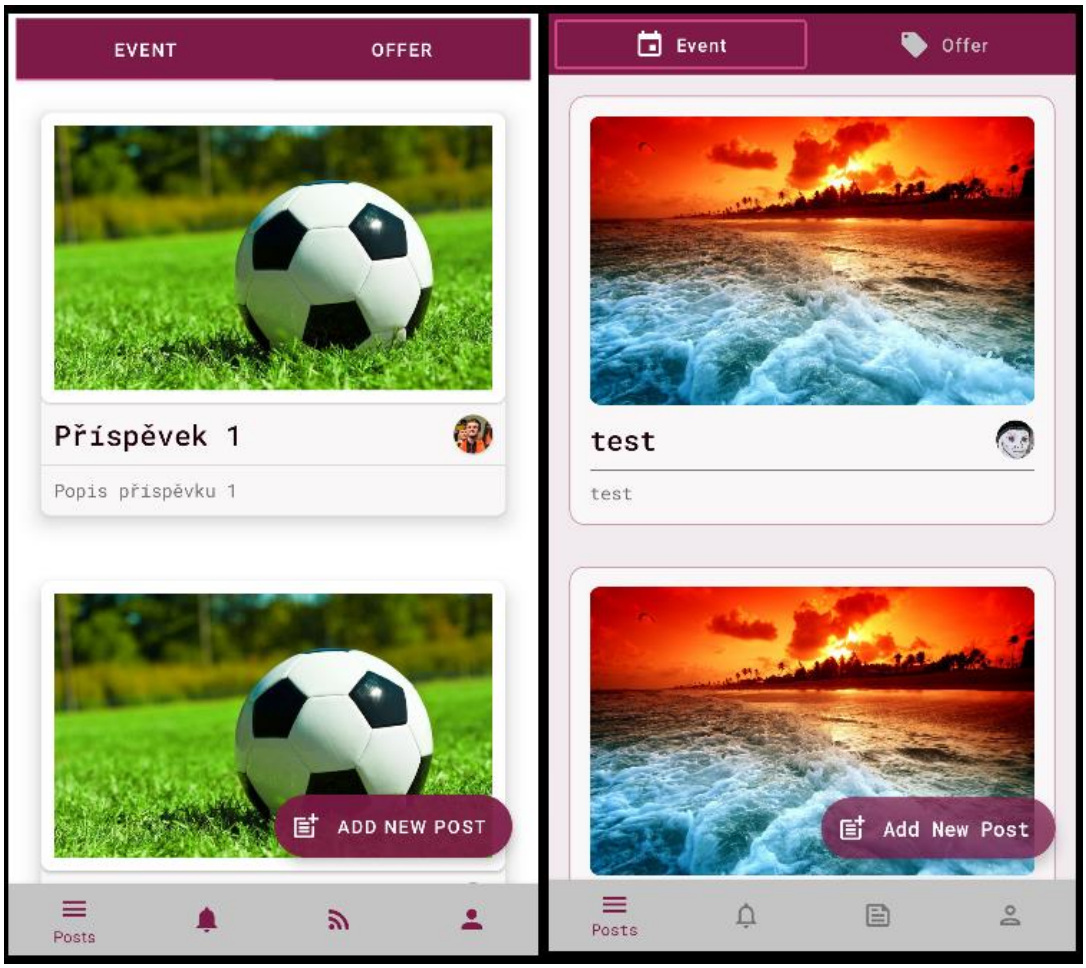
V XML aplikaci byla nejprve vytvořena spodní navigační lišta. Vytvoření bylo poměrně přímočaré – do XML souboru rozložení hlavní aktivity (*MainActivity*) byla přidána View komponenta BottomNavigationView. Dále byl přidán XML soubor pro rozložení, typ *menu*. V tomto souboru se definovaly jednotlivé lokace spodní navigační lišty. Nakonec byla v souboru MainActivity spodní lišta inicializována.

Dále bylo třeba vytvořit seznam s náhledy jednotlivých příspěvků, k čemuž pomohla View komponenta RecyclerView. Pro vytvoření RecyclerView seznamu bylo nejprve vytvořit soubor s rozložením pro náhled. Dále byly vytvořeny třídy typu Adapter a ViewHolder, které slouží ke správné inicializaci seznamu. Nakonec bylo třeba RecyclerView propojit se zdrojem dat (kterým byl v tomto příkladu seznam objektů typu *Post*).

Horní navigační lišta aplikace byla poté vytvořena pomocí View komponent AppBarLayout a TabLayout. Uvnitř komponenty TabLayout bylo třeba definovat položky horní lišty pomocí komponenty TabItem. Pro horní navigační lištu bylo také třeba vytvořit třídu typu Adapter, která umožňuje navigaci mezi obrazovkami, které jsou lokacemi horní lišty.

Do XML aplikace byly také přidány třídy typu Fragment, které představují jednotlivé obrazovky, jež jsou lokacemi spodní navigační lišty (*PostFragment, NotificationsFragment, NewsFragment, ProfileFragment*). Aby mohlo být mezi jednotlivými lokacemi navigováno, byla také přidána navigační komponenta aplikace (skrze XML soubor rozložení, typ *navigation*).

Na následujících dvou obrázcích jsou poté zobrazeny hlavní obrazovky s příspěvky, vytvořeny pomocí XML a Jetpack Compose. Obrazovka zkonstruována pomocí XML je nalevo, Jetpack Compose obrazovka vpravo.



Obrázek 50:Porovnání rozložení XML a Jetpack Compose
(zdroj: vlastní zpracování)

Dále bude hodnocen vývoj jednotlivých částí hlavní obrazovky s příspěvky pomocí XML přístupu.

Spodní navigační lištu lze implementovat snadno a veškeré kroky pro vytvoření lišty jsou intuitivní.

To samé se ovšem nedá říci o vytváření seznamů a horní navigační lišty. Vzhledem k tomu, že seznamy položek jsou v různých aplikacích velmi rozšířenou komponentou, jejich implementace, především pro vývojáře, kteří nemají s vývojem Android aplikací mnoho zkušeností, je dle autora velmi složitá. Ke složitosti přispívá především nutnost tříd typu Adapter a ViewHolder. Adapter třídy pro seznamy (RecyclerView) totiž dědí z abstraktní třídy RecyclerView.Adapter, tudíž je při vytváření adapter třídy nutné implementovat metody RecyclerView.Adapter. Oproti implementaci seznamů pomocí RecyclerView je

využití LazyColumn v Jetpack Compose velice jednoduchou záležitostí, kterou lze vyřešit mnohem menším množstvím kódu.

Pro porovnání, pokud bude brána v potaz pouze délka kódu, která je nutná k implementaci seznamu (bez délky kódu souboru rozložení jedné položky seznamu), RecyclerView zabere v uvedeném příkladu **35 řádků** kódu, zatímco LazyColumn zabere pouhé **4 řádky** (Ukázka kódu 117).

```
LazyColumn{
    items(postList){ post ->
        PostComposable(post = post)
    }
}
```

Ukázka kódu 117: Vytvoření seznamu pomocí LazyColumn
(zdroj: vlastní zpracování)

Implementace horní navigační lišty pomocí XML přístupu byla pro autora také frustrující. Frustrace pochopitelně může pramenit z nedostatku autorových znalostí o XML přístupu, avšak při vytváření horní lišty byly vyzkoušeny 3 různé způsoby, které byly uvedeny v internetových návodech a ani u jedné z implementací nevypadala horní lišta stejně jako u Jetpack Compose lišty. Autor se také pokoušel horní lištu implementovat uvnitř třídy typu Fragment, což vedlo, z neznámých důvodů, k nesprávnému umístění horní lišty na obrazovce (ačkoliv v náhledu rozložení bylo umístění v pořádku). K vytvoření horní navigační lišty bylo navíc opět nutné vytvářet adapter třídu, tentokrát se ovšem jednalo o adapter typu FragmentPagerAdapter. Vytvoření horní navigační lišty v Jetpack Compose se naopak podobá implementaci Jetpack Compose spodní navigační lišty, tudíž pro autora konstrukce horní navigační lišty nedělala problémy – ačkoliv ji vytvářel v rámci aplikace Kolejka pomocí Jetpack Compose poprvé.

Pro další srovnání budou v následující tabulce uvedeny některé číselné údaje o vývoji hlavní obrazovky s příspěvky pomocí XML a Jetpack Compose.

Tabulka 4: Číselné údaje o vývoji hlavní obrazovky aplikace

Metrika	**XML**	**Jetpack Compose**
Délka kódu (znaky)	12 360	8 098
Délka kódu (řádky)	302	225
Počet souborů (tříd, …)	8	6

(zdroj: vlastní zpracování)

Do počtů znaků a řádků byly započítány i prázdné řádky a kód, který generuje vývojové prostředí. V tabulce lze vidět, že Jetpack Compose kód je ve všech ohledech kratší: XML kód má o 52,63 % více znaků než Jetpack Compose, o 34,2 % více řádků a o 33,3 % více souborů. Je však nutné brát v potaz, že se jedná o srovnání na jednom typu příkladu – při implementaci jiných funkcí by se rozdíly lišily.

Dle autora je tedy vývoj pomocí Jetpack Compose mnohem snadnější než vývoj pomocí XML přístupu. Při srovnávání je nutné brát v potaz autorovu krátkou zkušenost s XML vývojem (a krátkou zkušenost s vývojem Android aplikací celkově). Autor se však domnívá, že především pro nové Android vývojáře je Jetpack Compose přívětivější, neboť spousta věcí v JC, ačkoliv jde o čistě subjektivní názor, jednoduše „dává smysl“ a jejich implementace je pro programátory více intuitivní. Autor dále věří, že vývoj UI aplikací pomocí Jetpack Compose pomůže vývojářům při učení dalších deklarativních jazyků, mezi něž patří např. Flutter či React. Např. jazyk Flutter má totiž syntaxi velmi podobnou Jetpack Compose.

Závěr

Tato diplomová práce byla věnována nové, moderní sadě nástrojů k tvorbě uživatelských rozhraní, která nese název Jetpack Compose. V teoretické části byly předvedeny základní postupy pro vytváření uživatelských rozhraní pomocí Jetpack Compose a také byly představeny principy moderní aplikační architektury Android aplikací, k čemuž pomáhá sada knihoven Android Jetpack.

Praktická část práce poté byla věnována popisu vývoje mobilní Android aplikace, jejíž uživatelské rozhraní bylo vytvořeno pomocí Jetpack Compose. Součástí vývoje mobilní aplikace byl také vývoj aplikačního backendu. V praktické části bylo dále popsáno uživatelské testování aplikace, včetně implementace funkcí, které byly navrženy respondenty. Konečně, v závěru práce bylo provedeno zhodnocení vývoje aplikace.

Praktickým výstupem diplomové práce je mobilní aplikace Kolejka, aplikace, která umožňuje studentům, kteří jsou ubytováni na kolejích Harcov, nabízet ostatním studentům předměty (zdarma či za stanovenou cenu) nebo vytvářet s ostatními studenty události. Uživatelským testováním aplikace Kolejka bylo zjištěno, že aplikace má potenciál být mezi ubytovanými používána, což autora práce motivuje dále aplikaci rozšiřovat nad rámec funkcí, které byly popsány v této diplomové práci.

Z hlediska vývoje aplikace je dle autora práce je Jetpack Compose správným krokem, kam by se vývoj Android aplikací měl vyvíjet. Velkou předností této sady nástrojů je programovací jazyk Kotlin, na kterém je aktivně pracováno firmou JetBrains, a který Jetpack Compose plně využívá – k vytvoření Android aplikací tak již stačí pouze jeden programovací jazyk. Firma JetBrains navíc neomezuje sadu nástrojů Jetpack Compose pouze pro vývoj mobilních aplikací – na konci roku 2021 spatřila světlo světa verze 1.0 frameworku Compose Multiplatform, který dále umožňuje vývoj UI pro desktopové a webové aplikace pomocí jazyka Kotlin a hlavních principů Jetpack Compose pro vývoj UI mobilních aplikací – composable funkce, stavy apod.

Prozatím tento framework neposkytuje všechny nutné funkce, které nabízí jiné frameworky (např. React.JS pro vývoj UI web aplikací, či WPF pro UI desktopových aplikací), avšak pokud se firmě JetBrains podaří vyladit nedostatky Compose Multiplatform, tento framework má potenciál stát se skvělým nástrojem k tvorbě multiplatformích aplikací.

Mimo Compose Multiplatform také firma JetBrains pracuje na sadě pro vývoj softwaru Kotlin Multiplatform, která umožňuje vytvářet business logiku aplikací pro obě hlavní platformy – Android a iOS – s pomocí jazyka Kotlin (tuto sadu nástrojů je již možné v současné době použít). Pokud se firmě JetBrains podaří umožnit i vytváření uživatelských prostředí iOS aplikací, vytvořil by tím silného konkurenta jazykům Flutter či React Native, které jsou v současné době hlavními jazyky pro tvorbu hybridních mobilních aplikací pro iOS a Android. Kotlin Multiplatform by byl v případě zmíněného rozšíření také ideální volbou pro vývoj aplikace Kolejka pro iOS.

Z hlediska problematiky vývoje Android aplikací však nelze jasně říci, zda se v následujících letech Jetpack Compose stane primárním způsobem pro vytváření UI Android aplikací. Velké množství dostupných, nativních Android aplikací je vytvořeno za pomoci XML přístupu, a ačkoliv je možné zakomponovat do XML souborů composable funkce (ComposeView), plná transformace aplikací do Jetpack Compose bude pro vývojářské firmy náročný proces, ke kterému dle autora velká část společností nepřistoupí. Jetpack Compose v nynější verzi také obsahuje velké množství funkcí, které jsou stále experimentální a mohou být v budoucnu změněny. Kromě toho Jetpack Compose také mnoho funkcí oproti XML přístupu postrádá a zaostává i v některých výkonnostních aspektech, jak bylo uvedeno v podkapitole 2.3.4.

I přes současné nedostatky Jetpack Compose se autor práce ovšem domnívá, že především pro vývoj nových Android aplikací (nových projektů) začnou vývojáři (a vývojářské firmy) upřednostňovat Jetpack Compose, což s postupem času povede k vyšší popularitě, než jakou má využívání XML přístupu. A to především pokud bude firma Google protěžovat Jetpack Compose do popředí a kontinuálně jej zlepšovat a zrychlovat, jak se tomu zdá nyní být.

Seznam použité literatury

BHARADWAJ, Ashesh, 2019. *What is Architecture? — Android app architecture by example Part 1*. Medium [online]. [cit. 2022-04-07]. Dostupné z: https://medium.com/@asheshb/what-is-architecture-android-app-architecture-by-example-part-1-7ffa2cbfc0df

CALLAHAM, John, 2021. *Android history: The evolution of the biggest mobile OS in the world* [online] [cit. 2022-04-07]. Dostupné z: https://www.androidauthority.com/history-android-os-name-789433/

CALLAHAM, John, 2021. *Google made its best acquisition nearly 16 years ago: Can you guess what it was?* Android Authority [online] [cit. 2022-04-07]. Dostupné z: https://www.androidauthority.com/google-android-acquisition-884194/

COYIER, Chris, 2013. *A Complete Guide to Flexbox | CSS-Tricks - CSS-Tricks* [online] [cit. 2022-04-07]. Dostupné z: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

ELYE, 2021. *Android Jetpack Compose: Remember Made Easy. Mobile App Development Publication* [online]. [cit. 2022-04-07]. Dostupné z: https://medium.com/mobile-app-development-publication/android-jetpack-compose-remember-made-easy-8bd86a48536c

GILSKI, Przemyslaw a Jacek STEFANSKI, 2015. *Android OS: A Review*. *TEM Journal*. **4**(1), 116–120. ISSN 22178309.

GLOBALSTATS, 2022. *Mobile Operating System Market Share Worldwide*. StatCounter Global Stats [online] [cit. 2022-04-07]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide

GOOGLE, 2021. *Android's Kotlin-first approach. Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/kotlin/first

GOOGLE, 2021. *Dependency injection in Android*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/training/dependency-injection

GOOGLE, 2021. *Domain layer*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/guide/domain-layer

GOOGLE, 2021. *Fragments. Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/guide/fragments

GOOGLE, 2021. *Guide to app architecture. Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/guide

GOOGLE, 2021. *Navigation. Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/guide/navigation

GOOGLE, 2021. *State and Jetpack Compose*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/state

GOOGLE, 2021. *ViewModel Overview*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/topic/libraries/architecture/viewmodel

GOOGLE, 2022. *Android 12*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/about/versions/12

GOOGLE, 2022. *Arrangement*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/Arrangement

GOOGLE, 2022. *Compose layout basics | Jetpack Compose*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/layouts/basics

GOOGLE, 2022. *Configure your build*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/studio/build

GOOGLE, 2022. *Create dynamic lists with RecyclerView*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/guide/topics/ui/layout/recyclerview

GOOGLE, 2022. *Drawable*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/reference/android/graphics/drawable/Drawable

GOOGLE, 2022. *Getting started with Android Jetpack*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/getting-started

GOOGLE, 2022. *Graphics in Compose | Jetpack Compose | Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/graphics

GOOGLE, 2022. *Material Theming in Compose | Jetpack Compose | Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/themes/material

GOOGLE, 2022. *Meet Android Studio*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/studio/intro

GOOGLE, 2022. *Modifier*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier

GOOGLE, 2022. *Paging library overview*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/topic/libraries/architecture/paging/v3-overview

GOOGLE, 2022. *Save data in a local database using Room*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/training/data-storage/room

GOOGLE, 2022. *Side-effects in Compose | Jetpack Compose*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/side-effects

GOOGLE, 2022. *Thinking in Compose | Jetpack Compose*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/mental-model

GOOGLE, 2022. *View*. *Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/reference/android/view/View

GOOGLE, 2022. *Material Components and layouts | Jetpack Compose | Android Developers* [online] [cit. 2022-04-07]. Dostupné z: https://developer.android.com/jetpack/compose/layouts/material

HARIDAS, Vishnu, 2020. *Jetpack Compose—A Modern Declarative UI Toolkit for Android. QBurst Blog* [online]. [cit. 2022-04-07]. Dostupné z: https://blog.qburst.com/2020/10/jetpack-compose-a-modern-declarative-ui-toolkit-for-android/

HILL, Pamela, 2020. *Why do we need Jetpack Compose?. There has been much excitement in the… | by Pamela Hill | ProAndroidDev* [online] [cit. 2022-04-07]. Dostupné z: https://proandroiddev.com/why-do-we-need-jetpack-compose-d69a5fd20122

HODGES, Nick, 2019. *What Is Dependency Injection? Medium* [online] [cit. 2022-04-07]. Dostupné z: https://betterprogramming.pub/what-is-dependency-injection-b2671b1ea90a

IBM CLOUD EDUCATION, 2020. *What is Three-Tier Architecture* [online] [cit. 2022-04-07]. Dostupné z: https://www.ibm.com/cloud/learn/three-tier-architecture

JAK PSÁT WEB, 2016. *Padding* [online] [cit. 2022-04-07]. Dostupné z: https://www.jakpsatweb.cz/css/padding.html

JENKOV, Jakob, 2014. *Android Activity* [online] [cit. 2022-04-07]. Dostupné z: http://tutorials.jenkov.com/android/activity.html

JONES, Michael, John BRADLEY a Nat SAKIMURA, 2015. *JSON Web Token (JWT)* [online]. Request for Comments RFC 7519. B.m.: Internet Engineering Task Force [cit. 2022-04-07]. Dostupné z: doi:10.17487/RFC7519

KLIPPENSTEIN, Zach, 2021. *remember { mutableStateOf() } – A cheat sheet. DEV Community* [online] [cit. 2022-04-07]. Dostupné z: https://dev.to/zachklipp/remember-mutablestateof-a-cheat-sheet-10ma

KUMAR, Priyank, 2020. *Understanding MVVM Architecture in Android. The Startup* [online]. [cit. 2022-04-07]. Dostupné z: https://medium.com/swlh/understanding-mvvm-architecture-in-android-aa66f7e1a70b

LARDINOIS, Frederic, 2019. *Google launches Jetpack Compose, an open-source, Kotlin-based UI development toolkit. TechCrunch* [online]. [cit. 2022-04-07]. Dostupné z: https://social.techcrunch.com/2019/05/07/google-launches-jetpack-compose-an-open-source-kotlin-based-ui-development-toolkit/

MASVITA, Tanya, 2022. *Build Single Activity Apps With Jetpack Compose*. *Medium* [online] [cit. 2022-04-08]. Dostupné z: https://betterprogramming.pub/single-activity-apps-with-jetpack-compose-bba4938ad630

MATERIAL DESIGN, 2021j. *Material Design. Material Design* [online] [cit. 2022-04-07]. Dostupné z: https://material.io/components?platform=android

MCGINNIS, Tyler, 2016. *Imperative vs Declarative Programming*. *ui.dev* [online] [cit. 2022-04-07]. Dostupné z: https://ui.dev/imperative-vs-declarative-programming

MEJIA, Robert, 2019. *Declarative and Imperative Programming using SwiftUI and UIKit. Medium* [online]. [cit. 2022-04-07]. Dostupné z: https://medium.com/@rmeji1/declarative-and-imperative-programming-using-swiftui-and-uikit-c91f1f104252

NATESAN, N, 2019. *How to implement Design Pattern – Separation of concerns*. *Default* [online] [cit. 2022-04-07]. Dostupné z: https://www.castsoftware.com/blog/how-to-implement-design-pattern-separation-of-concerns

PAVEL, 2018. *Lekce 9 - Android programování - ConstraintLayout - Vytvoření omezení* [online] [cit. 2022-04-07]. Dostupné z: https://www.itnetwork.cz/android-programovani-constraintlayout-vytvoreni-omezeni

PROQUEST, 2021. *Databáze článků ProQuest* [online]. Ann Arbor, MI, USA: ProQuest. [cit. 2021-09-26]. Dostupné z: http://knihovna.tul.cz

RICHARDSON, Leland, 2020. *Under the hood of Jetpack Compose — part 2 of 2. Android Developers* [online]. [cit. 2022-04-07]. Dostupné z: https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd

SHORE, James, 2006. *James Shore: Dependency Injection Demystified* [online] [cit. 2022-04-07]. Dostupné z: http://www.jamesshore.com/v2/blog/2006/dependency-injection-demystified

TECHOPEDIA, 2020. *What is Android SDK?* - Definition from Techopedia. *Techopedia.com* [online] [cit. 2022-04-07]. Dostupné z: http://www.techopedia.com/definition/4220/android-sdk

SHELOR, Will, 2021. *Measuring Render Performance with Jetpack Compose. Medium* [online] [cit. 2022-05-02]. Dostupné z: https://engineering.premise.com/measuring-render-performance-with-jetpack-compose-c0bf5814933

TECHNICKÁ UNIVERZITA V LIBERCI, 2011. *MANUÁL JEDNOTNÉHO VIZUÁLNÍHO STYLU TECHNICKÉ UNIVERZITY V LIBERCI* [online]. 30. březen 2011. B.m.: TUL. Dostupné z: http://www.ft.tul.cz/document/126

TUTORIALS POINT, 2015. *Android - Fragments* [online] [cit. 2022-04-07]. Dostupné z: https://www.tutorialspoint.com/android/android_fragments.htm

Seznam příloh

Příloha 1: APK soubor aplikace Kolejka

Příloha 2: Zdrojový kód Android aplikace Kolejka

Příloha 3: Zdrojový kód serverové aplikace vytvořené pomocí frameworku KTor