



Zadání diplomové práce

Název: Tvorba UI pomocí Compose Multiplatform
Student: Bc. Filip Trokšiar
Vedoucí: Ing. Petr Šíma
Studijní program: Informatika
Obor / specializace: Softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání:

Pokyny pro vypracování

Cílem práce je analýza frameworku Compose Multiplatform sloužící pro tvorbu UI, které je sdíleno napříč více platformami.

1. Analyzujte Compose Multiplatform framework
2. Porovnejte framework s ostatními možnostmi pro tvorbu UI (Flutter, React Native apod.) a prozkoumejte jeho limitace oproti nativnímu řešení
3. Po konzultaci s vedoucím práce vyzkoušejte framework implementovat na Vámi zvolené aplikaci a zaměřte se na další problémy související s multiplatformním vývojem UI (navigace, lokalizace atd.)
4. Uveďte jaké jsou možnosti testování UI a následně i tyto testy implementujte
5. Zhodnoťte použitelnost tohoto frameworku.

Diplomová práce

TVORBA UI POMOCÍ COMPOSE MULTIPLATFORM

Bc. Filip Trokšiar

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Petr Šíma
10. května 2024

České vysoké učení technické v Praze
Fakulta informačních technologií

© 2024 Bc. Filip Trokšiar. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Trokšiar Filip. *Tvorba UI pomocí Compose Multiplatform*. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Poděkování	vii
Prohlášení	viii
Abstrakt	ix
Seznam zkratek	x
1 Úvod	1
1.1 Cíle práce	2
1.2 Definice multiplatformního vývoje UI	2
1.3 Uplatnění multiplatformního vývoje a sdíleného UI	2
1.4 Důvody multiplatformního vývoje a sdíleného UI	2
2 Analýza	3
2.1 Přehled existujících frameworků	3
2.1.1 React Native	3
2.1.2 Flutter	6
2.1.3 Compose Multiplatform	9
2.2 Porovnání vybraných multiplatformních frameworků	16
2.2.1 Porovnání výkonu	16
2.3 Limitace Compose Multiplatform oproti nativnímu řešení	18
3 Návrh	19
3.1 Výběr aplikace	19
3.2 Případy užití	20
3.3 Specifikace požadavků	23
3.3.1 Funkční požadavky	23
3.3.2 Nefunkční požadavky	24
3.4 Architektura aplikace	25
3.5 Uživatelské rozhraní	29
3.5.1 Drátěné modely	29
3.5.2 Design systém	32
3.5.3 Mockup modely	33
4 Implementace	36
4.1 Založení Compose Multiplatform projektu	36
4.1.1 Gradle	36
4.2 Design systém	38
4.3 Tvorba UI obrazovek	40
4.3.1 Domovská obrazovka	41
4.3.2 Obrazovka <i>Události</i>	42
4.3.3 Obrazovka <i>Parkování</i>	42
4.4 Navigace	44

4.5	Použité technologie	44
4.5.1	Vrstva uživatelského rozhraní	45
4.5.2	Vrstva aplikační logiky	47
4.6	Logika uživatelského rozhraní	49
4.6.1	Stav uživatelského rozhraní	49
4.6.2	UI události	50
4.6.3	ViewModel	51
5	Testování	52
5.1	Možnosti testování UI v Compose Multiplatform	52
5.2	Testovací případy	53
5.3	Implementace testů	54
5.3.1	Testy jednotlivých UI komponent	54
5.3.2	End-to-end testy	55
6	Závěr	56
6.1	Závěr o použitelnosti frameworku	56
6.2	Shrnutí dosažených výsledků	57
6.3	Návrhy pro budoucí vývoj a výzkum	57
A	Snímky obrazovky	58

Seznam obrázků

2.1 React Native Zdroj: [11]	5
2.2 React Native vykreslovací fáze Zdroj: [12]	6
2.3 Flutter architectural layers Zdroj: [14]	8
2.4 Flutter build proces Zdroj: [17]	9
2.5 Compose Multiplatform iOS Zdroj: [26]	10
2.6 UI struktura a její sémantický strom Zdroj: [28]	11
2.7 Možnosti implementace KMP Zdroj: [24]	12
2.8 Hierarchická struktura KMP [33]	14
2.9 Strom závislostí Zdroj: [34]	15
2.10 KMP průzkum Zdroj: [37]	16
2.11 APK/IPA size in megabytes Zdroj: [38]	17
 3.1 Diagram případů užití	22
3.2 Architektura UI vrstvy Zdroj: [46]	26
3.3 Reprezentace vztahu UI a stavu aplikace Zdroj: [48]	27
3.4 Architektura datové vrstvy Zdroj: [46]	28
3.5 Provázanost jednotlivých částí datové vrstvy Zdroj: [54]	29
3.6 Drátěný model obrazovky <i>Domů</i>	30
3.7 Drátěný model obrazovky <i>Události</i>	30
3.8 Drátěný model obrazovky <i>Parkování</i>	31
3.9 Drátěný model obrazovky <i>Více</i>	31
3.10 Ukázka navrženého design systému	32
3.11 Obrazovka <i>Domů</i>	34
3.12 Obrazovka <i>Události</i>	34
3.13 Obrazovka <i>Parkování</i>	35
3.14 Obrazovka <i>Více</i>	35
 A.1 Obrazovka <i>Domů</i>	58
A.2 Obrazovka <i>Domů</i>	58
A.3 Obrazovka <i>Domů</i>	59
A.4 Obrazovka <i>Domů</i>	59
A.5 Obrazovka <i>Domů</i>	60
A.6 Obrazovka <i>Domů</i>	60
A.7 Obrazovka <i>Události</i>	61
A.8 Obrazovka <i>Události</i>	61
A.9 Obrazovka <i>Události</i>	62
A.10 Obrazovka <i>Události</i>	62
A.11 Obrazovka <i>Události</i>	63
A.12 Obrazovka <i>Události</i>	63
A.13 Obrazovka <i>Parkování</i>	64
A.14 Obrazovka <i>Parkování</i>	64
A.15 Obrazovka <i>Parkování</i>	65
A.16 Obrazovka <i>Parkování</i>	65

A.17 Obrazovka <i>Parkování</i>	66
A.18 Obrazovka <i>Parkování</i>	66
A.19 Obrazovka <i>Více</i>	67
A.20 Obrazovka <i>Více</i>	67

Seznam tabulek

2.1 Porovnání rychlosti spuštění ukázkové aplikace na platformě Android	17
2.2 Porovnání rychlosti spuštění ukázkové aplikace na platformě iOS	17
3.1 Pokrytí případů užití funkčními požadavky aplikace	25

Seznam výpisů kódů

2.1 Popis UI komponent pomocí JSX	5
2.2 Popis UI widgetů pomocí jazyka Dart	8
2.3 Popis UI widgetů pomocí jazyka Kotlin	11
4.1 Integrace Compose Multiplatform zásuvného modulu do sestavovacího scriptu	36
4.2 Lib integration	37
4.3 Lib integration	38
4.4 Version katalog	38
4.5 Zadefinování barev	39
4.6 Definice barevných motivů	39
4.7 Definice barevných motivů	40
4.8 Ukázka použití stylu písma	40
4.9 Příklad tvorby UI pomocí frameworku Compose Multiplatform	41
4.10 Implementace posuvného řádku pomocí <code>LazyRow</code>	42
4.11 GoogleMap element	43
4.12 Motiv mapy ve formátu JSON	43
4.13 Ukázka použití navigace založené na záložkách	46
4.14 Composable funkce <code>AsyncImage</code> poskytovaná knihovnou	46
4.15 Konfigurace HTTP klienta	47
4.16 Zaslání požadavku	47
4.17 SQL dotaz pro získání všech novinek v databázi	48
4.18 SQL vygenerovaný dotaz	48
4.19 Nativní databázový ovladač pro platformu <i>desktop</i>	48
4.20 DI databázového ovladače pomocí Koinu	49
4.21 Implementace stavu obrazovky <i>Události</i>	50
4.22 Integrace stavu obrazovkou <i>Události</i>	50
4.23 Použití stavu v aplikaci	50

4.24 Použití stavu v aplikaci	51
4.25 Implementace ViewModelu	51
5.1 Metody pro testování UI komponent	55
5.2 Implementace E2E testu	55

Chtěl bych poděkovat především vedoucímu diplomové práce Ing. Petru Šímovi za metodické vedení práce a také své rodině a přítelkyni za podporu během studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. května 2024

Abstrakt

Magisterská práce se zabývá analýzou frameworku Compose Multiplatform sloužícího k tvorbě multiplatformních uživatelských rozhraní. Framework je nejprve podroben analýze a porovnán s obdobnými aktuálně používanými technologiemi. V další části popisuje návrh prototypové aplikace, na které ukazuje praktickou použitelnost technologie a její implementaci. V závěru shrnuje pravidla uživatelsky přívětivého prostředí užitá v aplikaci a uživatelské rozhraní drátěných modelů. Aplikace je nakonec podrobena testovací fázi a finálnímu zhodnocení použitelnosti frameworku.

Klíčová slova Compose Multiplatform, Kotlin Multiplatform, multiplatformní uživatelské rozhraní, Kotlin, mobilní aplikace

Abstract

The master's thesis deals with the analysis of the Compose Multiplatform framework used to create cross-platform user interfaces. The framework is first analyzed and compared with similar currently used technologies. In the next section, it describes the design of a prototype application to demonstrate the practical applicability of the technology and its implementation. Finally, it summarizes the user-friendly rules used in the application and the user interface of the wireframes. Finally, the application is subjected to a testing phase and a final evaluation of the usability of the framework.

Keywords Compose Multiplatform, Kotlin Multiplatform, cross-platform user interface, Kotlin, mobile application

Seznam zkrátek

KMP	Kotlin Multiplatform
KMM	Kotlin Multiplatform Mobile
UI	User interface
JSX	JavaScript XML
DSL	Domain Specific Language
DI	Dependency Injection
MVVM	Model-View-ViewModel
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
IS	Informační systém
SSOT	Single Source of Truth
SQL	Structured query language
API	application programming interface
DAO	data access object
DTO	data transfer object
IDE	integrated development environment
FR	functional requirement
NFR	Non-functional requirement
UC	use case
JSON	JavaScript Object Notation
TOML	Tom's Obvious, Minimal Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
E2E	End-to-End

Kapitola 1

Úvod

V rámci úvodní kapitoly se práce zabývá otázkami, co to vlastně multiplatformní vývoj je, proč se jím zabývat a jakých zařízení se může týkat. Dále je probrána otázka, v jakých případech se vyplatí multiplatformní uživatelské rozhraní (UI) aplikovat a na jakých konkrétních zařízeních.

V rámci analýzy jsou detailně představeny aktuálně používané frameworky pro tvorbu multiplatformních aplikací včetně samotného frameworku Compose Multiplatform a jeho vazby na technologii Kotlin Multiplatform. Obě tyto technologie jsou následně podrobně rozebrány a nakonec porovnány s aktuálně používanými technologiemi z hlediska výkonu a velikosti samotných aplikací.

Druhá kapitola se věnuje výběru a návrhu aplikace, která bude sloužit k otestovaní použitelnosti technologie Compose Multiplatform. Při fázi návrhu je nejprve přistoupeno k vytyčení případů užití a následně jsou specifikovány funkční a nefunkční požadavky pro vybranou aplikaci. Pro aplikaci je navržena základní architektura, která nejlépe vyhovuje vytyčeným požadavkům a následně již práce přechází k detailnímu návrhu samotného UI. To je nejprve navrženo pomocí takzvaných drátěných modelů a následně převedeno na takzvané mockup modely, které lépe vystihují výslednou podobu UI.

Kapitola implementace popisuje samotné založení multiplatformního projektu pomocí frameworku Compose Multiplatform a implementaci navrženého uživatelského rozhraní pomocí této technologie. Dále se tato kapitola věnuje kritickým UI komponentám jako je použitá navigace a následně také důležitým technologiím použitých pro správné fungování aplikační logiky. Závěr této kapitoly je pak věnován implementované logice uživatelského rozhraní.

Předposlední kapitolou je kapitola věnující se testovaní implementované aplikace, v rámci které jsou nejprve představeny možnosti testování multiplatformního UI, následně jsou tyto testy navrženy a implementovány.

Poslední kapitola je věnována shrnutí celého procesu implementace a konkretizaci následujících problémů. Úplně poslední část této kapitoly se týká celkové evaluace vlastností frameworku Compose Multiplatform a závěrečnému zhodnocení použitelnosti toho frameworku v praxi.

1.1 Cíle práce

Prvním důležitým cílem práce je zanalyzovat použitelnost nově nastupující technologie na poli vývoje multiplatformního UI a otestovat, jaké nové principy přináší. Dále tuto technologii porovnat s ostatními frameworky pro tvorbu multiplatformního UI a pokusit se rozebrat důležité principy, na kterých jsou tyto technologie postaveny.

Dalším podstatným cílem je vytvoření aplikace, která tuto technologii implementuje a zaměření se při tom na slabé stránky, které aktuálně multiplatformní UI provází. Posledním hlavním cílem je vytvořenou aplikaci otestovat a zhodnotit použitelnost daného frameworku v praxi.

1.2 Definice multiplatformního vývoje UI

Technologie sloužící k tvorbě multiplatformních UI umožňují vývojářům vytvářet jednotná uživatelská rozhraní, která mohou být nasazena na různá zařízení jako jsou mobilní zařízení, tablety, televize, hodinky, obrazovky a to buďto v podobě desktopové nebo webové aplikace. Cílem multiplatformního vývoje je tak dosáhnout jednotného uživatelského zážitku bez nutnosti psát a udržovat oddělené kódy pro každou platformu. Nicméně, jak je vidno z předchozího výčtu zařízení, tak nevždy má smysl multiplatformní přístup aplikovat. Tomu, kdy je vhodné aplikovat multiplatformní vývoj, je věnována následující sekce.

1.3 Uplatnění multiplatformního vývoje a sdíleného UI

I přesto, že multiplatformní frameworky umožňují implementaci multiplatformního UI na většinu nejvíce používaných platform jako je Android, iOS, Android TV, tvOS, Wear OS, watchOS nebo Android Automotive, tak vždy je vhodné těchto možností využít. Z pohledu UI mají tyto platformy často jiné velikosti obrazovek, jiné možnosti ovládaní a proto je pravidelně multiplatformní UI a často i veškerá logika implementována konkrétně pro danou platformu nebo typ zařízení. Je proto vždy nutné vědět, jaký typ aplikace bude z pohledu aplikační logiky implementován, jaké budou jeho způsoby užití a na jakých platformách bude daná aplikace implementována. Obecně lze tedy říci, že efektivita implementace multiplatformních aplikací se může dle těchto omezení a požadavků výrazně lišit.

1.4 Důvody multiplatformního vývoje a sdíleného UI

Mezi primární důvody vedoucí firmy a jednotlivce k vývoji multiplatformních aplikací patří především snížení nákladů na vývoj a následně také na údržbu. Dále také jednodušší dosažení konzistentního vzhledu na odlišných platformách nebo například možnost znova používat komponenty UI na různých platformách.

Mezi další důvody může například patřit možnost rychlejších aktualizací, jelikož nové funkce mohou být implementovány jednotně a rychle na všech platformách.

Kapitola 2

Analýza

Tato kapitola je zaměřena na představení aktuálně používaných multiplatformních frameworků a porovnání těchto frameworků s technologií Compose Multiplatform. Kapitola dále shrnuje jejich podstatné rysy a následně slouží k jednoduššímu porovnání a vyhodnocení použitelnosti frameworku Compose Multiplatform. Zároveň je v této kapitole detailněji rozebrán samotný Compose Multiplatform a to jak z pohledu implementace uživatelského rozhraní, tak i z pohledu implementace aplikační logiky, ke které využívá technologii Kotlin Multiplatform.

Po dokončení analýzy obou těchto technologií se práce zaměřuje na porovnání všech těchto frameworků jednak z pohledu architektury a náročnosti implementace, tak z pohledu výkonu aplikací implementovaných pomocí těchto technologií.

Konec této kapitoly je věnován limitacím frameworku Compose Multiplatform oproti nativním řešením, které přímo vyplývají z provedené analýzy.

2.1 Přehled existujících frameworků

Mezi aktuálně nejpopulárnější multiplatformní frameworky jednoznačně patří Flutter a React Native. [1] V následujících kapitolách jsou proto tyto nejpoužívanější frameworky podrobně porovnány s frameworkem Compose Multiplatform a u každého z nich jsou vybrány důležité vlastnosti, které jsou pro tyto frameworky typické.

Jednotlivé frameworky jsou seřazeny postupně od vývojově nejstaršího po nejmladší, což umožňuje lepší náhled na to, jak se frameworky vyvíjely v čase. Pro lepší ilustraci rozdílů mezi vybranými frameworky byla zvolena podobná struktura jednotlivých kapitol, čehož bylo možné docílit díky mnoha společným rysům napříč frameworky.

2.1.1 React Native

React Native byl jedním z prvních frameworků pro tvorbu multiplatformního UI a do jisté míry ovlivnil i ostatní popisované frameworky. Jeho vývoj započal ve společnosti Facebook během interního hackatonu projektu a první jeho oficiálně publikovaná verze vyšla začátkem roku 2015. [2] Nyní se jedná o open-source framework, kde je jeho hlavním cílem umožnit vývojářům vytvářet nativní mobilní aplikace pro platformy Android a iOS z jednoho společného kódu napsaného v jazyce JavaScript nebo TypeScript.

Klíčové vlastnosti React Native

Komponentní architektura

React Native využívá komponentní architekturu, která vývojářům umožňuje vytvářet znovupoužitelné komponenty. [3] Tato architektura je jednak založena na obecných React komponentách, ale zároveň také na React Native specifických komponentách, které se dále dělí na takzvané core komponenty, komponenty vytvořené komunitou či vlastní nativní komponenty. [3]

Deklarativní zápis UI

React Native, stejně jako React pro webové aplikace, využívá pro zápis UI deklarativní způsob, při kterém využívá JSX (JavaScript XML) syntaxe k popisu struktury UI komponent. [4] Tento způsob zápisu začal na mobilních platformách růst na popularitě právě díky Reactu Native, který po svém uvolnění v roce 2013 defacto nastartoval éru deklarativního zápisu UI na mobilních platformách. [5]

Fast Refresh

Fast Refresh je funkce, která vývojářům umožňuje okamžitě vidět výsledky provedených změn v kódu bez nutnosti znova sestavení aplikace. [6]

JavaScript/TypeScript

Aplikační logika v React Native se píše v jazyce JavaScript nebo TypeScript, což usnadňuje snadnou integraci s existujícími webovými technologiemi. [7]

Rozsáhlá komunita

React Native má rozsáhlou komunitu vývojářů, což vede k bohatému ekosystému třetích stran, včetně mnoha dostupných knihoven a modulů. [8]

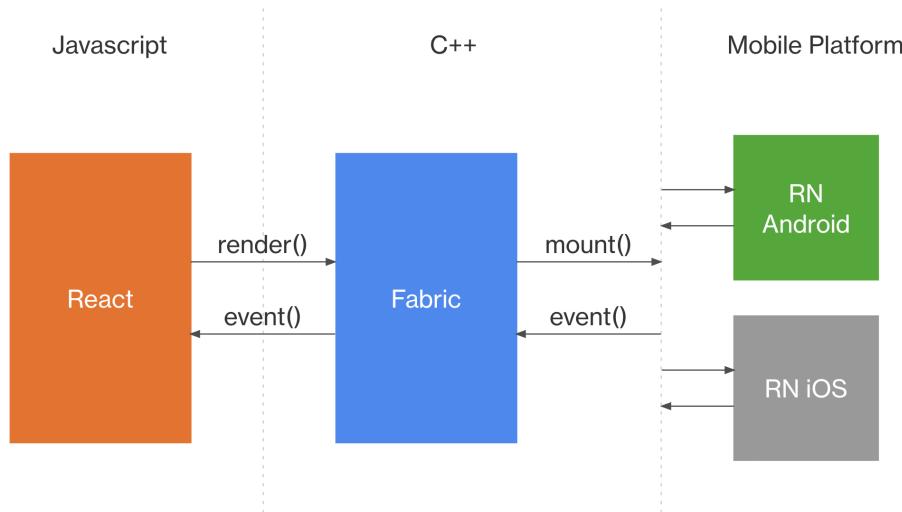
Expo framework

Pro ještě snazší start vývoje poskytuje React Native takzvaný Expo framework, který zjednodušuje proces vývoje a umožňuje rychlé prototypování. [9]

Architektura frameworku React Native

React native do roku 2022 využíval architekturu založenou na návrhovém vzoru Bridge, který spojoval kód napsaný v JavaScriptu s nativním kódem určeným pro danou platformu. Tyto dva celky byly spuštěny na souběžných vláknech a komunikovaly spolu pomocí zasílaní serializovaných zpráv. Časem se ale ukázalo, že se tato komunikace a další její charakteristiky stávají úzkým hrdlem celého systému, a byla proto od verze 0.68 nahrazena JavaScriptovým rozhraním zvaným JSI. [10]

JSI nového JavaScriptu umožňuje držet referenci na C++ objekty a volat nad nimi potřebné metody. [10] Toho využívá nový renderovací systém zvaný *Fabric*, který frameworku napomáhá sjednotit renderovací logiku prováděnou v C++ a zlepšit tak interoperabilitu s nativními platformami.



■ **Obrázek 2.1** React Native Zdroj: [11]

Jak je vidět na obrázku 2.1, tak *Fabric* de facto plní funkci prostředníka, který převádí React komponenty na nativní komponenty pro každou platformu.

Pro lepší představu, jak dané komponenty vypadají, slouží následující ukázka kódu 2.1.

■ **Výpis kódu 2.1** Popis UI komponent pomocí JSX

```

function MyComponent() {
  return (
    <View>
      <View
        style={{backgroundColor: 'red', height: 20, width: 20}}
      />
      <View
        style={{backgroundColor: 'blue', height: 20, width: 20}}
      />
    </View>
  );
}
// <MyComponent />

```

Na ukázce kódu 2.1 jsou aplikovány některé z nejpoužívanějších core komponent, které se při psaní React Native aplikací používají. [3] Jak již bylo zmíněno dříve, tak tyto komponenty jsou následně převedeny na nativní komponenty pro každou platformu a to, jakým způsobem *Fabric* dané komponenty převádí, se dá rozdělit do následujících tří na sebe navazujících fází: [12]

Render

Během této fáze se z jednotlivých komponent (React Elementů) sestaví strom elementů v JavaScriptu (viz levá část obrázku 2.2) a nad tímto stromem se následně spustí rekurzivní redukce, při které dojde k vytvoření nového stromu takzvaného React Shadow Tree (viz prostřední část

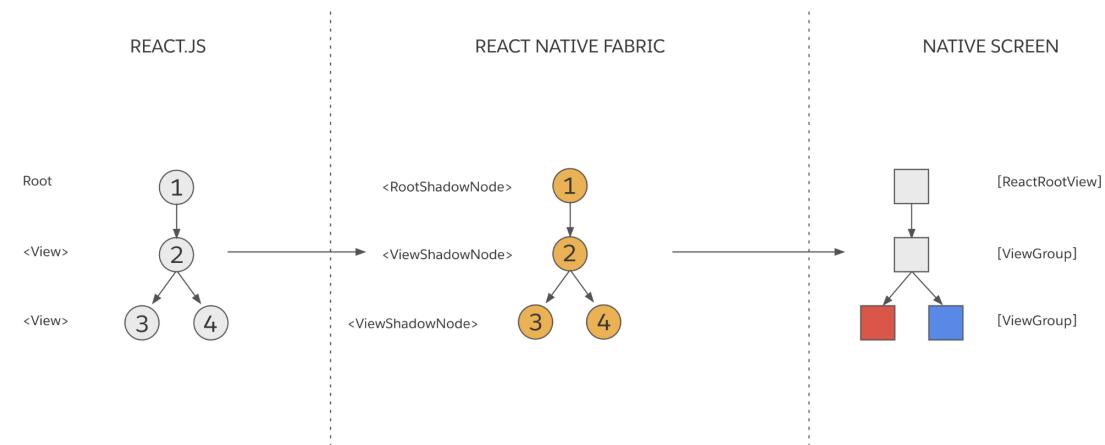
obrázku 2.2). [12] Ten se skládá z React Shadow Nodes, které reprezentují objekty v C++, a tím přechází tato renderovací fáze do další fáze zvané commit.[12]

Commit

V rámci této fáze je pro každý React Shadow Node vypočítána jeho pozice a velikost na koncovém zařízení, a díky tomu může renderovací systém přejít k poslední fázi zvané Mount. [12]

Mount

Během této poslední fáze dojde k transformaci *React Shadow Tree* na *Host View Tree* (viz pravá část obrázku 2.2) a to tak, že každý *React Shadow Node* se transformuje na jeho ekvivalent v nativní podobě. [12] Čili například na platformě Android se <*ViewShadowNode*> přetrasformuje na android.view.ViewGroup. [12]



■ Obrázek 2.2 React Native vykreslovací fáze Zdroj: [12]

2.1.2 Flutter

Flutter je open-source softwarový toolkit pro vývoj uživatelských rozhraní (UI). [13] Za vývojem stojí společnost Google a je určený k vytváření nativně komplikovaných aplikací pro mobilní zařízení, web a desktop z jednoho zdrojového kódu. [13] Byl vydán v roce 2017 a získal značnou popularitu mezi vývojáři díky svému snadnému použití, flexibilitě a schopnosti tvorby UI.

Klíčové vlastnosti Flutteru

UI založené na widgetech

Flutter využívá reaktivně deklarativní UI založené na widgetech. [14] Widgety jsou základními stavebními bloky Flutter aplikací představující vše od strukturálních prvků po stylistické komponenty. [15]

Hot Reload

Další z významných funkcí Flutteru, která významně zrychluje vývojový proces a zvyšuje produktivitu je funkce *Hot Reload*. Během vývoje běží Flutter aplikace na virtuálním počítači (Dart VM), který díky této funkci umožňuje okamžitě vidět provedené změny v kódu bez nutnosti úplné rekompilace aplikace. [16] Teprve pro fázi vydání jsou Flutter aplikace komplikovány přímo do strojového kódu a to i v případě, že už jde o instrukce Intel x64 nebo ARM, případně do JavaScriptu, pokud jsou cíleny na web. [17]

Jeden zdrojový kód pro více platform

S Flutterem mohou vývojáři psát jeden zdrojový kód pro obě platformy Android a iOS, což snižuje dobu vývoje a úsilí vynakládané na údržbu. Flutter také rozšířil svou podporu pro cílení webových a desktopových aplikací, umožňující širší dosah s minimálními změnami kódu. [18]

Rozsáhlá sada widgetů

Flutter poskytuje komplexní sadu přizpůsobitelných widgetů, které usnadňují vytváření složitých a vizuálně atraktivních uživatelských rozhraní. Tyto widgety zahrnují vše od základních tlačítek a textových polí až po pokročilé komponenty jako jsou grafy a animace. [19]

Programovací jazyk Dart

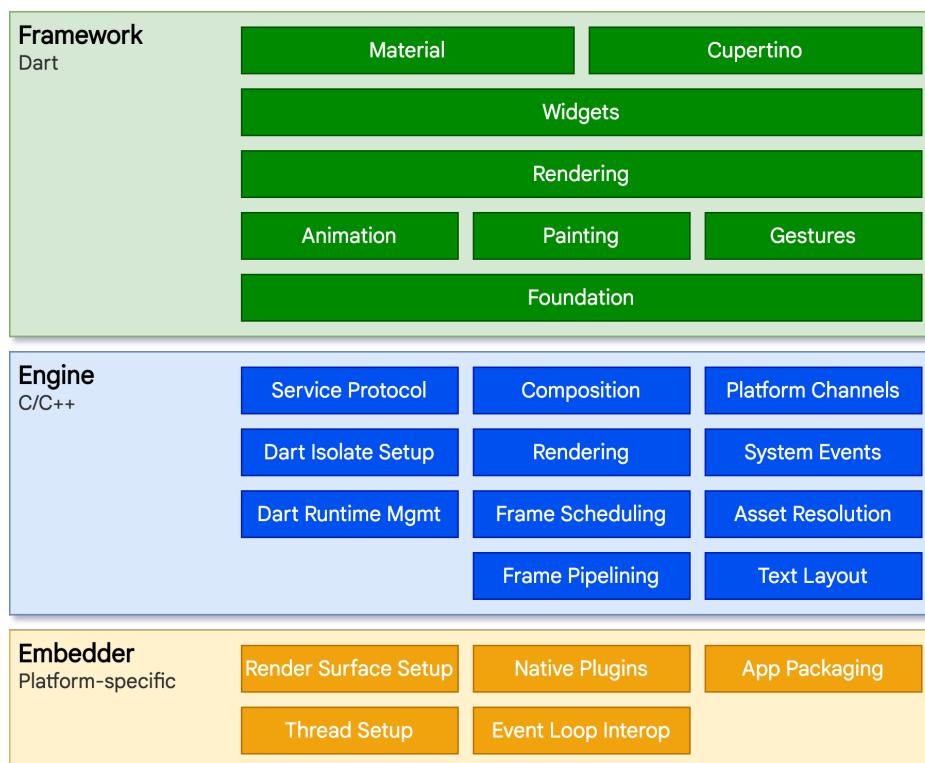
Aplikace vytvořené pomocí frameworku Flutter jsou psány v jazyce Dart - moderním a objektově orientovaném programovacím jazyce vyvinutém společností Google. Dart je navržen pro optimální výkon a produktivitu, což ho činí vhodným pro mobilní a webový vývoj. [20]

Způsob renderovaní UI

Na rozdíl od React Native Flutter nepoužívá platformě specifické komponenty koncových zařízení, ale veškeré UI komponenty (widgety) renderuje pomocí vlastního renderovacího enginu. [21]

Architektura frameworku Flutter

Jak je vidět na obrázku 2.3, tak architektura Flutteru je rozdělena do tří hlavních vrstev. [17] První z nich, embedder, je vrstva, která umožňuje integrovat Flutter do konkrétních platform, jako je Android, iOS, desktop nebo web. Každý embedder obsahuje platformě specifický kód, který je potřebný pro spuštění Flutteru na dané platformě. Engine je další vrstva starající se o vykreslování grafiky, kdykoli, kdy je potřeba vykreslit nový snímek. [17] Framework je vrstva, která je používána vývojáři k vytváření uživatelských rozhraní a definování chování aplikace. Obsahuje hotové widgety, funkce pro manipulaci s UI a další nástroje pro vývojáře. [17]



■ **Obrázek 2.3** Flutter architectural layers Zdroj: [14]

Mezi hierarchicky poslední a neméně důležitou část tohoto frameworku patří platformě specifické knihovny Material and Cupertino. Tyto knihovny jsou následně využívány widgety k implementaci konkrétního designu systému. Díky tomu je možné uživateli navodit nativní pocit z dané aplikace.

Z pohledu UI je důležitým prvkem právě Flutter framework, který zároveň definuje, jak spolu jednotlivé widgety interagují.

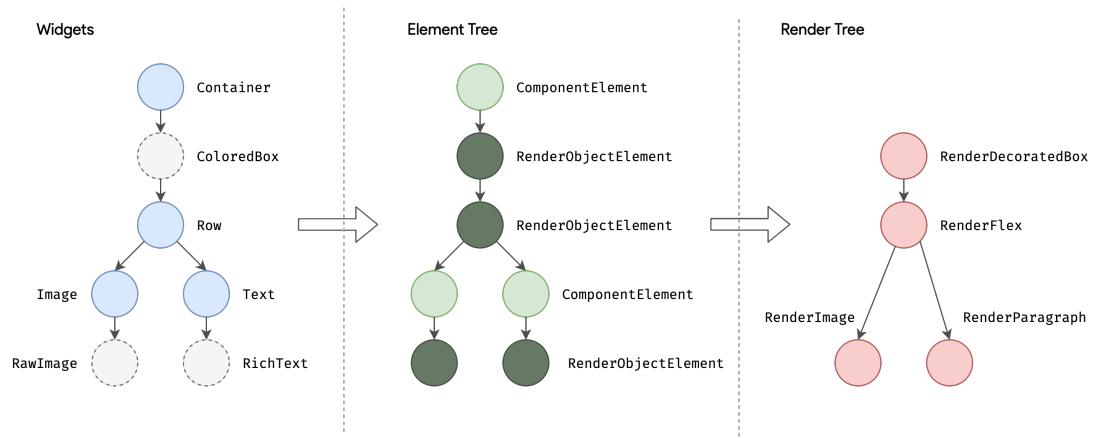
Widget je ve Flutteru základní stavební blok pro tvorbu uživatelského rozhraní. [15] V následující ukázce kódu 2.2 je pro příklad použito několik základních widgetů jako je *Image* nebo *Text*, taktéž layout widget zvaný *Container* a *Row* pro organizaci a rozložení vnořených widgetů na obrazovce.

■ **Výpis kódu 2.2** Popis UI widgetů pomocí jazyka Dart

```
Container(
  color: Colors.blue,
  child: Row(
    children: [
      Image.network('https://www.example.com/1.png'),
      const Text('A'),
    ],
),
);
```

Když Flutter potřebuje vykreslit tento blok, zavolá metodu *build()* a ta vrátí podstrom widgetů, které následně vykreslí uživatelské rozhraní na základě aktuálního stavu aplikace. [17]

Během fáze sestavování překládá Flutter widgety vyjádřené v kódu (například kód 2.2) do odpovídajícího stromu elementů viz obrázek 2.4, přičemž každý widget má přiřazený jeden element. Každý prvek pak představuje určitou instanci widgetu v daném umístění stromové hierarchie. [17]



■ Obrázek 2.4 Flutter build proces Zdroj: [17]

2.1.3 Compose Multiplatform

Compose Multiplatform je framework sloužící k tvorbě uživatelských rozhraní použitelných na vícero platformách, za jehož vývojem stojí společnost JetBrains. [22] Je založen na toolkitu zvaném Jetpack Compose, který je aktuálně doporučovaný k tvorbě nativních uživatelských rozhraní na platformě Android. [23] Podporuje platformy jako Android, iOS (Alpha), Windows, MacOS, Linux a Web (experimentální). [22]

Klíčové vlastnosti Compose Multiplatform

Deklarativní zápis UI

Používá deklarativní syntaxi pro popis uživatelského rozhraní. [24]

Jednotný kód pro různé platformy

Umožňuje sdílet kód pro Android, iOS, web i desktop. [22]

Snadné migrace díky postupné integraci

Díky KMP je možné postupně implementovat jednotlivé části aplikace i do již existujících aplikací s minimálním rizikem oproti ostatním multiplatformním technologiím. [25]

Znovupoužitelnost Kotlin kódu

Díky KMP je možné použít některé části kódu z již existujících Android aplikací i na ostatních platformách.

Programovací jazyk Kotlin

Frontendová i backendová část aplikace je psána v jazyce Kotlin pro bezproblémovou integraci se serverovou částí.

Podpora od JetBrains

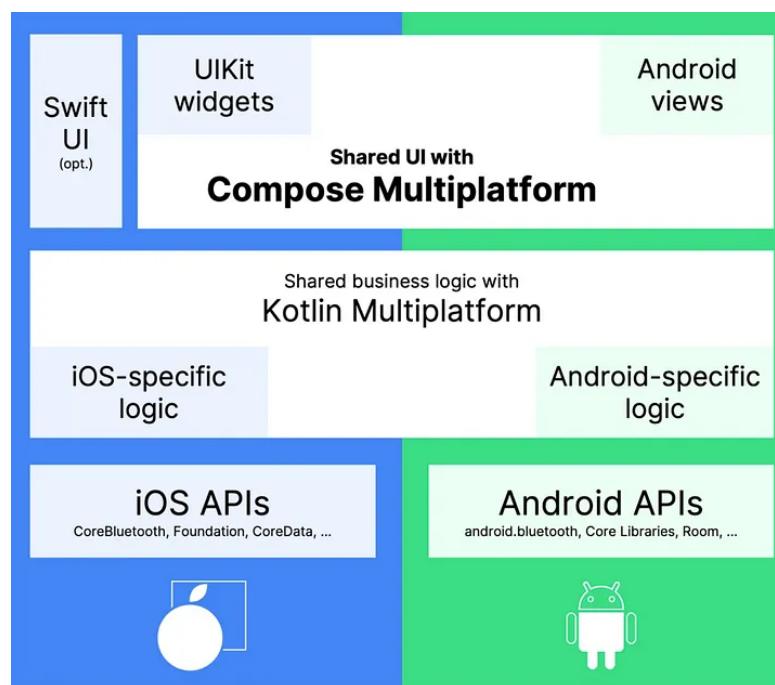
Poskytuje stabilní podporu od vývojářského týmu JetBrains.

Architektura frameworku Compose Multiplatform

Jelikož je framework Compose Multiplatform z velké části založen na frameworku Jetpack Compose [22], tak v této kapitole bude probírána právě architektura toho frameworku.

V porovnání s ostatními dříve zmíněnými frameworky slouží pouze tento k tvorbě multiplatformního UI. Z toho důvodu je převážně používán s toolkitem Kotlin Multiplatform, který dané aplikaci poskytuje i multiplatformní aplikační logiku. [25] Jelikož se jedná o základní komponentu, bez které by Compose Multiplatform nevznikl, je tomuto toolkitu věnována celá kapitola 2.1.3.1.

Pro lepší ukázku toho, z jakých částí se typická multiplatformní mobilní aplikace skládá, slouží následující obrázek 2.5, který vizualizuje propojenosť frameworku Compose Multiplatform s toolkitem Kotlin Multiplatform (KMP). Dále obrázek vizualizuje vztah platformě specifických knihoven pro tvorbu UI jako je SwiftUI k frameworku Compose Multiplatform. Compose Multiplatform lze nicméně použít i s technologiemi pro tvorbu UI jako je UI kit pro platformu iOS nebo Android views pro platformu Android. Každopádně, od těchto technologií je postupně upouštěno a přechází se k deklarativním frameworkům jako je Jetpack Compose nebo SwiftUI.



■ Obrázek 2.5 Compose Multiplatform iOS Zdroj: [26]

Co se týče samotného UI kódu, tak ten se skládá z jednotlivých funkcí označených anotací `@Composable`, která v těle obsahuje další *Composable* funkce (viz výpis kódu 2.3) jako jsou například `Column` pro strukturování vnořených elementů do sloupce nebo `Text` pro zobrazení textu na obrazovce.

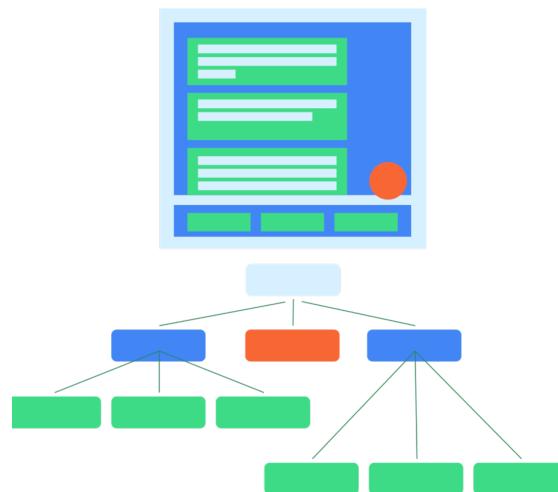
■ **Výpis kódu 2.3** Popis UI widgetů pomocí jazyka Kotlin

```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
```

Takto strukturovaný kód může být následně frameworkem vykreslen na obrazovku, k čemuž dochází v těchto třech následujících fázích:

Composition

Během této fáze Jetpack Compose vytvoří stromovou strukturu reprezentující UI komponenty, které mají být vykresleny.[27] Tato stromová struktura je tvořena na základě do sebe zanořených Composable funkcí, které představují jednotlivé prvky uživatelského rozhraní. Lepé tutu skutečnost reprezentuje obrázek 2.6, na kterém je vidět, že nejspodnejší vrstvu UI reprezentuje kořenový uzel sémantického stromu a následující vnořené prvky jsou jeho potomky.



■ **Obrázek 2.6** UI struktura a její sémantický strom Zdroj: [28]

Layout

V této fázi se komponentám určí na jakých pozicích se mají vykreslit a jakou mají mít šířku a výšku. Compose během této fáze prochází strom UI komponent tak, že nejprve změří velikost svých potomků (pokud existují) a následně se na základě těchto měření rozhodne o vlastní velikosti. Nakonec umístí každého potomka relativně ke své pozici.[27]

Díky použití tohoto algoritmu je k prostupu celého stromu zapotřebí navštívit každý uzel pouze jednou, což je výhodné, jelikož s přibývajícími uzly roste čas potřebný k průchodu celého stromu pouze lineárně. [27]

Drawing

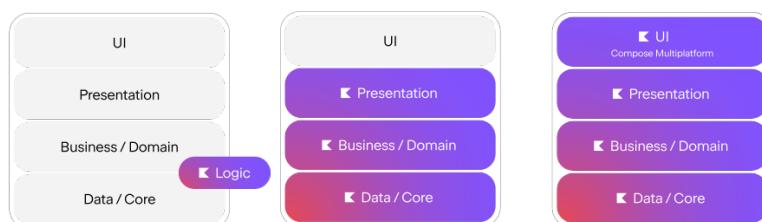
V rámci poslední fáze dochází k samotnému vykreslení komponent na obrazovku koncového zařízení. [27] Compose během této fáze znovu postupně prochází strom UI elementů od kořene

až do listů a každý element tohoto stromu vykreslí sám sebe na pozici určenou během Layout fáze. [27]

2.1.3.1 Kotlin Multiplatform

Kotlin Multiplatform je často základním kamenem pro tvorbu multiplatformních aplikací založených na Compose Multiplatform a to z toho důvodu, že díky KMP je možné implementovat multiplatformní aplikační logiku, která může doplňovat multiplatformní UI poskytované frameworkem Compose Multiplatform. [24] Důležitým rozdílem oproti ostatním multiplatformním frameworkům je právě ta možnost, kterou KMP nebo případně Compose Multiplatform vývojářům poskytuje.

Jelikož se jedná o SDK, umožňuje vývojářům implementovat multiplatformní funkcionality postupně, bez nutnosti implementovat v Kotlinu celé vrstvy aplikací. Díky tomu dává vývojářům možnost sdílet napříč platformami jen ty části kódu, které mají největší smysl implementovat pro veškeré platformy a zbylé části kódu psát v nativním jazyce pro danou platformu. [25] Lépe je tato skutečnost ilustrována na následujícím obrázku 2.7, na kterém jsou vidět různé možnosti, jakými může být KMP na daných platformách implementován.



Obrázek 2.7 Možnosti implementace KMP Zdroj: [24]

Zároveň je tento obrázek ukázkou toho, jak může probíhat migrace z již naimplementované nativní aplikace na aplikaci multiplatformní. Nejprve je tedy možné implementovat jen malou část aplikační logiky pro vícero platforem a postupem času přesouvat do multiplatformní části aplikace například i celé vrstvy starající se o prezentační, datovou nebo jinou aplikační logiku. [24]

Aby k takové migraci mohlo dojít, je zapotřebí, aby pro technologie používané v původní nativní aplikaci existovaly multiplatformní knihovny nebo alespoň jejich ekvivalent.

Multiplatformní knihovny

Společnost JetBrains ve svém článku z konce roku 2023 zmiňuje, že existuje již přes 1500 KMP knihoven s tím, že ještě v roce 2020 jich bylo pouze několik málo desítek. [29]

Právě počet a vyspělost těchto knihoven se podle společnosti, které již KMP používají, jeví jako zatím jedna z nejslabších stránek KMP (viz sekce *Aktuální použití KMP v praxi*).

V případě, že žádná vhodná knihovna není k dispozici, tak je zde stále možnost implementovat danou knihovnu nativně čehož je možné docílit díky funkcionalitě jazyka Kotlina zvané *expected* a *actual* deklarace.

Expected a actual deklarace

Deklarace *expected* a *actual* umožňují přistupovat k platformě specifickým API z Kotlin Multiplatform modulů. [30] Nejprve je ve společné části potřeba vytvořit například třídu nebo funkci, u které chceme, aby její obsah byl implementován typicky pomocí platformě specifických knihoven a tuto část společného modulu označit klíčovým slovem `expect`. [30] Její implementaci

následně provést v rámci platformě specifického kódu. Tato implementace musí být provedena pod stejným názvem a ve stejném balíku jako deklarace ve společném modulu a musí být označena klíčovým slovem `actual`. [30]

Během komplikace pak Kotlin komplilátor každou deklaraci s klíčovým slovem `expect` sjednotí s příslušnou `actual` deklarací pro danou platformu a vygeneruje z ní jednu deklaraci obsahující `actual` implementaci. [30]

Struktura KMP projektů

Jak již bylo zmíněno v úvodu, tak jedním z hlavních cílů multiplatformního vývoje je mít jeden kód, který by bylo možné zkompilovat tak, aby byl spustitelný na všech požadovaných platformách. To však u KMP není vždy možné, a proto je nutné zvlášť rozdělit kód, který je určen pro všechny platformy a kód, který je platformě specifický.

Společný kód

Jedná se o kód sdílený mezi různými platformami a měl by být seskupen v adresáři `commonMain`. [31] Kotlin komplilátor na základě zdrojových kódů umístěných v tomto adresáři generuje sadu binárních souborů specifických pro danou platformu. [31] Při komplikaci tak může z téhož kódu vygenerovat několik souborů jako například soubory pro *Java Virtual Machine (JVM)*, anebo spustitelné soubory pro nativní platformu. [31]

Nicméně i pro jednotlivé platformy existují zařízení, která například využívají specifickou architekturu, a proto je třeba pro tato zařízení zkompilovat tento kód jiným způsobem. K této komplikaci KMP používá technologii Kotlin/Native, díky které je možné sestavovat kód napsaný v jazyce Kotlin do nativních binárních souborů, a ten díky tomu může běžet i bez JVM. [32]

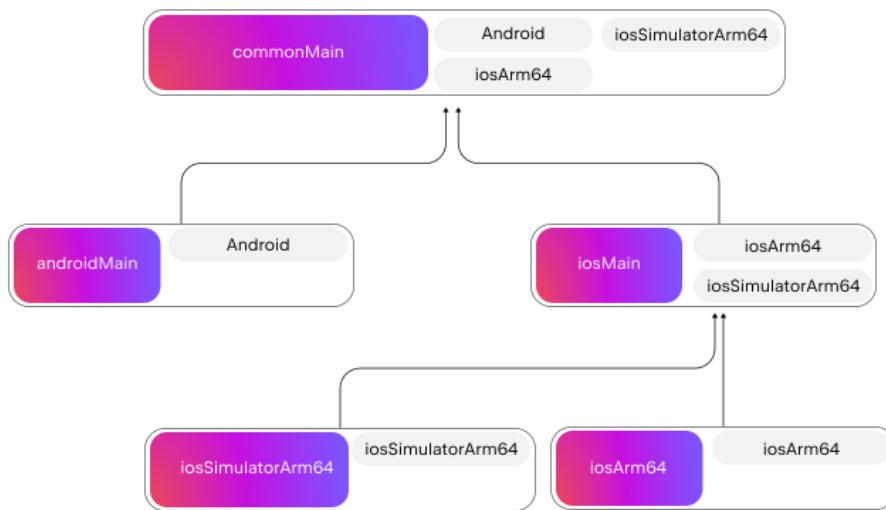
Pro ukázkou toho, jaká zařízení jsou technologií Kotlin/Native podporovaná, slouží následující obrázek 2.8



■ **Obrázek 2.8** Hierarchická struktura KMP [33]

Při této vizualizaci veškerých podporovaných zařízení se ale ukazuje, že rozřazení kódu do vhodných balíčků je pro orientaci v multiplatformním projektu založeném na KMP klíčové. Implementace kódu pro každé jednotlivé zařízení by byla zbytečná, jelikož velké části kódu a to i těchto platformě specifických, mohou být součástí kódu společného. Nesdílení tohoto kódu s ostatními platformami by tak zapříčinilo duplikaci velké části kódu.

Aby se zdvojení zabránilo, je vhodné aplikovat dříve zmíněné hierarchické struktury a využít takzvané *source sets*, které slouží pro jakési seskupení kódů specifických pro konkrétní zařízení nebo platformy. Pro lepší představu o tom, jak jsou jednotlivé *source sets* definovány, slouží obrázek 2.9, který vizualizuje celkem pět zadefinovaných *source sets* a to konkrétně **commonMain**, **androidMain**, **iosMain**, **iosSimulatorArm64** a **iosArm64**. V rámci těchto zdrojových sad mohou být následně implementovány platformě specifické knihovny, čemuž se detailněji věnuje kapitola *Gradle 4.1.1* v implementační části práce.



Obrázek 2.9 Strom závislostí Zdroj: [34]

Ze struktury těchto zdrojových sad zároveň vyplývá koncová struktura projektu, která ve většině případů obsahuje adresář pro sdílený kód (`commonMain`), který smí obsahovat pouze kód a případné knihovny. Tyto knihovny a kódy jsou multiplatformní a pokrývají tak veškeré platformy používané v projektu.

Dále se adresáře mohou dělit podle vybrané platformy jako například `AndroidMain`, `iosMain`, které již mohou být závislé na pouze platformě specifický knihovnách.

Aktuální použití KMP v praxi

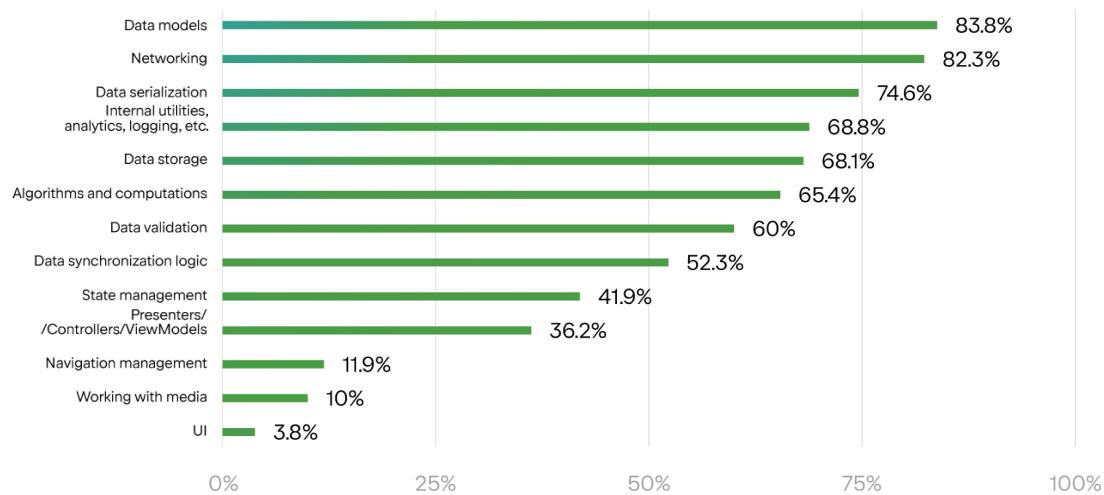
I přesto, že je KMP ve stabilní verzi teprve od listopadu 2023 [29], tak tuto technologii používá již několik světově známých firem v produkčním nasazení, jsou to například firmy Forbes či McDonald's. Forbes uvádí, že právě díky KMP dokáží sdílet až 80 % aplikační logiky napříč platformami Android a iOS. [35] Druhý ze jmenovaných McDonald's jako hlavní výhody uvádí jednodušší testování. [36]

Naopak mezi největší výzvy do budoucna, které obě společnosti ve svých souhrnech uvádějí, patří adaptace na KMP prostředí (především iOS vývojářů) nebo nedostatečné množství potřebných multiplatformních knihoven (případně jejich nedostatečná vyspělost). [35] [36]

Pro upřesnění je ještě potřeba zmínit, že veškeré vyjmenované společnosti využily pouze možnosti sdílení logiky a nepoužívají tak sdílené UI pomocí Compose Multiplatform. To se ukázalo i na průzkumu, který provedla společnost JetBrains v roce 2021, kde pouze 3,8 % respondentů odpovědělo, že byli schopni sdílet UI ve svých aplikacích.

Pro ukázkou veškerých částí, které respondenti sdíleli ve svých aplikacích napříč platformami, slouží obrázek 2.10, který jednoznačně ukazuje, že mezi nejčastěji sdílené části aplikací patří datová a síťová vrstva.

What parts of your code were you able to share between platforms?



■ Obrázek 2.10 KMP průzkum Zdroj: [37]

2.2 Porovnání vybraných multiplatformních frameworků

V rámci této kapitoly budou porovnány vybrané multiplatformní frameworky z pohledu výkonu a velikosti jednotlivých aplikací.

2.2.1 Porovnání výkonu

Mezi klíčové parametry, kvůli kterým většina firem upřednostňuje vývoj nativních aplikací, patří především jejich výkon. Z toho důvodu se následující podkapitola věnuje právě porovnání výkonu a dalších parametrů multiplatformních aplikací s nativními verzemi aplikací.

Pro testování byly použity toolkity pro tvorbu nativního UI pro platformu Android (Jetpack Compose) a iOS (SwiftUI) v porovnání s multiplatformním frameworkem Compose Multiplatform a aktuálně nejpoužívanějším multiplatformním frameworkem zvaném Flutter. [1]

Mezi hlavní testované parametry patřil čas nastartování testované aplikace a její velikost.

Ukázková aplikace

Pro porovnání důležitých parametrů byla pro test vytvořena jednoduchá aplikace s jednou obrazovkou, která načetla obrázky z veřejného API a zobrazila je v horizontálním seznamu. Na každý obrázek šlo kliknout a zobrazit jeho přiblížení pod seznamem.

Veškeré testy proběhly na zařízeních Pixel 4a a iPhone 12 Mini a byly opakovány celkem pětkrát.

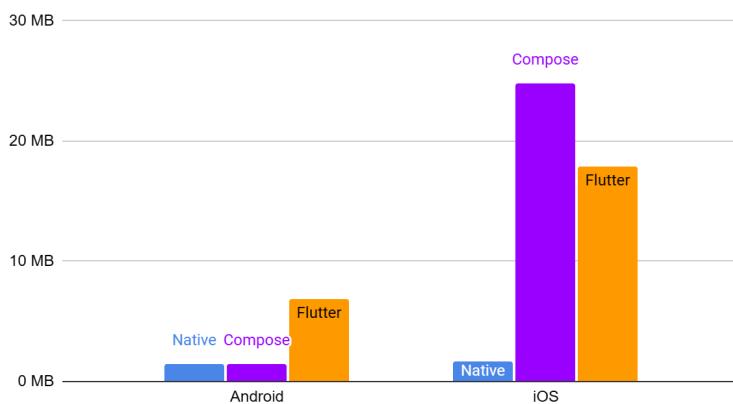
Velikost aplikace

Na obrázku 2.11 je vidět, že velikost aplikace založené na Compose Multiplatform, je identická s velikostí nativní aplikace pro Android. Je tomu tak z důvodu, že výsledná aplikace pro Android neobsahuje kód pro jiné platformy. [38] Kdežto u velikosti iOS aplikace je situace výrazně odlišná. Samotná aplikace pro iOS je v porovnání s její nativní aplikací o 23,1 MB větší. Tento rozdíl ve velikosti je způsoben především grafickou 2D knihovnou Skia, která je na platformě Android

dostupná, kdežto na platformě iOS se nenachází. Knihovna proto musí být spolu s aplikací dodána. [38]

Zajímavé je následné porovnání s velikostí aplikace založené na frameworku Flutter, jelikož ta, stejně jako Compose Multiplatform, využívá knihovnu Skia, ale i přesto je o něco velikostně menší než Compose Multiplatform aplikace na platformě iOS. Součástí aplikace Flutter je ještě vlastní engine, který je součástí aplikace a zvětšuje tak velikost aplikace asi o 3–4 MB pro Android a 10 MB pro iOS. [39]

Native vs. Compose Multiplatform vs. Flutter: App Size



■ **Obrázek 2.11** APK/IPA size in megabytes Zdroj: [38]

Rychlosť spuštění aplikace

Při porovnání rychlosti spuštění Compose Multiplatform aplikace na platformě Android není mezi rychlostmi téměř žádný rozdíl, stejně tak jako tomu bylo při porovnání velikosti aplikací v předchozí kapitole. O něco delší dobu spuštění měla aplikace napsaná pomocí frameworku Flutter, která se spouštěla v průměru o 221 ms déle než Compose Multiplatform aplikace. (viz tabulka 2.1) Toto zpomalení je s největší pravděpodobností způsobeno dobou spuštění Flutter Enginu, což by korespondovalo s oficiální Flutter dokumentací. [40] U posledního porovnání na platformě iOS se doba spuštění aplikací založených na Compose Multiplatform a frameworku Flutter o tolik nelišila od nativní aplikace (viz tabulka 2.2).

■ **Tabulka 2.1** Porovnání rychlosti spuštění ukázkové aplikace na platformě Android

	min.	median	max.
Native Android	408.7 ms	413.1 ms	423.1 ms
KMP Android	403.6 ms	425.3 ms	466.4 ms
Flutter Android	600.5 ms	634.2 ms	649.8 ms

■ **Tabulka 2.2** Porovnání rychlosti spuštění ukázkové aplikace na platformě iOS

	Duration (AppLaunch)
Native iOS	1.441 s
KMP iOS	1.618 s
Flutter iOS	1.608 s

2.3 Limitace Compose Multiplatform oproti nativnímu řešení

Jak již bylo zmíněno v kapitole *Kotlin Multiplatform 2.1.3.1*, tak jedním z největších problémů multiplatformního vývoje pomocí KMP je především nedostatečné množství knihoven.

Co se týče UI, tak zde je situace díky možnosti využití Jetpack Compose o něco lepší. Většina omezení, která jsou aktuálně spjata s UI, se týká především plynulosti anebo použití nativně vy padajících komponent na platformě iOS. Ta v její nativní podobě používá styl zvaný Cupertino, který frameworkem Compose Multiplatform není aktuálně oficiálně podporován. Co se týče plynulosti UI na platformě iOS, se momentálně nejvíce řeší problém s rychlostí vykreslování.

Většina výše zmíněných problémů se již řeší a je zmíněna v Kotlin Multiplatform vývojářském plánu pro rok 2024. [41] Konkrétně se jedná o veškeré zmíněné problémy kromě zařazení HarmonyOS mezi podporované platformy. V tomto plánu je mimo jiné uvedeno, že aktuální prioritou číslo jedna je změna frameworku Compose Multiplatform na platformě iOS do verze Beta. [41]

Kapitola 3

Návrh

V rámci této kapitoly bude nejprve vybrána aplikace, která bude použita k otestování použitelnosti frameworku Compose Multiplatform a následně bude tato aplikace navržena tak, aby splnila veškeré vytyčené cíle z úvodní sekce *Cíle práce 1.1*.

Po vybrání typu aplikace následuje sekce věnující se detekci případů užití. To je zejména vhodné specifikovat proto, aby bylo zřejmé jaké uživatelé budou danou aplikaci používat. Dalším aspektem je výběr platform, pro které bude následně aplikace implementována. Zároveň je vhodné skloubení případů užití s grafickou podobou UI tak, aby nejčastějšemu uživateli bylo možné splnit cíle, s co nejmenší dávkou úsilí. To je pro správný návrh UI klíčové.

Další fáze návrhu je věnována vytyčení funkčních požadavků, pomocí kterých je jasně specifikováno, jakými funkcionalitami by měla výsledná aplikace disponovat.

Dále je v rámci této kapitoly navržena architektura aplikace, která je pro následné navržení a správné fungování UI nezbytná. Sekce *Návrh architektury* se tedy věnuje rozebrání používaných typů architektury pro mobilní zařízení a následnému detailnímu rozebrání jednotlivých vrstev vybrané architektury.

V poslední části návrhu je přistoupeno k samotnému návrhu uživatelského rozhraní aplikace a to nejprve za použití takzvaných drátěných modelů. V dalším kroku je přistoupeno k návrhu design systému a nakonec k návrhu konečné podoby UI včetně grafických prvků.

3.1 Výběr aplikace

Základním požadavkem bylo vybrat takovou aplikaci, ve které by bylo možné použít velké množství různých komponent, jejichž funkčnost by následně mohla být otestována na různých platformách. Mezi další požadavky při výběru aplikace patřilo zaměření se na kritické problémy, které aktuálně multiplatformním vývojem prochází. Mezi takové problémy patří například využití funkcí, které jsou silně spjaty s hardwarem koncových zařízení. V aplikaci je to například použití jednotné navigace, fotoaparátu nebo služeb lokalizace.

Z těchto důvodů byla k implementaci vybrána aplikace sloužící pro občany měst či obcí, která kombinuje veškeré funkční požadavky, které plynou ze zadání.

Tato aplikace by sloužila občanům k získání informací o aktuálních novinkách, pořádaných akcích nebo například nabízela možnost vyhledat a zaplatit parkovné.

Na základě tohoto popisu je již zřejmé, že témto požadavkům nejlépe vyhoví mobilní aplikace, která bude použitelná na aktuálně nejrozšířenějších mobilních platformách, kterými jsou Android a iOS [42].

Pro potvrzení této hypotézy byly specifikovány následující případy užití.

3.2 Případy užití

Případy užití (zkráceně UC - z anglického „use case“) jsou scénáře popisující, jakým způsobem by případní uživatelé aplikace mohli využívat určité funkce nebo vlastnosti aplikace k dosažení svých cílů. [43] Tyto scénáře zachycují interakce mezi uživatelem a systémem a popisují, jak systém reaguje na určité vstupy od uživatele. Zároveň popisují jakou zpětnou vazbu uživatel na základě provedených vstupů od systému dostává.

Každý případ užití obvykle obsahuje následující prvky:

- **Název:** Stručný název, který popisuje, čeho chce uživatel dosáhnout.
- **Aktér:** Uživatel nebo systém, který spouští případ užití.
- **Popis:** Podrobný popis scénáře, který obsahuje kroky, které uživatel vykonává a odpovědi systému na tyto kroky.

Aktéři

Hlavními aktéry systému jsou především občané příslušného města, kteří budou aplikaci využívat za účelem splnění cílů zmíněných v rámci této kapitoly. Dalším aktérem tohoto systému je informační systém (IS) města, který aplikaci poskytuje potřebná data.

Pozn. Následující případy užití jsou pro jednoduchost popsány pouze z pohledu občanů města a z toho důvodu explicitně nespecifikují aktéra, jehož se daný případ užití týká.

UC1 Získání aktuálních novinek

Získání přístupu k aktuálním novinkám a důležitým oznámením ze svého města nebo obce.

1. Systém uživateli poskytne seznam aktualit načtený z webových stránek města a dalších informačních zdrojů.
2. Uživatel si vybere aktuality, která ho zaujmeme.
3. Systém uživateli vrátí detail vybrané aktuality obsahující její název, obsah, datum vytvoření, obrázek a případně odkazy na další zdroje.

UC2 Získání aktuálních událostí

Získání přehledu o aktuálně připravovaných akcích, událostech a kulturních aktivitách ve vybraném městě nebo obci.

1. Systém uživateli poskytne aktuální seznam událostí z webových stránek města, divadel, kin a dalších informačních zdrojů.
2. Uživatel si vybere událost, která ho zaujme.
3. Systém uživateli poskytne detail vybrané události obsahující její název, datum konání, popis, místo konání obrázek a případně odkazy na další zdroje.

UC3 Získání událostí za základě času

Filtrování zobrazených událostí na základě data konání jednotlivých událostí.

1. Systém uživateli poskytne seznam událostí z webových stránek města, divadel, kin a dalších informačních zdrojů, které jsou rozděleny na základě kategorií jako například kino, divadlo, hudba atd..
2. Uživatel si vybere konkrétní den nebo časový rozsah, pro který chce zobrazit konané události.
3. Systém uživateli poskytne seznam událostí konaných ve vybraný den (případně vybraný časový rozsah) a tyto události zároveň rozdělí podle příslušných kategorií.

UC4 Získání kontaktů na místní úřady

Získání kontaktů na místní úřady nebo správní orgány.

1. Systém uživateli poskytne seznam městských institucí.
2. Uživatel vybere konkrétní instituci, pro kterou chce zobrazit detailní informace.
3. Systém vrátí podrobné informace o vybrané instituci.

UC5 Nahlášení závady ve městě

Možnost nahlášení závady na území města.

1. Systém uživateli poskytne formulář k vyplnění městských institucí.
2. Uživatel vyplní název závady, nahraje fotografiu závady, vyplní místo, kde se závada nachází, vybere kategorii závad do, které závada spadá a záznam o závadě odešle.
3. Systém závadu odešle příslušnému orgánu a uživateli vrátí informaci o jejím zaslání.

UC6 Propojení se sociálními kanály města

Možnost konzumovat aktuální novinky prostřednictvím videí z městského YouTube kanálu nebo prostřednictvím sociální sítě Facebook.

1. Systém uživateli poskytne městem používané sociální sítě.
2. Uživatel zvolí požadovanou sociální síť.
3. Systém uživatele přesměruje na požadovanou sociální síť.

UC7 Vyhledání parkovacích zón

Možnost jednoduše vyhledat veškeré parkovací zóny ve městě včetně informace o provozní době a ceníku parkovací zóny.

1. Systém uživateli poskytne mapu parkovacích zón.
2. Uživatel vybere parkovací zónu, pro kterou požaduje zobrazit detailní informace.
3. Systém uživateli vrátí detailní informace týkající se vybrané parkovací zóny.

UC8 Platba parkovného

Možnost rychle zaplatit poplatek za parkování.

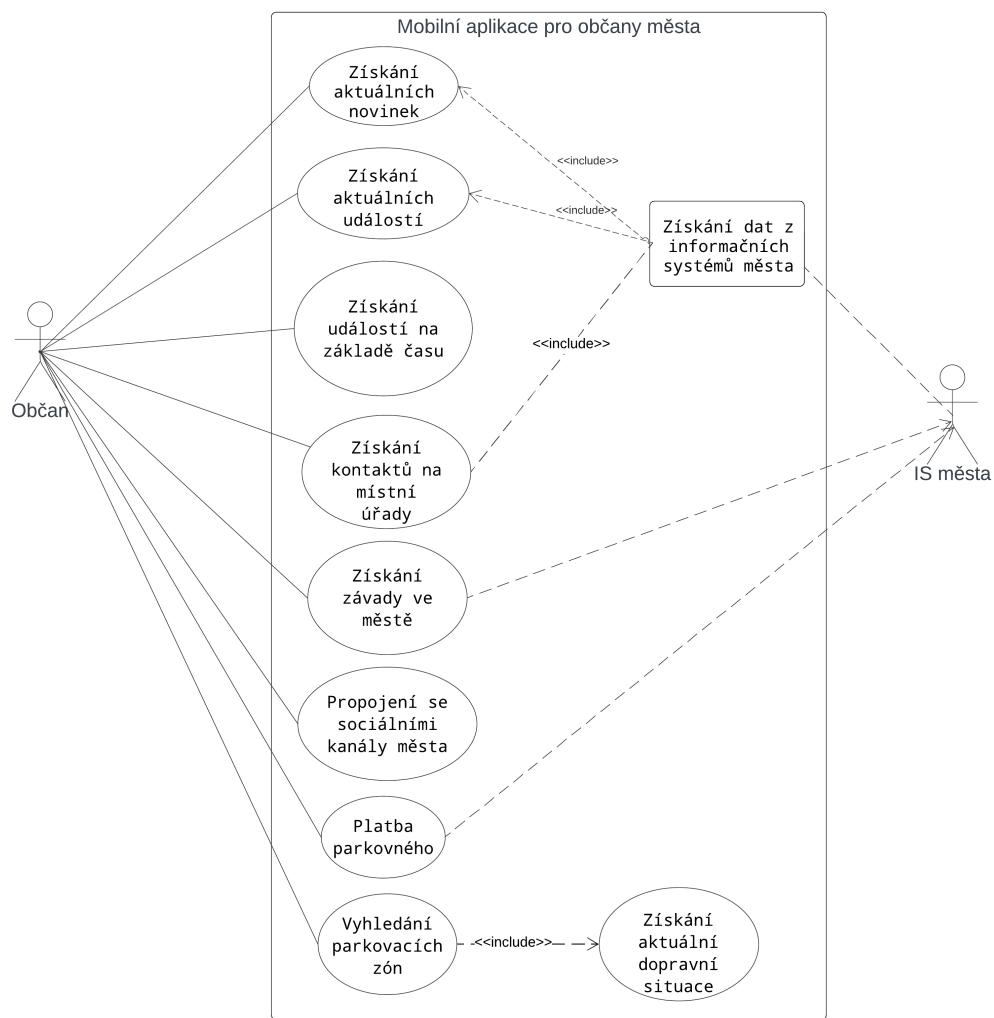
1. Systém uživateli zobrazí mapu parkovacích zón v příslušném městě.
2. Uživatel vybere parkovací zónu, pro kterou si přeje zaplatit poplatek za parkování.
3. Systém uživateli vrátí detailní informace týkající se vybrané parkovací zóny.
4. Uživatel v detailu vybrané parkovací zóny vybere akci „zaplatit poplatek za parkování“.
5. Systém uživatele přesměruje na platební systém používaný daným městem pro platbu parkovného s předvyplněnou informací o vybrané parkovací zóně.

UC9 Získání aktuální dopravní situace

Možnost rychle získat přehled o aktuální dopravní situaci ve městě.

1. Systém uživateli zobrazí mapu vytíženosti jednotlivých silnic v rámci města.
2. Uživatel tuto informaci zpracuje.

Pro dotvoření představy o tom, jak jsou spolu jednotlivé případy užití navzájem propojeny, slouží obrázek 3.1, který zároveň obsahuje i typy uživatelů pracujících s navrženou aplikací.



■ Obrázek 3.1 Diagram případů užití

Případy užití jsou klíčovým nástrojem pro pochopení potřeb uživatelů a návrhu funkcionalit aplikace tak, aby co nejlépe odpovídaly těmto potřebám. Jsou také důležitým zdrojem pro testování a validaci funkčnosti aplikace, jelikož lze díky nim jednoduše ověřit, zda systém správně reaguje na očekávané uživatelské interakce. Podrobněji se tomuto tématu věnuje poslední kapitola *Testování 5*, v rámci které byly případy užití použity jako základ k tvorbě UI testů.

3.3 Specifikace požadavků

Na základě případů užití sepsaných v předchozí sekci, je nyní možné sepsat konkrétní požadavky na vybranou aplikaci. Pro konkretizaci těchto požadavků byla použita specifikace na základě funkčních a nefunkčních požadavků, která je součástí ISO/IEC/IEEE 29148:2018, jenž popisuje proces vývoje softwarových produktů.

3.3.1 Funkční požadavky

Funkční požadavky (zkráceně FR - z anglického „functional requirement“) definují konkrétní funkce a chování systému z pohledu uživatele. Obecně se dá říci, že specifikují, co má systém dělat, jaké úkoly má provádět a jaké operace musí uživatelům umožňovat.

FR1 Zobrazení aktualit a událostí

Systém bude přehledně zobrazovat seznamy aktualit a událostí v daném městě včetně jejich detailních informací jako jsou názvy jednotlivých událostí a novinek, časy vydání případně datová rozmezí konání událostí a příslušné popisy těchto událostí a novinek.

- *Priorita: 5*
- *Složitost: 4*

FR2 Filtrvání událostí

Systém bude filtrovat konané události v daném městě podle data jednotlivých událostí a tyto události zároveň přehledně rozřadí do patřičných kategorií jako například *Kino*, *Divadlo*, *Celodenní akce* *Akce pro děti* a další.

- *Priorita: 4*
- *Složitost: 4*

FR3 Propojení s komunikačními kanály města

Systém poskytne uživateli propojení s aktuálně používanými sociálními sítěmi města buďto v podobě odkázání na webové stránky dané sociální sítě nebo rovnou pomocí otevření profilu města v aplikaci vybrané sociální sítě.

- *Priorita: 3*
- *Složitost: 1*

FR4 Zpracovávat nahlášené závady

Systém uživateli poskytne přehledný formulář pro zasílaní závady do informačního systému města a tuto závadu zároveň patřičně zpracuje.

- *Priorita: 1*
- *Složitost: 4*

FR5 Zobrazit kontakty na místní úřady

Pomocí systému bude možné zobrazit aktuální kontakty na místní úřady, jejich pracovní dobu a konkrétní otevírací hodiny pro každý z místních úřadů.

- *Priorita: 2*
- *Složitost: 2*

FR6 Správa parkovací zón

Pomocí systému bude možné zobrazit detailní informace o parkovacích zónách jako je provozní doba nebo výše parkovacích poplatků a současně jednoduše zaplatit parkovací poplatky pro vybranou zónu.

- *Priorita: 3*
- *Složitost: 4*

FR7 Zobrazení dopravy na mapě města

Pomocí systému bude možné jednoduše a rychle zobrazit aktuální vytíženosť silnic ve městě pomocí intuitivní překryvné vrstvy mapových podkladů.

- *Priorita: 2*
- *Složitost: 2*

3.3.2 Nefunkční požadavky

Nefunkční požadavky (zkráceně FR - z anglického „Non-functional requirement“) se oproti funkčním požadavkům nezaměřují na funkčnost samotného produktu („co má systém dělat“), ale spíše na jeho vlastnosti („jaký má být“) z pohledu výkonu, spolehlivosti, bezpečnosti, uživatelské přívětivosti a dalších aspektů, které nepřímo ovlivňují uživatelský zážitek a kvalitu systému. Tyto požadavky jsou z tohoto důvodu často spojeny s technickými a provozními charakteristikami systému.

NFR Multiplatformnost

Systém by měl být dostupný alespoň na mobilních aplikacích.

NFR1 Rozšiřitelnost

Systém by měl být rozšiřitelný na ostatní podporované platformy.

NFR2 Offline dostupnost

Systém by měl správně fungovat i v případě, že není připojen k internetu.

NFR3 Přístupnost

Systém by měl být přizpůsoben širokému spektru lidí - například zrakově postiženým.

Pokrytí případů užití

Pro ověření a názornou ukázkou, že takto sepsané funkční požadavky pokrývají všechny případy užití, byla vytvořena následující tabulka pokrytí. 3.1

	F1	F2	F3	F4	F5	F6	F7
UC1	*						
UC2	*						
UC3		*					
UC4				*			
UC5			*				
UC6		*					
UC7					*		
UC8					*		
UC9						*	

■ **Tabulka 3.1** Pokrytí případů užití funkčními požadavky aplikace

3.4 Architektura aplikace

Navržená architektura se z velké části drží architektonických principů shrnutých v Android dokumentaci.

Důležité principy

Separation of concerns

Separation of concerns je princip návrhu softwaru, který popisuje důležitost oddělení různých funkcionalit do samostatných, dobře definovaných a izolovaných částí. [44] Hlavním cílem tohoto principu je zlepšit modularitu, udržitelnost, znovupoužitelnost a spravovatelnost kódu.

Princip oddělení zájmů rozděluje systém na jednotlivé komponenty nebo moduly, přičemž každý z nich se zabývá jednou konkrétní oblastí funkcionality. Každá část systému má jasné definované rozhraní, které umožňuje interakci s ostatními částmi. [44] Tímto způsobem lze snadněji spravovat a udržovat kód, protože změny v jedné části systému nemusí mít nežádoucí dopady na ostatní části.

Příklady oddělení zájmů zahrnují oddělení prezentační vrstvy od logiky aplikace (Model-View-Controller), oddělení datového přístupu od podnikové logiky (Repository pattern) a oddělení různých vrstev aplikace (např. vrstva uživatelského rozhraní, aplikační logika, datová vrstva).

Unidirectional Data Flow

Unidirectional Data Flow (jednosměrný tok dat) je architektonický vzor, který popisuje způsob, jakým data cestují skrz aplikaci. [45] V tomto přístupu data proudí v jednom směru, což znamená, že existuje jasný a jednoznačný tok dat od zdroje až po jejich konečný cíl. Takovýto přístup například umožňuje efektivní aktualizaci uživatelského rozhraní v reakci na změny dat a přispívá k jednoduchosti, předvídatelnosti a udržitelnosti softwarových aplikací.

Single source of truth

Princip Single Source of Truth (SSOT) je koncept, který zdůrazňuje, že by v aplikaci měl existovat jeden zdroj dat, který obsahuje aktuální a spolehlivé informace. [45] Tento zdroj dat je považován za „pravdivý“ a slouží jako jediný zdroj, ze kterého mohou ostatní části aplikace čerpat informace. Tím se zajišťuje konzistence dat napříč aplikací a minimalizuje se riziko konfliktů nebo chyb v datech.

Díky použití principu SSOT jsou data v aplikaci konzistentní a snadněji spravovatelná, protože všechny komponenty a moduly aplikace pracují se stejnými daty. To usnadňuje údržbu a vývoj aplikace, protože úpravy a aktualizace dat lze provádět centrálně. Z hlediska kódu přispívá tento princip k jednoznačnosti a přehlednosti, protože vývojáři vědí, kde hledat relevantní data pro různé části aplikace.

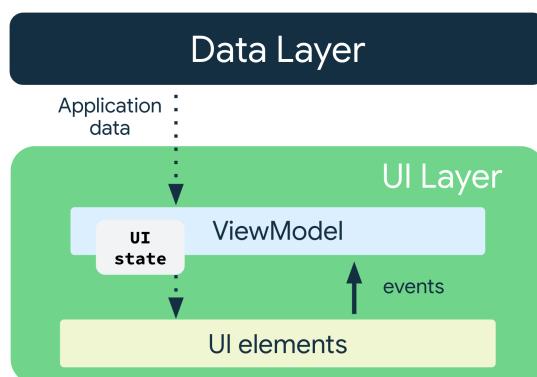
Cílem tohoto konceptu je zlepšit konzistenci, přehlednost a údržbu aplikace a poskytnout spolehlivý zdroj dat pro všechny části aplikace.

Vrstvy architektury

Na základě těchto principů vznikla níže popsaná architektura, která je aktuálně doporučovaná pro vývoj mobilních Android aplikací a rozděluje se do následujících třech základních vrstev.

UI vrstva

UI vrstva se stará o zobrazení dat na obrazovce, která odpovídají aktuálnímu stavu aplikace. [45] Zároveň kontroluje logiku, která se při změně těchto dat stará o opětovné překreslení UI tak, aby opět odpovídalo aktuálnímu stavu aplikace. Tohoto chování UI vrstva dociluje pomocí tří částí, které jsou zobrazeny na obrázku 3.2 a následně detailněji popsány pod ním.



■ Obrázek 3.2 Architektura UI vrstvy Zdroj: [46]

ViewModel

Hlavní účel ViewModelu je uchovávat a spravovat data potřebná pro zobrazení na obrazovce a to i při změnách konfigurace zařízení (například při otáčení obrazovky) [47]. ViewModel také poskytuje metody a události pro interakci s daty jako je načítání dat z úložiště, jejich aktualizace a předávání událostí od uživatele zpět do aplikace. Na základě těchto událostí například aktualizuje stav uživatelského rozhraní.

UI state

UI state (stav uživatelského rozhraní) je část UI vrstvy, který popisuje aktuální stav a chování uživatelského rozhraní v daném okamžiku [45]. Tento stav může zahrnovat různé informace jako jsou aktuální hodnoty vstupních polí, stav vybraných prvků, aktuální zobrazené obrazovky nebo panely, stav animací a další.

UI state je důležitý pro udržování konzistence a interaktivnosti uživatelského rozhraní. Změny v UI stavu mohou být způsobeny uživatelskými interakcemi jako například kliknutím na tlačítko, zadáním textu do pole, rolováním seznamu nebo mohou být vyvolány událostmi v aplikaci v případech jako je získání dat ze serveru nebo jakoukoliv změnou interního stavu aplikace.

V rámci frameworků, které k tvorbě UI používají deklarativní způsob zápisu se často k definice stavu a jeho vlivu na UI používá funkce na obrázku 3.3. Ta zjednodušeně popisuje výsledné uživatelské rozhraní, které je tvořeno na základě jeho stavu.



■ **Obrázek 3.3** Reprezentace vztahu UI a stavu aplikace Zdroj: [48]

Správa UI stavu je důležitou součástí vývoje aplikací s uživatelským rozhraním a může být implementována pomocí různých technik a nástrojů. Důležité je udržovat UI stav v souladu s interním stavem aplikace a zajistit jeho konzistence a správnou aktualizaci v reakci na uživatelské interakce a události v aplikaci.

UI Elements

UI elements neboli prvky uživatelského rozhraní jsou komponenty tvořící vizuální část aplikace a umožňují uživatelům interagovat s aplikací. [45] Tyto prvky zahrnují různé vizuální komponenty jako jsou tlačítka, textová pole, seznamy, obrázky, ikony, přepínače, posuvníky atd., díky nimž se zobrazují data reprezentující aktuální stav aplikace nebo je pomocí nich tento stav měněn.

UI prvky jsou navrženy tak, aby poskytovaly uživatelům intuitivní způsob jak s aplikací komunikovat a manipulovat. Každý prvek má své vlastnosti jako je velikost, barva, text, stav atd., které lze nastavit a upravovat pomocí kódu. Tyto prvky jsou pak umístěny na obrazovce podle určeného rozvržení (layout), které určuje jejich pozici a vzájemné uspořádání.

UI prvky jsou základními stavebními bloky uživatelského rozhraní a hrají klíčovou roli při vytváření uživatelsky přívětivé a atraktivní aplikace. Jejich vhodné použití a umístění má vliv na celkový uživatelský zážitek a efektivnost aplikace.

UI Events

Nejčastějším typem UI událostí jsou takzvané uživatelské události, které jsou generovány interakcí uživatele s uživatelským rozhraním aplikace. [49] Patří k nim například kliknutí myši, dotyk na obrazovce, stisknutí klávesy nebo jakékoli jiné akce prováděné uživatelem, které vyvolávají reakci v UI aplikaci.

O zpracování uživatelských událostí se většinou stará příslušný ViewModel, který nabízí veškeré funkce, které je pomocí uživatelského rozhraní možné volat. Nicméně, některé typy uživatelských událostí může UI zpracovat přímo. Takovými událostmi jsou například navigování na jinou obrazovku nebo zobrazení informativní zprávy pomocí takzvaných *Snackbars*.

Doménová vrstva

Doménová vrstva typicky obsahuje způsoby užití. Nicméně, pro menší aplikace jako je tato, může být vynechána. [50] Často se tímto způsobem „odlehčuje“ ViewModeley, které by jinak obsahovaly příliš mnoho kódu, což by vedlo k nepřehlednosti.

V tomto projektu doménová vrstva zahrnuje deklarace výčtových typů, objektů pro datovou a modelovou vrstvu a mapovací funkce pro převod objektů datové vrstvy na objekty používané v prezentační vrstvě a naopak.

Doménová vrstva by měla být nezávislá na technických aspektech aplikace a měla by se zaměřovat pouze na reprezentaci a správu doménových konceptů a procesů. To zjednodušuje testování, údržbu, rozšiřitelnost aplikace a umožňuje snadnou změnu technologií nebo platformy bez vlivu na doménovou logiku.

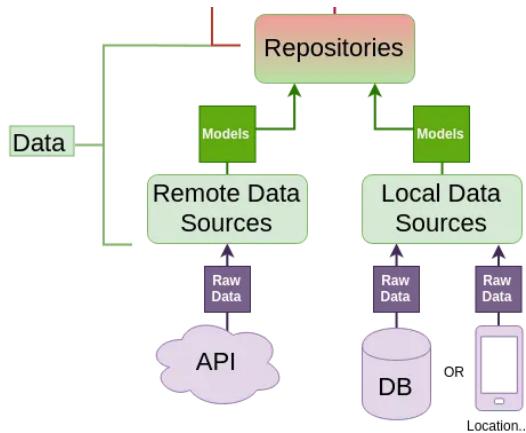
Datová vrstva

Datová vrstva se stará o persistencí dat a komunikaci s datovými úložišti jako jsou databáze, soubory nebo vzdálené API. Jejím hlavním úkolem je poskytovat rozhraní mezi doménovou logikou aplikace a datovými zdroji, aby bylo možné ukládat, načítat, aktualizovat a mazat data podle potřeby. [51]

K tomu aby aplikace fungovala, jak bylo popsáno v úvodu této kapitoly, je zapotřebí, aby datová vrstva této aplikace obsahovala mechanismy pro získání dat z informačních systémů města a vhodně je kombinovala s daty uloženými v lokální databázi.

Jelikož se data v informačních systémech města nachází ve formátu XML, byla proto navržena služba získávající tato data ze vzdáleného datového zdroje (IS města) způsobem zobrazeným v levé části obrázku 3.4. Získaná data jsou následně propojena s aplikací pomocí takzvaných *Data access object*, jejichž princip je podrobněji popsán dále v této sekci.

Pro persistenci těchto dat byla vybrána SQL databáze založená na enginu SQLite, která se stará o poskytnutí dříve načtených dat v případech, kdy aplikace nemá přístup k internetu.



Obrázek 3.4 Architektura datové vrstvy Zdroj: [46]

Následující výčet prvků má za úkol představit důležité komponenty datové vrstvy používané v rámci navržené aplikace.

Data access object

Data access objects (zkráceně *DAOs*) jsou zodpovědné za přímý přístup k datovým úložištěm a provádění operací jako čtení, zápis, aktualizace a mazání dat. [52] Tyto objekty poskytují abstrakci nad konkrétními technologiemi datových úložišť a umožňují snadnou změnu úložišť bez vlivu na ostatní části aplikace.

Data transfer object

Data transfer object (zkráceně *DTO*) jsou objekty používané k přenosu dat mezi vrstvami aplikace. Tyto objekty slouží k přenosu dat mezi datovou vrstvou a doménovou vrstvou a často mapují datové entity na jednodušší objekty nebo struktury pro snadnější manipulaci. [53]

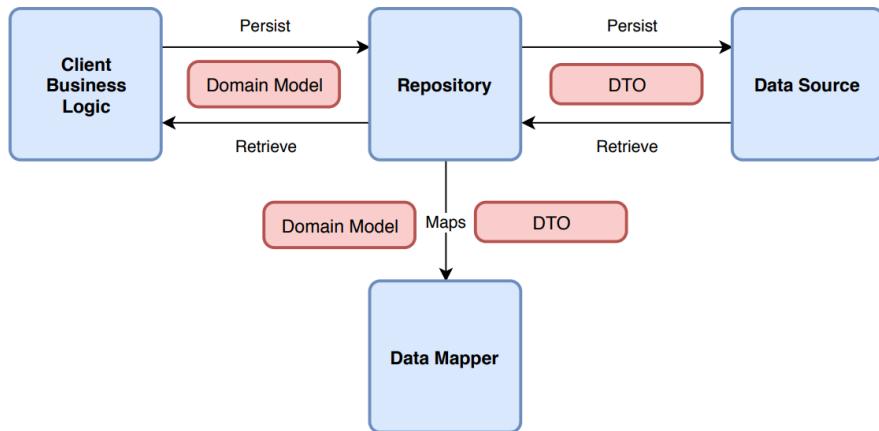
Datové entity

Datové entity představují strukturu a schéma dat uložených v databázi nebo jiném datovém úložišti. Tyto entity obvykle přímo odražejí strukturu tabulek v relační databázi nebo dokumentů v NoSQL databázi a poskytují model, se kterým mohou pracovat ostatní vrstvy aplikace.

Datová úložiště

Datová úložiště jsou fyzická místa, kde jsou data uložena. Mohou to být relační databáze, NoSQL databáze, soubory na disku nebo externí API. Datová vrstva zajišťuje, aby bylo možné efektivně pracovat s těmito úložišti, a zároveň byla data bezpečně uložena a získána.

Tyto části jsou pak spolu v rámci datové vrstvy propojeny následujícím způsobem:



■ **Obrázek 3.5** Provázanost jednotlivých částí datové vrstvy Zdroj: [54]

3.5 Uživatelské rozhraní

3.5.1 Drátěné modely

Navržené drátěné modely byly vytvořeny na základě funkčních požadavků a to tak, aby nejdůležitější a nejčastěji používané funkce byly umístěny na domovskou obrazovku. Zbylé méně podstatné nebo prostorově náročnější byly umístěny na sekundární obrazovky.

Obrazovka Domů

Obrazovka *Domů* (viz obrázek 3.6) byla navržena tak, aby jakožto hlavní stránka celé aplikace obsahovala pro uživatele co nejpřínosnější informace a zároveň, aby sloužila jako určitý rozcestník na další obrazovky aplikace. Z tohoto důvodu byla do horní části obrazovky navržena tlačítka, která uživateli umožňují rychlý proklik na YouTube kanál města, nahlášení závady, zobrazení úředních hodin nebo stránku služeb města.

Do střední části obrazovky byl navržen posuvný řádek obsahující nově přidané události a to především díky možnosti zobrazit veškeré potřebné události kompaktně na malém prostoru, ale přesto uživateli nabídnout možnost danou událost otevřít a dohledat zbylé informace. Akce poutají největší pozornost především svou vizuální reprezentací a převážná část informace je uživateli předána právě pomocí titulního obrázku, názvu a času konání.

Oproti tomu pro seznam novinek ve spodní části obrazovky bylo vybráno usporádání novinek do posuvného sloupce. Ten lépe umožňuje využít celou šíři obrazovky k zobrazení delších titulků a textů, které jsou pro uživatele v případě aktualit nejdůležitější.

Obrazovka Události

V rámci obrazovky *Události* (viz obrázek 3.7) je použit stejný systém pro zobrazení událostí - pomocí posuvného řádku, stejný způsob je použit na obrazovce *Domů*.

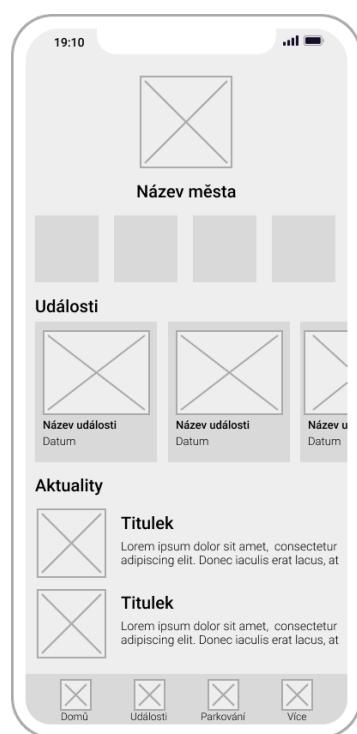
Oproti domovské stránce jsou zde však zobrazeny veškeré události, které se ve městě budou konat. Pro lepší přehled jsou události rozděleny do několika kategorií jako například *Kino*, *Divadlo*, *Celodenní akce*, *Akce pro děti* nebo *Přednášky*.

V horní části aplikace je pak uživateli dána možnost vybrat den, pro který chce zobrazen plánované události a případně si zvolit požadované rozmezí.

To je možné navolit pomocí kalendáře, který se zobrazí po kliknutí na ikonu kalendáře v pravé horní části obrazovky.

Aplikace byla navržena tak, aby uživatelé mohli jednoduše filtrovat zobrazené události primárně podle času.

Filtrování podle kategorie bylo nejprve navrženo, ale následně od něho bylo upuštěno kvůli malému počtu kategorií.



Obrázek 3.6 Drátěný model obrazovky *Domů*



Obrázek 3.7 Drátěný model obrazovky *Události*

Ve finále byl zvolen přístup, který vybrané události přiřadí do patřičné kategorie a zobrazí jen ty kategorie, které obsahují alespoň jednu událost.

Filtrování podle kategorie bylo nejprve navrženo, ale následně od něho bylo upuštěno kvůli malému počtu kategorií.

Obrazovka *Parkování*

Obrazovka parkování (viz obrázek 3.8) slouží uživateli dle dříve specifikovaných případů užití, především k vyhledání parkovací zóny, získání informací o ceně parkování a případně k platbě parkovacích poplatků. Z toho důvodu byla hlavní část obrazovky věnována mapovému podkladu, který uživateli umožňuje snadné nalezení parkovací zóny v mapě města.

Pro ještě snadnější hledání parkovacích zón dává aplikace uživateli možnost vyhledání požadované parkovací zóny pomocí vyhledávacího pole, které umožňuje vyhledat příslušnou parkovací zónu buďto podle čísla nebo názvu zóny.

Dále zde byla implementována výsuvná karta ze spodní části obrazovky, která se zobrazí pouze v případě, že uživatel vybere jednu z nabízených parkovacích zón. Díky této implementaci se uživateli nezměnuje prostor pro rychlé vyhledání parkovací zóny, které je hlavním cílem k použití této obrazovky.

Tato výsuvná karta uživateli zobrazí detailní informace k vybrané parkovací zóně a umožní zaplatit parkovací poplatek pro vybranou parkovací zónu.

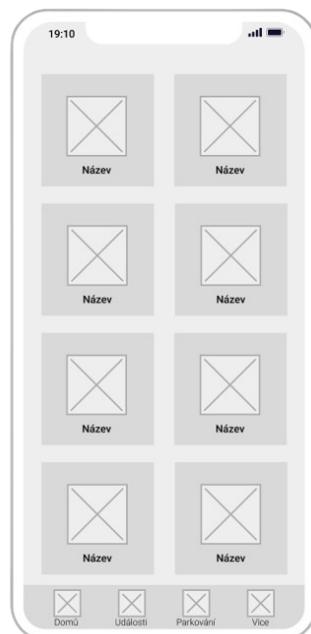
Poslední funkcí, kterou tato obrazovka nabízí je zobrazení aktuální dopravní situace ve městě, která je na mapě zobrazena spolu s parkovacími zónami.

Obrazovka *Více*

Poslední navrženou obrazovkou je obrazovka *Více*, která obsahuje přehled veškerých méně podstatných částí aplikace.



Obrázek 3.8 Drátěný model obrazovky *Parkování*



Obrázek 3.9 Drátěný model obrazovky *Více*

3.5.2 Design systém

Před tím něž bylo přistoupeno ke grafickému návrhu aplikace, tak byl vytvořen obecný design systém, který poslouží k vytvoření konzistentního uživatelského rozhraní. Zároveň se díky němu urychlí proces navrhovaní grafické podoby UI a při následné fázi implementace zajistí intuitivní ovládání aplikace na všech implementovaných platformách.

Design systém je obecně vzato sada předem definovaných pravidel, komponent, standardů a návrhových principů, které slouží k vytváření a udržování konzistentního a jednotného vizuálního stylu v rámci celé aplikace. Takový systém zahrnuje různé prvky jako jsou barvy, typografie, ikony, způsoby rozložení UI nebo komponenty uživatelského rozhraní a umožňuje vývojářům a designérům efektivně spolupracovat a vytvářet aplikace s jednotným vzhledem. Cílem návrhu design systému je zajistit, aby všechny části aplikace měly jednotný vzhled a chování, což zlepšuje především uživatelskou zkušenosť s implementovanou aplikací.

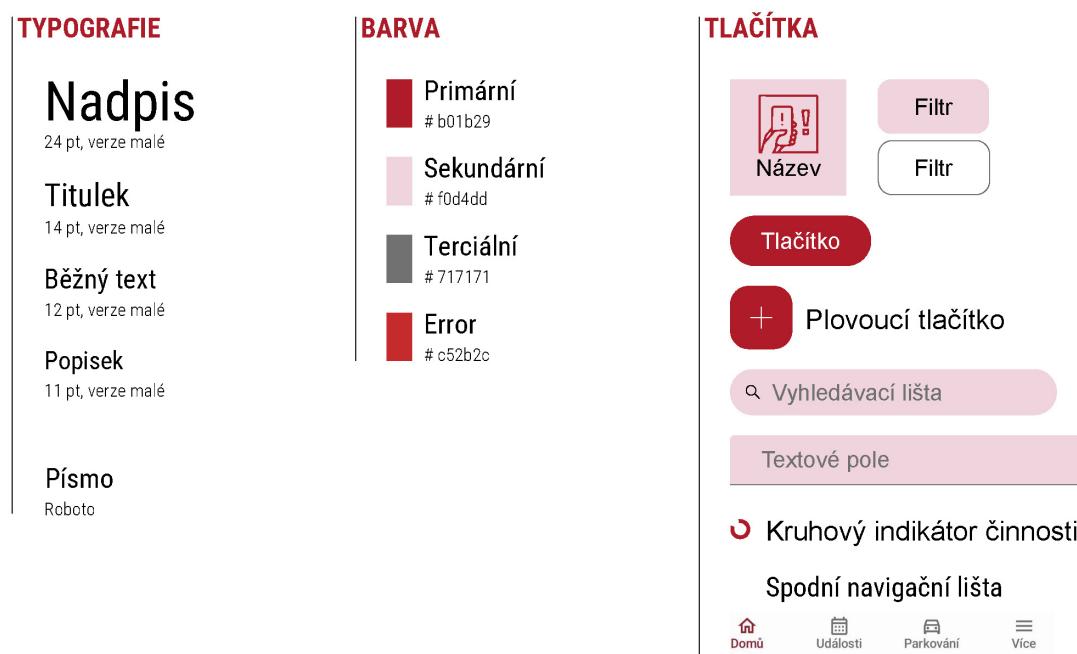
V rámci vybraného města Příbrami není takový systém nikde explicitně zadefinovaný, a proto byl odvozen z designu webových stránek města.

Zároveň byl zkombinovaný s aktuálně nejnovějším design systémem od společnosti Google zvaným *Material Design 3*.

S ohledem na to, že aplikace je navrhována pro použití na mobilních zařízeních, tak je tomu uzpůsoben i návrh samotného design systému, který specifikuje především barvy, styly textů a prvky použitelné na mobilních platformách. Zbylé části mohou být případně v budoucnu rozšířeny o další prvky používané na jiných platformách.

Součástí návrhu je tak pár základních UI prvků, které budou použity v rámci implementované aplikace a mohou být použity i případně dalšími aplikacemi města.

Výsledek základní ukázky takto navrženého systému je k vidění na obrázku 3.10



■ Obrázek 3.10 Ukázka navrženého design systému

3.5.3 Mockup modely

Po zadefinování použitelných komponent v předchozí sekci *Design systém 3.5.2* je již samotný návrh grafické podoby UI zjednodušen na nezbytné minimum. Vytvořené mockup modely jsou totiž de facto výsledkem sloučení drátěných modelů s navrženým designem systémem.

Při návrhu grafické podoby UI byla proto pozornost primárně soustředěna na význam jednotlivých UI komponent a na základě jejich významu byly přiřazovány například odpovídající barvy. Navržené UI se tímto přístupem snaží respektovat zásady správného UI návrhu dle Android dokumentace a to především z pohledu přístupnosti, rozložení obsahu, druhu použitých UI komponent nebo vhodnosti použití barev z hlediska sémantiky barev.

Přístupnoust

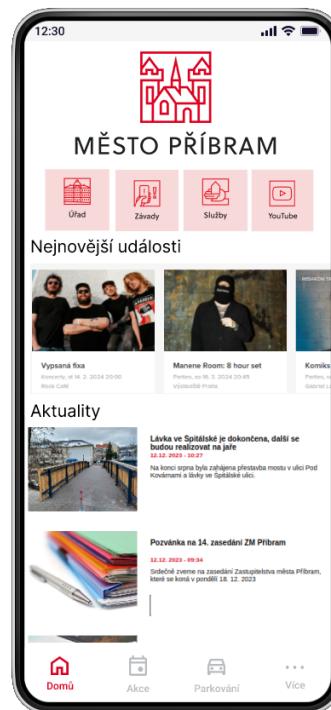
Z hlediska přístupnosti je UI aplikace navrženo tak, aby jednotlivé prvky UI měly vhodně zvolenou barvu z pohledu správného kontrastu vůči podkladu anebo z pohledu dostatečné čitelnosti.

Dále byla věnována pozornost správnému popisu jednotlivých UI komponent a správnému použití dle druhu komponent tak, aby bylo možné aplikaci používat i s takzvanou TalkBack čtečkou, která umožňuje nevidomým nebo jinak zrakově postiženým lidem používat aplikaci například na základě hlasové odesady.

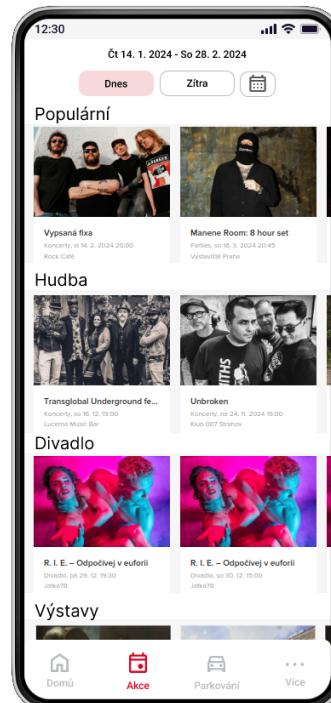
Barvy

I přesto, že základní barvy byly zvoleny již při navrhování design systému, během fáze návrhu mockup modelu byl kladen důraz na jejich vhodné použití a to především z pohledu sémantiky barev nebo důležitosti barvených komponent.

Základní popisy barev použité při návrhu design systému byly odvozeny z *Material 3* design systému, v rámci kterého mají jasně vymezené role. [55] Primární barva je proto v případě implementovaného UI například použita pro potvrzovací akci „zaplacení parkovacích poplatků“. Sekundární barva byla naopak implementována například na filtrační tlačítka na obrazovce *Události*, ježlikož hlavním předmětem pozornosti na dané obrazovce jsou zobrazované akce a nikoliv filtrační možnosti.



■ Obrázek 3.11 Obrazovka Domů



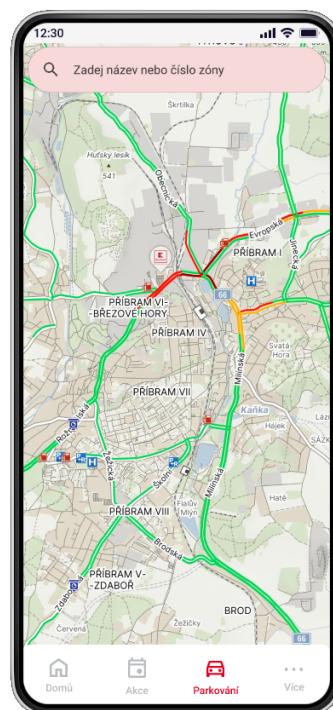
■ Obrázek 3.12 Obrazovka Události

Rozložení

Co se rozložení UI komponent na obrazovky týká, tak zde bylo respektováno několik pravidel, jako například liniového zavrhání UI komponent tak, aby nedocházelo k narušení čitelnosti nebo dodržení minimální vzdálenosti od krajů obrazovky.

Komponenty jako primární navigace nebo tlačítka, která slouží pro vykonávání důležitých akcí byla implementována na taková místa, aby byla v takzvané dosahové vzdálenosti.

Pro obrázky byly voleny běžné poměry jako například 4:3, aby nedocházelo k jejich zbytečnému ořezávání.



Obrázek 3.13 Obrazovka *Parkování*



Obrázek 3.14 Obrazovka *Vice*

Kapitola 4

Implementace

4.1 Založení Compose Multiplatform projektu

K založení projektu byl použit nástroj *Kotlin Multiplatform Wizard* od společnosti JetBrains, který slouží k rychlému vytvoření projektu založeném na technologiích KMP. Tento nástroj se také stará o založení sestavovacího skriptu na technologii Gradle, který je následně možné použít k sestavení multiplatformních aplikací založených na KMP a frameworku Compose Multiplatform.

4.1.1 Gradle

Gradle je open-source nástroj pro automatizaci sestavení a správu závislostí při vývoji softwaru. Je podobný technologiím Ant nebo Maven, které byly vyvinuty pro zjednodušení procesu sestavení a nasazení softwaru.

Gradle používá deklarativní doménově specifické jazyky (DSL), díky kterým je možné definovat sestavení a konfiguraci projektu pomocí srozumitelné a flexibilní syntaxe. Od verze 3.0 z roku 2016 je možné psát Gradle skripty v jazyce Kotlin DSL a od roku 2023 je Kotlin DSL primárním jazykem pro jejich zápis. To umožňuje vývojářům snadno konfigurovat sestavení a spravovat závislosti na knihovnách a modulech v jazyku se stejnou syntaxí jako je ta, používaná pro ostatní části projektu.

K tomu, aby bylo možné Compose Multiplatform v projektu použít, je zapotřebí do projektového souboru `build.gradle.kts` přidat Compose Multiplatform plugin (viz výpis kódu 4.1).

■ Výpis kódu 4.1 Integrace Compose Multiplatform zásuvného modulu do sestavovacího scriptu

```
plugins {
    id("org.jetbrains.compose") version "1.6.0"
}
```

Dále je zapotřebí vybrat platformy, pro které má být aplikace sestavována a podle nich vytvořit příslušné balíčky (viz sekce *Kotlin Multiplatform 2.1.3.1*). V případě implementované aplikace se jedná o platformy *Android*, *iOS* a *desktop*, které bylo nutné specifikovat v projektovém souboru `build.gradle.kts` (viz řádky 2 - 21 na výpisu kódu 4.2). Podle těchto platform lze následně v příslušném `build.gradle.kts` vytvořit takzvané `sourceSets`, díky kterým

je možné pro každou platformu implementovat platformě specifické závislosti. Tyto závislosti se následně vkládají do příslušných sourceSets podle toho, zdali se jedná o multiplatformní (`commonMain`) nebo nativní závislost (`androidMain`, `iosMain`, atd.).

Pro ukázku, jak taková multiplatformní závislost může vypadat, slouží řádky 23 - 28 na ukázce kódu 4.2, pomocí kterých je do projektu přidána knihovna umožňující implementaci multiplatformní navigace.

■ Výpis kódu 4.2 Lib integration

```
1  kotlin {
2      androidTarget {
3          compilations.all {
4              kotlinOptions {
5                  jvmTarget = "11"
6              }
7          }
8      }
9
10     jvm("desktop")
11
12     listOf(
13         iosX64(),
14         iosArm64(),
15         iosSimulatorArm64()
16     ).forEach { iosTarget ->
17         iosTarget.binaries.framework {
18             baseName = "ComposeApp"
19             isStatic = true
20         }
21     }
22
23     sourceSets {
24         commonMain.dependencies {
25             implementation(libs.voyager.navigator)
26             ...
27         }
28     }
29 }
```

Dále je potřeba do spouštěcích souborů každé platformy integrovat multiplatformní část aplikace, která může být do nativního kódu přidána například pomocí *Composable* funkce, která je předmětem společné části aplikace.

Ve výpisu kódu 4.3 je to funkce `App()`, která je volána z nativního Android kódu a stará se o deklaraci veškerého multiplatformního UI.

■ Výpis kódu 4.3 Lib integration

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            App()  
        }  
    }  
}
```

Gradle version catalog

Gradle version catalog je obyčejný soubor ve formátu TOML, který umožňuje snadněji přidávat a spravovat závislosti během celého projektu. Místo ručního přidávání závislostí a pluginů do každého modulu zvlášť, je možné shromáždit veškeré závislosti v tomto souboru a zadefinovat rovněž verzi, která má být pro tento modul použita napříč celým projektem.

Příklad toho, jak mohou být dříve zmíněné závislosti přepsány do version katalogu, je uveden ve výpisu kódu 4.4.

■ Výpis kódu 4.4 Version katalog

```
[versions]  
compose-plugin = "1.6.0"  
voyagerVersion = "1.0.0"  
  
[libraries]  
voyager-navigator = { module = "cafe.adriel.voyager:voyager-navigator",  
                      version.ref = "voyagerVersion" }  
  
[plugins]  
jetbrainsCompose = { id = "org.jetbrains.compose",  
                     version.ref = "compose-plugin" }
```

4.2 Design systém

Při implementaci design systému pomocí Compose Multiplatform byla snaha, co nejpřesněji napodobit navržený design systém ze sekce *Design system 3.5.2* tak, aby byl uživateli navozen maximální pocit jednoty s dosavadními informačními systémy města.

Od verze 1.6 Compose Multiplatform umožňuje přístup ke zdrojům ve společném kódu na všech podporovaných platformách, což výrazně usnadňuje implementaci navrženého design systému.

Barvy

Základní barvy pro světlý motiv byly zvoleny na základě navrženého design systému a zbylé barvy jako například barvy pro výplň pozadí nebo pro tmavý režim byly dopočítány ze základních barev nástrojem *Material Theme Builder*.

Pro použití vybraných barev v aplikaci bylo zapotřebí jednotlivé barvy nejprve zadefinovat následujícím způsobem:

■ Výpis kódu 4.5 Zadefinování barev

```
val md_theme_light_primary = Color(0xFFBC004C)
val md_theme_light_onPrimary = Color(0xFFFFFFFF)
val md_theme_light_primaryContainer = Color(0xFFFFD9DE)
val md_theme_light_onPrimaryContainer = Color(0xFF400015)
...
val md_theme_dark_primary = Color(0xFFFFB2BE)
val md_theme_dark_onPrimary = Color(0xFF660026)
val md_theme_dark_primaryContainer = Color(0xFF900039)
val md_theme_dark_onPrimaryContainer = Color(0xFFFFD9DE)
...
```

Takto zadefinované barvy je následně možné použít napříč aplikací k obarvení tlačítek, textů, ploch a dalších UI komponent.

Mimo jiné jsou barvy taktéž použity pro definování barevných motivů, které se definují následujícím způsobem:

■ Výpis kódu 4.6 Definice barevných motivů

```
private val LightColors = lightColorScheme(
    primary = md_theme_light_primary,
    onPrimary = md_theme_light_onPrimary,
    primaryContainer = md_theme_light_primaryContainer,
    onPrimaryContainer = md_theme_light_onPrimaryContainer,
    ...
)
private val DarkColors = darkColorScheme(
    primary = md_theme_dark_primary,
    onPrimary = md_theme_dark_onPrimary,
    primaryContainer = md_theme_dark_primaryContainer,
    ...
)
```

Díky takto zadefinovaným motivům je následně možné obarvit veškeré komponenty aplikace podle aktuálně používaného režimu.

V aplikaci byla proto implementována *Composable* funkce AppTheme (viz výpis kódu 4.7), která je z pohledu stromové struktury UI předkem veškerých vykreslovaných UI komponent. Díky této implementaci je tak možné měnit barevné režimy i za běhu aplikace.

■ Výpis kódu 4.7 Definice barevných motivů

```
@Composable
fun AppTheme(
    useDarkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable() () -> Unit
) {
    val colors = if (!useDarkTheme) {
        LightColors
    } else {
        DarkColors
    }

    MaterialTheme(
        colorScheme = colors,
        content = content
    )
}
```

Typografie

Vybraný Material Design definuje celkem 15 stylů písma, kde každé z nich má předem určený způsob použití. [56] Jelikož se jedná o styly písma přímo zvolené systémem Material Design, tak lze k těmto stylům přistupovat přes *Composable* funkci `MaterialTheme` z balíčku `androidx.compose.material3` popsanou v kódu 4.7. Na tomto příkladě je vidět implementace jednoho stylu písma - konkrétně `headlineMedium`, který je odvozen ze základních stylů písma definovaném v rámci sekce *Design system 3.5.2*.

■ Výpis kódu 4.8 Ukázka použití stylu písma

```
Text(
    text = news.title,
    style = MaterialTheme.typography.headlineMedium
)
```

4.3 Tvorba UI obrazovek

Jak již bylo zmíněno v kapitole *Architektura frameworku Compose Multiplatform 2.1.3*, tak k tvorbě UI se v Compose Multiplatform používají takzvané *Composable* funkce, které představují jednotlivé UI prvky zobrazované na obrazovce koncového zařízení.

Tyto *Composable* funkce, které přestavují stavební bloky celého UI mohou sloužit k reprezentaci jednotlivých tlačítek (`Button()`), textů (`Text()`) obrázků `Image()`, ale také celých souborů těchto prvků jako například celé obrazovky.

Mimo těchto funkcí je UI možné tvořit ještě pomocí takzvaných Layout funkcí, mezi které se řadí například `Column`, která vnořené *Composable* funkce uspořádává do sloupce nebo `Row`, která vnořené elementy řadí naopak do rádku.

Jelikož se tato sekce věnuje právě tvorbě UI jednotlivých obrazovek, na následujícím výpisu kódu 4.9 je ukázáno, jak mohou být do sebe dříve zmíněné *Composable* funkce zanořeny, čímž

tvoří takzvaný sémantický strom, taktéž popsaný v kapitole *Architektura frameworku Compose Multiplatform 2.1.3*

■ **Výpis kódu 4.9** Příklad tvorby UI pomocí frameworku Compose Multiplatform

```
@Composable
fun HomeScreen {
    // Přidá vnitřní okraj kolem celé zprávy
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(DrawableResource("profile_picture.jpg")),
            contentDescription = "Contact profile picture",
            modifier = Modifier
                // Nastaví velikost obrázku na 40 dp
                .size(40.dp)
                // Ořízne obrázek do kruhu
                .clip(CircleShape)
        )

        // Přidá horizontální mezeru mezi obrázek a sloupec
        Spacer(modifier = Modifier.width(8.dp))

        Column {
            Text(text = msg.author)
            // Přidá vertikální mezeru bez texty
            Spacer(modifier = Modifier.height(4.dp))
            Text(text = msg.body)
        }
    }
}
```

Obsahem následujících sekcí je ukázka UI komponent, které byly použity k implementaci navrženého UI a celkově dává náhled na to, jak je deklarativní UI pomocí Compose Multiplatform tvořeno. K tvorbě celého uživatelského rozhraní bylo nejprve přistupováno na základě vytvořených drátových modelů (viz kapitola *Tvorba drátových modelů 3.5.1*) a následně bylo pomocí implementovaného design systému stylizováno tak, aby odpovídalo navržené podobě UI z kapitoly *Mockup modely 3.5.3*.

4.3.1 Domovská obrazovka

Uživatelské rozhraní domovské obrazovky je kompletně implementováno pomocí frameworku Compose Multiplatform. Ani při jeho implementaci nezblo potřeba přistoupit k nativnímu řešení a to ani v případě platformy iOS nebo desktop.

K implementaci posuvného řádku, který slouží pro zobrazení událostí byla použita vlastní *Composable* komponenta zvaná `EventLazyRow` (viz výpis kódu 4.10), která je založená na základní UI komponentě zvané `LazyRow`. Tato komponenta se obvykle používá v případech, že je na obrazovku potřeba efektivně vykreslit velkého množství položek (kolekce položek), čož je přesně případ vykreslení událostí. Komponenta `LazyRow` totiž vykresluje pouze ty položky, které jsou aktuálně viditelné, čímž nezpomaluje načítání obsahu. V případě většího množství událostí pracuje na podobném principu a výrazně tak šetří zdroje potřebné k vykreslení.

■ Výpis kódu 4.10 Implementace posuvného řádku pomocí `LazyRow`

```
@Composable
fun EventLazyRow(category: EventCategory, state: EventsState) {
    val scrollState = rememberLazyListState()

    Column {
        Text( ... )
        LazyRow(
            state = scrollState,
        ) {
            ...
            items(eventList) { event ->
                EventItem(event = event, onClick = {
                    navigator.push(EventDetailScreen(event))
                })
            }
        }
    }
}
```

Na stejném principu jako `LazyRow`, funguje i komponenta `LazyColumn`, která naopak posloužila pro vykreslení seznamu novinek.

Dále byla na této obrazovce implementována vlastní tlačítka zvaná `ImageButton()`, která například plní funkci přesměrování uživatele na příslušné informační kanaly města.

4.3.2 Obrazovka *Události*

V rámci obrazovky *Události* je používána stejná komponenta pro zobrazování událostí, která již byla aplikována na domovské obrazovce.

Každý z nich obsahuje události z jedné kategorie. Ty jsou předem připraveny ViewModelem, který při zavolení nějaké z filtrovacích událostí nejprve vybere události, které spadají do vymezeného časového intervalu a následně u nich zjistí, do jakých kategorií tyto události patří. Pro každou z těchto kategorií poté vytvoří jeden LazyRow, do kterého umístí příslušné události.

Uživatel může filtrovat události buďto podle předem zvolených časových rozmezí jako je *Dnes*, *Zítra* nebo může zvolit vlastní rozmezí pomocí kliknutí na ikonu kalendáře. Po kliknutí na ikonu kalendáře se uživateli zobrazí takzvaný *DatePicker*, který umožňuje zvolit požadované časového rozmezí.

K volbě časových rozmezí *Dnes* a *Zítra* byly zvoleny takzvané *Chips*, konkrétně `FilterChips`, které jsou navrženy k filtrování a uživatelé jsou na jejich použití zvyklí i z jiných aplikací, které používají Material Design.

Celá obrazovka je vzhledem k nepředvídatelnosti množství zobrazovaných událostí implementována jako rolovačí seznam, který se v případě potřeby rozšíří a umožní tak zobrazit veškeré události pro vybrané datové rozmezí.

4.3.3 Obrazovka *Parkování*

Hlavní UI komponentou navrženou pro obrazovku *Parkování* je komponenta sloužící k zobrazení mapových podkladů, která by zároveň umožňovala vykreslení překryvné vrstvy reprezentující rozmístění parkovacích zón. Taková knihovna, ale doposud nebyla pro použití na všech

požadovaných platformách dostupná, a proto bylo zapotřebí použít nativní způsob implementace.

Implementace mapových podkladů

Mapové podklady byly implementovány pouze pro platformu Android, pro kterou existuje nativní knihovna *Maps Compose* od společnosti Google, která poskytuje Jetpack Compose komponenty pro práci s (Google) Maps SDK.

K integraci mapových podkladů do aplikace bylo využito UI komponenty `GoogleMap` (viz výpis kódu 4.11) z balíčku `com.google.maps.android.compose`, která umožňuje vložení mapových podkladů na platformě Android. K propojení této nativní části se zbytkem aplikace byla použita `expect/actual` deklarace.

■ Výpis kódu 4.11 GoogleMap element

```
GoogleMap(  
    modifier = Modifier.fillMaxSize(),  
    properties = MapProperties(  
        mapType = MapType.NORMAL,  
        mapStyleOptions = MapStyleOptions(MapStyle.JSON_LIGHT)  
    ),  
    cameraPositionState = cameraPositionState  
) {...}
```

Implementace motivu mapy

Aby byla implementace motivů z kapitoly *Design systém 4.2* kompletní, bylo zapotřebí přizpůsobit i mapové podklady tak, aby se jejich motiv měnil v závislosti na aktuálním motivu zařízení.

Z tohoto důvodu a mimo jiné pro zlepšení přehlednosti byly na mapové podklady implementovány určité styly (viz výpis kódu 4.12) [57]. Ty jednotlivým částem mapových podkladů přiřazují konkrétní barvy, díky čemuž bylo docíleno jednotnějšího vzhledu. Původní mapové podklady používaly kobarvení jednotlivých mapových prvků několik druhů barev a to mělo za důsledek, že na něm byly vyznačené parkovací zóny hůře viditelné.

■ Výpis kódu 4.12 Motiv mapy ve formátu JSON

```
{  
    "featureType": "road.highway",  
    "elementType": "labels.text.fill",  
    "stylers": [  
        {  
            "color": "#616161"  
        }  
    ]  
,  
    ...}
```

Získání a zpracování mapových dat

Mapová data v podobě seznamu parkovacích zón a jejich podrobnostech jako jsou jednotlivé názvy, typy, popisy a umístění parkovacích zón byla získána od města Příbrami. Poskytnutá data byla uvedena ve formátu Keyhole Markup Language (KML), který je primárně určen pro publikaci a distribuci geografických dat. Tento formát ke svému zápisu používá syntaxi jazyka XML a z toho důvodu byl k jeho převedení do Kotlin objektů použit stejný nástroj jako pro převod veškerých dat přicházejících z informačních systémů města.

Zobrazení parkovacích zón na mapě bylo implementováno pomocí *Composable* funkce *Polygon*, která je poskytována v rámci stejného balíčku jako dříve zmíněná komponenta *GoogleMap*.

Pro vystředění vybrané parkovací zóny na obrazovce byla napsána funkce pro vycentrování mapy tak, aby se zvolená zóna pokaždé umístila do středu obrazovky. Implementovaná funkce tedy hledá geometrický střed zadaných souřadnic $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, jenž tvoří mnohoúhelník parkovací zóny a pomocí aritmetického průměru vypočítává souřadnice středu zóny následujícím způsobem:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \quad \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

Výsledné souřadnice (\bar{x}, \bar{y}) , jsou následně použity k vycentrování mapy.

Dále byla tato obrazovka rozšířena o vyhledávací pole, které bylo již možné implementovat v rámci společné části aplikace. To dává uživateli možnost jednoduchého vyhledání parkovací zóny podle názvu nebo čísla parkovací zóny.

Poslední implementovanou komponentou v rámci obrazovky *Parkování* je takzvaný *Bottom sheet*, který je zobrazen v případě, že uživatel vybere nebo vyhledá parkovací zónu, pro kterou si přeje zobrazit podrobné informace. Konkrétně byl pro implementaci vybrán *BottomSheetScaffold*, který oproti jeho modálním variantám (např. *ModalBottomSheet*), umožňuje souběžnou integraci mezi tímto výsuvným panelem a mapovým podkladem.

4.4 Navigace

Navigace je klíčovou součástí moderních aplikací uživatelského rozhraní, která uživatelům umožňuje pohybovat se mezi různými obrazovkami či částmi aplikace. Hlavním cílem navigace je poskytnout uživatelům intuitivní a plynulý způsob průchodu obsahu a samovolně zlepšit jejich provádění akcí v aplikaci.

Aktuálně však komponenta navigace ze sady knihoven Jetpack Compose není k dispozici, a proto je potřeba zvolit nějakou alternativu poskytovanou třetí stranou. [58] Výčet aktuálně doporučených knihoven pro implementaci multiplatformní navigace je uveden v oficiální Kotlin multiplatform dokumentaci a zahrnuje například navigaci Voyager, Decompose nebo Appyx.

K implementaci byla vybrána navigace Voyager a to především díky přehledné dokumentaci, intuitivní implementaci a jednoduchému principu navigace založeném na zásobníku, který pro aktuálně implementovanou aplikaci plně dostačuje. V budoucnu by však měla být dostupná i oficiální navigace, ale do té doby je potřeba vybrat z alternativních knihoven.

4.5 Použité technologie

Následující sekce jsou věnovány konkrétním technologiím, které byly pro tvorbu uživatelského rozhraní použity. Jedná se o soubor multiplatformních technologií, které byly převážně použity k implementaci UI, ale zároveň jsou zde zmíněny některé důležité technologie, na kterých je postavena aplikační logika aplikace.

4.5.1 Vrstva uživatelského rozhraní

Voyager

Voyager je multiplatformní navigační knihovna, která umožňuje přecházet mezi různými obrazovkami a destinacemi v multiplatformních aplikacích. [59] Jejím hlavním cílem je poskytnout jednotné rozhraní především pro navigaci na mobilních platformách jako jsou Android a iOS. Voyager zároveň dokáže maximalizovat sdílený kód a snižovat duplikaci kódu.

V rámci implementované aplikace byla navigace Voyager použita především k implementaci základní navigace mezi jednotlivými obrazovkami. Příkladem takové navigace jsou například veškeré možné přechody v rámci domovské obrazovky. Tato navigace vyžívá ke své funkci takzvané *Stack API*, jehož implementace funguje na principu zásobníku. S tím, že pokud uživatel vstoupí na novou obrazovku (například přejde na stránku zobrazující detail vybrané události), tak je do tohoto zásobníku přidán odkaz na takzvaný `SnapshotState`, který uchovává stav právě opuštěné obrazovky. Znovu obnoven je v případě, že se uživatel na tuto obrazovku znovu vrátí. Zároveň tento zásobník slouží k tomu, aby navigace vždy věděla na jakou stránku se má vrátit (poslední v zásobníku), když uživatel provede akci zpět například pomocí tlačítka zpět na platformě Android nebo jakéhokoliv tlačítka zpět implementovaného v rámci aplikace. *Stack API* zároveň umožňuje použití několika funkcí jako jsou například funkce `push`, `replace`, `replaceAll` používané pro průchod vpřed nebo funkce `pop`, `popAll`, `popUntil` pro zpětný návrat. Díky této implementaci je možné procházené obrazovky do sebe libovolně zanořovat a v případě, že je potřeba nějakým způsobem upravit stav předchozích stavů v zásobníku, lze nad tímto zásobníkem zavolat některou z dříve uvedených funkcí.

Způsob navigace založený na *Stack API* však není vhodný pro navigování mezi jednotlivými záložkami aplikace (lišta ve spodní části obrazovky), a proto je tato část navigace implementována pomocí takzvané *Tab navigation*, kterou knihovna Voyager, taktéž poskytuje. V případě použití záložek se průchod obrazovkami nezapisuje do zásobníku, jako tomu bylo v předchozím případě, ale místo toho poskytuje pouze vlastnost `current`, díky které je možné nastavit nebo získat aktuální záložku. [60]

Příklad použití tohoto typu navigace je na následující ukázce kódu 4.13, díky které je vidět, jak jsou jednotlivé záložky uspořádány v rámci spodní navigační lišty (funkce `BottomNavigation()`).

■ **Výpis kódu 4.13** Ukázka použití navigace založené na záložkách

```
TabNavigator(
    tab = HomeTab
) {
    Scaffold(
        modifier = Modifier.fillMaxSize(),
        bottomBar = {
            BottomNavigation(
                backgroundColor = MaterialTheme.colorScheme.background
            ) {
                TabNavigationItem(HomeTab)
                TabNavigationItem(EventsTab)
                TabNavigationItem(ParkingTab)
                TabNavigationItem(MoreTab)
            }
        },
        content = { CurrentTab() },
    )
}
```

Coil

Coil je knihovna pro načítání a zobrazování obrázků napsaná v jazyce Kotlin, která byla původně napsaná pouze pro platformu Android, avšak od verze 3.0.0 z prosince roku 2023 existuje alpha verze podporující i platformy JVM, iOS, macOS, nebo Javascript.

Knihovna poskytuje jednoduché API pro načítání obrázků z různých zdrojů, včetně URL adres, souborů a zdrojů z paměti. Díky tomu je integrace obrázků do aplikace snadná a přizpůsobitelná potřebám projektu. Knihovna Coil dále nabízí automatické dohledávání velikosti obrázků na základě rozměrů zobrazení, což pomáhá optimalizovat paměť a výkon aplikace. Tato funkce umožňuje načítat obrázky ve správné velikosti a snižuje zátěž na síťové spojení. Knihovna podporuje širokou škálu formátů obrázků, včetně PNG, JPEG, GIF, SVG a WebP, což umožňuje pracovat s různými typy obrázků bez nutnosti dalšího přizpůsobování.

Další výhodou knihovny Coil je možnost efektivního cachování obrázků, které pomáhá snižovat čas načítání a zlepšuje uživatelský zážitek.

V rámci implementované aplikace jsou téměř všechny obrázky načítány z URL adres, a proto je možnost cachování zásadní hlavně v případech, kdy aplikace nemá přístup k internetu.

Pro ukázku *Composable* funkce, která je poskytována knihovnou Coil a byla v rámci aplikaci použita k načítání veškerých obrázků z internetu, slouží následující výpis kódu 4.14.

■ **Výpis kódu 4.14** Composable funkce `AsyncImage` poskytovaná knihovnou

```
AsyncImage(
    model = news.thumbnailUri,
    contentDescription = "News Image",
    contentScale = ContentScale.Crop,
    modifier = Modifier.aspectRatio(4f / 3f)
)
```

Kromě těchto druhů navigace byla knihovna Voyager ještě použita k implementaci ViewModelů, které je možné prostřednictvím integrace této knihovny použít napříč všemi platformami. V případě knihovny Voyager jsou tyto ViewModely nazývány jako *ScreenModels* a podobně jako u klasických ViewModelů umožňují například uchovat data i při změnách konfigurace (například otočení obrazovky). [61]

4.5.2 Vrstva aplikační logiky

Ktor

Ktor je open-source framework vytvořený společností JetBrains pro tvorbu jak serverových, tak klientských aplikací prostřednictvím programovacího jazyka Kotlin. Ktor byl vybrán především kvůli možnosti využití multiplatformního asynchronního HTTP klienta, pomocí kterého je možné zadávat HTTP požadavky a zároveň zpracovávat případné odpovědi.

Pro dotazování byl použit multiplatformní Engine CIO, jehož benefitem je možnost používat Ktor na platformách Android, JVM a všech platformách, které podporuje Kotlin/Native.

■ Výpis kódu 4.15 Konfigurace HTTP klienta

```
private val client = HttpClient(CIO) {
    engine {
        maxConnectionsCount = 1000
        requestTimeout = 5000
        ...
    }
}
```

Po nakonfigurování HTTP klienta je již možné zasílat běžné HTTP dotazy skrze metodu `post`, `get`, `put` nebo `delete`. Na následující ukázce kódu 4.16 je pomocí nakonfigurovaného klienta z ukázky kódu 4.15 zaslán HTTP požadavek na webový server města.

■ Výpis kódu 4.16 Zaslání požadavku

```
suspend fun getEventsXml(): EventsXmlDto {
    val response: String = client.get("https://kalendar.pribram.eu/...").body()
    return format.decodeFromString<EventsXmlDto>(response.cleanUpEventXml())
}
```

SQLDelight

SQLDelight je multiplatformní knihovna pro práci s relačními databázemi v aplikacích napsaných v jazyce Kotlin. Jedná se o moderní knihovnu, která umožňuje vytváření a správu SQL dotazů s pomocí typově bezpečných prostředků poskytovaných jazykem Kotlin.

Hlavní funkcionalitou knihovny SQLDelight je automatické generování Kotlin tříd a funkcí na základě definovaných SQL schémat. Díky tomu je možné psát SQL dotazy pomocí standardní SQL syntaxe. SQLDelight se postará o vygenerování příslušných funkcí v jazyce Kotlin, které umožňují snadný a bezpečný přístup k databázi. Jediným rozdílem oproti standardním SQL dotazům je označení jednotlivých SQL dotazů takzvaným štítkem (viz `selectNewestEvents`: ve výpisu kódu 4.17), skrze který je vygenerována příslušná Kotlin funkce.

■ **Výpis kódu 4.17** SQL dotaz pro získání všech novinek v databázi

```
selectNewestEvents:
SELECT *
FROM Events
ORDER BY startTime DESC
LIMIT 30;
```

Vygenerované funkce (dotazy) lze následně nalézt ve složce `build/generated/sqlodelight/...` a pro dotaz z výpisu kódu 4.17 vypadá jeho zkrácená podoba následovně:

■ **Výpis kódu 4.18** SQL vygenerovaný dotaz

```
public fun <T : Any> selectNewestEvents(mapper: (
    id: Long,
    categories: String,
    place: String,
    title: String,
) -> T): Query<T> = Query(-704_046_957, arrayOf("Events"), driver, "Events.sq",
    "selectNewestEvents",
    """
|SELECT Events.id, Events.categories, Events.place, Events.title
|FROM Events
|ORDER BY startTime DESC
|LIMIT 30
""".trimMargin() { cursor ->
    mapper(
        cursor.getLong(0)!!,
        cursor.getString(1)!!,
        cursor.getString(2)!!,
        cursor.getString(3)!!
    )
}
```

I přesto, že se jedná o multiplatformní knihovnu a většina kódu tak může být psána v balíčku sdíleném mezi všemi platformami, je zde i část z následující ukázky kódu 4.19, kterou je pro veškeré implementované platformy potřeba implementovat nativně. Konkrétně se jedná o databázové ovladače, k jejichž implementaci jsou zapotřebí nativní knihovny a musí být proto implementovány v balíčcích jednotlivých platform.

■ **Výpis kódu 4.19** Nativní databázový ovladač pro platformu *desktop*

```
actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        val driver: SqlDriver = JdbcSqliteDriver("jdbc:sqlite:AppDatabase.db")
        AppDatabase.Schema.synchronous().create(driver)
        return driver
    }
}
```

Koin

Koin je moderní framework pro vkládání závislostí (zkráceně DI - z anglického „dependency injection“). Jedná se o lehkou a snadno použitelnou alternativu ke knihovnám jako je například Hilt pro platformu Android.

Hlavním principem Koinu je deklarativní přístup k definici závislostí pomocí takzvaných modulů. V modulu se specifikují veškeré závislosti a jejich vytváření. Děje se tak prostřednictvím jednoduchých funkcí nebo takzvaných singletons - případně factories. Ty, jak název napovídá, odkazují na patřičné návrhové vzory, na jejichž základě se dané instance vytvářejí.

Během této aplikace byl Koin použit například pro vložení databázového ovladače (viz ukázka kódu 4.20), které museli být tvořeny za pomocí expect actual deklarací. Využití DI bylo v tomto případě vhodné předevší proto, že převážná většina databázové logiky je napsána v sdílené logice a v rámci nativní logiky byl tak implementován pouze nativní databázový ovladač. Ten musel být následně pro každou platformu vložen do společné třídy implementující Databázi (`DatabaseDaoImpl(databaseDriverFactory: DatabaseDriverFactory)`).

Dalšími důležitými komponentami, které byly do aplikace vkládány pomocí DI byly View-Modely a celá třída úložiště, na které jsou ViewModely závislé.

■ Výpis kódu 4.20 DI databázového ovladače pomocí Koinu

```
class MainActivity : ComponentActivity() {

    private val dbDriverFactoryModule = module {
        single { DatabaseDriverFactory(applicationContext) }
    }

    init {
        loadKoinModules(dbDriverFactoryModule)
    }
    ...
}
```

4.6 Logika uživatelského rozhraní

V této sekci je ukázáno, jakým způsobem byla implementována logika chování UI, jejíž části byly navrženy v kapitole *Vrstvy architektury 3.4*.

4.6.1 Stav uživatelského rozhraní

Tato sekce slouží k přiblížení toho, jak je takzvaný *UI State* (z dříve zmíněné kapitoly 3.4) implementován v ukázkové aplikaci.

Z pohledu UI se jedná o důležitou část aplikace, o jejíž aktualizaci se stará později zmíněný ViewModel (viz ViewModel4.6.3) neboli ScreenModel v rámci implementované aplikace. Pro zachování stavu jednotlivých obrazovek byla použita datová třída, která v sobě umožňuje uchovávat veškeré možné stavy dané obrazovky. Pro ukázku toho, jak byla taková datová třída implementována, byla vybrána obrazovka událostí (viz výpis kódu 4.21), která v sobě například uchovává informace typu - jaké filtry událostí jsou aktuálně zvoleny, zdali je otevřeno dialogové okno pro výběr datového rozmezí nebo zdali jsou aktuálně načítána nějaká data.

■ **Výpis kódu 4.21** Implementace stavu obrazovky *Události*

```
data class EventsState(
    val events: StateFlow<List<Event>> = MutableStateFlow(emptyList()),
    val selectedFilterOption: FilterOption = FilterOption.TODAY,
    val isDatePickerOpen: Boolean = false,
    val isFetchingEvents: Boolean = false,
    val filteredEventCategories: List<EventCategory> = emptyList(),
    ...
)
```

Takto zadefinované stavy je následně možné používat v rámci jednotlivých obrazovek, které jsou při jakémkoliv změně tohoto stavu znovu sestaveny.

■ **Výpis kódu 4.22** Integrace stavu obrazovky *Události*

```
val screenModel = getScreenModel<EventsScreenModel>() // ViewModel
val state by screenModel.state.collectAsState()
...
FlowColumn {
    state.filteredEventCategories.forEach {
        EventLazyRow(it, state)
    }
}
```

4.6.2 UI události

Jak již bylo zmíněno, tak nejčastějším typem UI události jsou takzvané uživatelské události, produkované uživatelem při interakci s aplikací. Tyto události jsou zpracovány aplikací pomocí metod jako je například metoda `onClick()`, která je často typickým příkladem toho, jak jednotlivé UI komponenty obsluhují patřičné uživatelské události.

V případě takovýchto metod je pak vhodné uplatnit reprezentaci UI události například v podobě datových objektů zadefinovaných ve výpisu kódu 4.23.

■ **Výpis kódu 4.23** Použití stavu v aplikaci

```
sealed interface EventsEvent {
    data object OnFilterTodayIsSelected : EventsEvent
    data object OnFilterDateIsSelected : EventsEvent
    ...
}
```

Takto zadefinované objekty je již možné použít u UI obrazovek například následujícím způsobem 4.24.

■ **Výpis kódu 4.24** Použití stavu v aplikaci

```
FilterChip(
    onClick = { screenModel.onEvent(EventsEvent.OnFilterTodayIsSelected) },
    label = {
        Text(text = FilterOption.TODAY.filterName)
    },
)
```

Na ukázce kódu 4.24 je zároveň vidět propojenosť UI událostí s příslušným ViewModelem, čemuž se více věnuje následující sekce *ViewModel*.

4.6.3 ViewModel

V případě implementované aplikace mají ViewModely jasně definovanou roli a to zpracovávat události přicházející z UI a na jejich základě by vždy měli měnit stav aplikace. [62] V tomto duchu jsou implementovány všechny ViewModely. Díky tomu je možné zaručit, že stav uživatelského rozhraní nebude změněn například v důsledku konfiguračních změn.

Co se implementace jednotlivých ViewModelej týká, tak vesměs všechny mají podobnou strukturu. K vytvoření instance ViewModelu (ScreenModelu) je zapotřebí dodat příslušnou instanci úložiště (například `RepositoryImpl`) a následně je již možné implementovat příslušné funkce zpracovávající UI události (viz ukázka kódu 4.25).

■ **Výpis kódu 4.25** Implementace ViewModelu

```
fun onEvent(event: EventsEvent) {
    when (event) {
        EventsEvent.OnFilterTodayIsSelected -> {
            screenModelScope.launch {
                val today: LocalDate = Clock.System.todayIn(...)
                ...
                _state.update {
                    it.copy(
                        selectedFilterOption = FilterOption.TODAY,
                        ...
                    )
                }
            }
        }
    }
}
```

Kapitola 5

Testování

Mezi základní dva druhy testování UI patří manuální a automatizované testování. Za účelem analýzy frameworku Compose Multiplatform se tato kapitola věnuje především UI testům, které tento framework přímo podporuje, a tím jsou právě automatizované UI testy. Ty jsou dále rozděleny na takzvané *Unit* testy neboli testy jednotlivých UI komponent a takzvané *End-to-End* testy, které simulují reálný průchod aplikací - například uživatelem. Před samotnou ukázkou způsobu implementace těchto testů, se tato kapitola věnuje ještě samotným konceptům, na kterých jsou oba tyto typy testování postaveny a ukázce testovacích případů, které byly k implementaci těchto testů použity.

5.1 Možnosti testování UI v Compose Multiplatform

Od verze 1.6.0 Compose Multiplatform umožňuje testování UI na všech podporovaných platformách. [63] Testování aplikací založených na frameworku Compose Multiplatform je stejně jako tvorba samotného UI založeno na Jetpack Compose, a využívá proto i stejných konceptů.

Mezi tyto klíčové koncepty testovaní UI se řadí následující:

Sémantika

Jak již bylo zmíněno několikrát, tak UI frameworku Compose Multiplatform je založeno na stromové struktuře elementů, a z toho důvodu i samotné UI testy používají tuto strukturu k integraci s jednotlivými prvky této hierarchie. K rozlišení jednotlivých prvků stromu se používají buďto samotné popisy jednotlivých elementů jako je například název tlačítka nebo lze také využít popisků sloužících k přístupnosti (v kódu označovaných pomocí štítků `contentDescription`).

Testování rozhraní

K samotnému testovaní UI elementů se používají tři hlavní principy, které se označují jako *Finders*, *Assertions* a *Actions*.

V první fázi testovaní je potřeba samotné testované elementy vyhledat a k tomu slouží právě *Finders*, které umožňují najít uzel nebo uzly ve stromové UI struktuře pomocí odpovídajícího sémantického stromu. Případně je dané elementy možné najít pomocí takzvaných *Matchers*, které umožňují najít hledané UI komponenty například podle aktuálního stavu (funkce `isEnabled()`, `hasProgressBarRangeInfo()` a další).

Po nalezení potřebných komponent je možné přistoupit například k takzvaným *Actions*, prostřednictvím kterých je možné vykonávat akce podobné těm, které vykonávají uživatelé (funkce `performClick()`, `performScrollTo()` a další).

Pro kontrolu správnosti provedení těchto akcí se používají takzvaná tvrzení (*Assertions*), které ověří zdali vybrané UI prvky mají požadované atributy. [64]

Synchronizace

Dnešní asynchronní povaha mobilních aplikací a frameworků často ztěžuje psaní spolehlivých a opakovatelných testů. Testovací rámce musí v případě asynchronních operací čekat, než na ni aplikace zareaguje a až následně vykonávat příslušné testovací operace. Tyto reakce testovacích rámců, jsou v případě moderních testovacích rámců jako je v případě Compose, již běžná a automatická věc. Přesto jsou zde ale situace, při kterých je synchronizaci testů s uživatelským rozhraním potřeba explicitně řídit.

Takovým příkladem může být například situace, kdy aplikace čeká na odpověď webového serveru. V takové situaci je v rámci testů možné definovat funkce jako například `waitForNodeCount(matcher, count, timeoutMs)`, které pozdrží testovaní do doby, než bude aplikace připravena s pokračovaním testů.

5.2 Testovací případy

Testovací případ (zkráceně TC - z anglického „test case“) je sada kroků nebo akcí, které jsou prováděny při testování softwarových produktů. Cílem zadefinování testovacích případů je ověřit, zda se testovaná aplikace chová podle očekávání a splňuje předem definované požadavky.

Každý testovací případ obvykle obsahuje následující prvky:

- **Popis:** Stručný popis toho, co daný testovací případ testuje.
- **Kroky:** Konkrétní kroky, které musí být k provedení testu provedeny.
- **Očekávané výsledky:** Popis očekávaných výsledků testu po dokončení kroků.

Na základě této struktury byly sepsány následující testovací případy:

TC1: Zobrazení aktuálních novinek

Zobrazení aktuálních novinek a jejich detailu.

1. Kroky:

- a. Navigace na stránku zobrazující aktuální novinky.
- b. Ověření, že systém zobrazil seznam aktuálních novinek.
- c. Kliknutí na konkrétní novinku.
- d. Ověření, že systém zobrazil detaily vybrané novinky.

2. Očekávaný výstup:

- Seznam aktuálních novinek byl zobrazen.
- Po kliknutí na konkrétní novinku byl zobrazen její detail.

TC2: Odkázání na zaplacení parkovacích poplatků

Zobrazení parkovacích zón a následné vybrání parkovací zóny s odkazem na zaplacení parkovacího poplatku.

1. Kroky:

- a. Navigace na stránku zobrazující parkovací zóny.
- b. Kliknutí do vyhledávacího pole.
- c. Zvolení požadované parkovací zóny.
- d. Ověření, že systém zobrazil vybranou parkovací zónu.
- e. Kliknutí na tlačítko „zaplatit“.
- f. Ověření, že systém uživatele přesměroval na platební portál města.

2. Očekávaný výstup:

- Seznam parkovacích zón byl zobrazen.
- Dialogový panel byl zobrazen.
- Uživatel byl přesměrován na platební portál města.

TC3: Kontrola stavu po zrušení akce výběru datového rozsahu

Zobrazení dialogového okna pro výběr datového rozsahu a následné zrušení prováděné akce.

1. Kroky:

- a. Navigace na stránku zobrazující události.
- b. Ověření, že systém zobrazil filtrační možnosti.
- c. Zvolení tlačítka pro výběr datového rozsahu.
- d. Ověření, že systém zobrazil patřičné dialogové okno.
- e. Zvolení tlačítka zrušit pro zrušení akce výběru datového rozsahu.
- f. Ověření stavu filtračních možností.

2. Očekávaný výstup:

- Systém zobrazí dialogové okno pro výběr datového rozsahu.
- Filtrační možnosti zůstanou ve stejném stavu jako před provedou akcí.

5.3 Implementace testů

5.3.1 Testy jednotlivých UI komponent

Před tím, než bylo přistoupeno k testování UI dle výše zmíněných scénářů, bylo nejprve implementováno několik testů pro ověření správné funkcionality a vzhledu jednotlivých komponent jako jsou tlačítka, textová pole, seznamy, obrázky atd.

Účelem těchto testů je zajistit, že každá komponenta aplikace funguje správně a že se zobrazuje v souladu s návrhem a specifikacemi. To zahrnuje ověření interakcí s komponentou jako je klikání, vyplňování políček, posouvání seznamů atd., stejně jako ověření vzhledu komponenty v různých situacích jako je například změna orientace zařízení.

Pro ukázkou metod používaných pro ověření správnosti zobrazení jednotlivých UI komponent slouží následující ukázka kódu 5.1.

■ **Výpis kódu 5.1** Metody pro testování UI komponent

```
@OptIn(ExperimentalTestApi::class)
@Test
fun check_properties_check_return() = runComposeUiTest {
    ...
    //Ověří, že daná komponenta byla nalezena a je součástí stromu UI komponent.
    onNodeWithText("Závady").assertExists()
    //Ověří, že je daný uzel sémantického stromu zobrazen na obrazovce.
    onNodeWithText("Závady").assertIsDisplayed()
    //Ověří, že rozvržení daného uzlu má výšku větší než 100 dp
    onNodeWithText("Závady").assertHeightIsAtLeast(100.dp)
}
```

5.3.2 End-to-end testy

End-to-end (E2E) testy jsou typem testování, který simuluje reálné uživatelské scénáře a ověřuje fungování celé aplikace z pohledu uživatele. Tyto testy prověřují integraci jednotlivých komponent aplikace a zajistují, že aplikace pracuje korektně a očekávaným způsobem od začátku až do konce vykonávaného procesu.

Při E2E testování je často využíván emulátor nebo reálné zařízení, aby se testování co nejvíce blížilo reálným uživatelským podmínkám.

Cílem E2E testů je zajistit, že aplikace funguje spolehlivě a správně napříč všemi vrstvami a komponentami a že uživatelé mohou dosáhnout svých cílů bez problémů.

Při implementaci těchto testů bylo postupováno na základě testovacích scénářů uvedených v sekci *Testovací případy 5.2* a pro ukázkou implementace byl zvolen test: 5.2

■ **Výpis kódu 5.2** Implementace E2E testu

```
fun nav_to_events_open_calendar_press_cancel_check_buttons_state() {

    val dbDriverFactoryModule = module {
        single { DatabaseDriverFactory(instrumentationContext) }
    }
    loadKoinModules(dbDriverFactoryModule)

    rule.setContent { App() }

    rule.onNodeWithText("Události").performClick()
    rule.onNodeWithText("Dnes").assertIsSelected()

    rule.onNodeWithContentDescription("calendar icon").performClick()
    rule.onNodeWithText("Zrušit").performClick()

    rule.onNodeWithText("Dnes").assertIsSelected()
    rule.onNodeWithText("Zítra").assertIsNotSelected()
    rule.onNodeWithContentDescription("calendar icon").assertIsNotSelected()
}
```

Kapitola 6

Závěr

6.1 Závěr o použitelnosti frameworku

Podporované platformy

Co se pokrytí aktuálně podporovaných platforem týká, tak Compose Multiplatform pokrývá většinu z aktuálně nejzádanějších platforem pro vývoj multiplatformních aplikací a dá se tak očekávat, že po uvolnění stabilních verzí pro veškeré aktuálně nabízené platformy se jeho obliba ještě zvýší.

Výkon a optimalizace

V oblasti výkonu je dle sekce *Porovnání výkonu* 2.2.1 Compose Multiplatform konkurenčeschopný s ostatními multiplatformními frameworky, ale kdyby bylo srovnání výkonu provedeno na rozsáhlejších projektech, tak je možné, že by rozdíl mezi naměřenými výsledky na platformě iOS byl mnohem výraznější.

Co se oblasti optimalizace týká, tak zde má Compose Multiplatform ještě poměrně velké nedostatky a to zejména co se týká plynulosti provozu na platformě iOS. Pro tu je však Compose Multiplatform stále ve verzi Alfa a z toho důvodu se dá očekávat brzké zlepšení. [41]

Optimalizace nástrojů pro vývoj

Pokud jde o připravenost nástrojů pro vývoj, tak mezi hlavními šesti problémy, které dle posledního oficiálního výzkumu KMP provází, jsou celkem tři, které se optimalizace nástrojů pro vývoj týkají. Konkrétně šlo o problémy s nastavením sestavování, problémy s vývojovým prostředím a o potíže co se rychlosti sestavování týká. [37]. Aktuálně nelze objektivně říci jaké problémy již byly vyřešeny a jaké ne, ale v rámci procesu implementace této aplikace byly největší problémy detekovány právě u podpory vývojových prostředí.

Z testovaných prostředí se vývojové prostředí Android Studio jevilo z pohledu optimalizace lépe optimalizované než konkurenční vývojové prostředí Fleet. Na druhou stranu z pohledu funkcí podporuje Fleet některé užitečné funkce jako například *Preview* [65], které buďto Android Studio nepodporovalo nebo se u nich vyskytovali zásadní problémy, kvůli kterým je nebylo možné použít. Dalším příkladem může být například spouštění jednotlivých testů pomocí takzvaných *Gutter Icons*, které pro multiplatformní testy na platformě Android nebylo možné vůbec použít. [66]

Dokumentace

Z pohledu dokumentace má Compose Multiplatform významné nedostatky, ale díky tomu, že je z velké části odvozen z frameworku Jetpack Compose, tak nejsou tyto nedostatky nakonec tak

zásadní. Většina potřebných částí pro tvorbu uživatelského rozhraní je totiž dobře dokumentovaná společností Google. Co se přidružené technologie KMP týká, tak ta v porovnání s ostatními technologiemi v oblasti dokumentace stejně jako Compose Multiplatform zaostává i přesto, že se již nachází ve stabilní verzi.

Náročnost implementace

Vyjma výše zmíněných problémů, celý proces implementace frameworku Compose Multiplatform proběhl téměř bez kombinací. Tato bezproblémovost a možnost postupné integrace do již stávajících aplikací, které aktuálně často přechází od imperativních způsobů zápisu UI k deklarativním způsobům, může tomuto frameworku pomoci nabýt v budoucnu na popularitě.

Shrnutí

Při závěrečném hodnocení frameworku Compose Multiplatform se dá říci, že je aktuálně vhodným nástrojem pro vývoj multiplatformních aplikací, které nevyžadují úplnou stabilitu použitých technologií, ale za to upřednostňují jednoduše použitelné technologie, které poskytují poměrně rychlou možnost tvorby UI s budoucím výhledem co se stability týká.

6.2 Shrnutí dosažených výsledků

Co se vytvořené aplikace týká, tak ta byla úspěšně přizpůsobena konkrétnímu městu a napojena na jeho infrastrukturu pro získávání dat o událostech, aktualitách a parkovacích zónách. Díky implementaci lokální databáze je navíc aplikace použitelná i v offline režimu, címž usnadňuje přístup všem uživatelům a urychluje tak přehled o aktuálním dění ve městě. Navržená architektura je zároveň dostatečně flexibilní, aby pokryla případně změny ve struktuře města.

A co se frameworku Compose Multiplatform týká, tak zde se podařilo na základě prakticky sestavené aplikace potvrdit, že je možné aby převážná většina kódu byla sdílena mezi vybranými platformami. a také se podařilo docílit toho, že množství sdíleného kódu bylo násobně větší oproti dosavadním aplikacím používaných v praxi. Množství sdíleného kódu se zvýšilo především díky sdílení UI pomocí Compose Multiplatform, ale také díky využití většího množství multiplatformních KMP knihoven. Zároveň je ale nutné zohlednit fakt, že rozsah zkušební aplikace nedosahuje takových rozměrů jako aktuálně používané aplikace v praxi. Každopádně lze i tak konstatovat, že se i v současné fázi vývoje se podařilo navrhnout aplikaci tak, aby fungovala na více platformách a která z pohledu UI splňuje veškeré stanovené požadavky, které mohou být pro aplikace menších rozměru dostačující.

6.3 Návrhy pro budoucí vývoj a výzkum

Z pohledu aplikace jsou zde velké možnosti pro budoucí vývoj jako například propojení aplikace s dalšími systémy města, implementace dalších UI testů a nebo celkové dokončení aplikaci tak, aby ji bylo možné použít v produkčním nasazení. Výsledkem by tak mohla být obecně použitelná aplikace pro města, která by díky multiplatformním vlastnostem pokryla co největší množství zařízení a zároveň by byla snadněji a levněji udržovatelná než nativní aplikace.

Příloha A

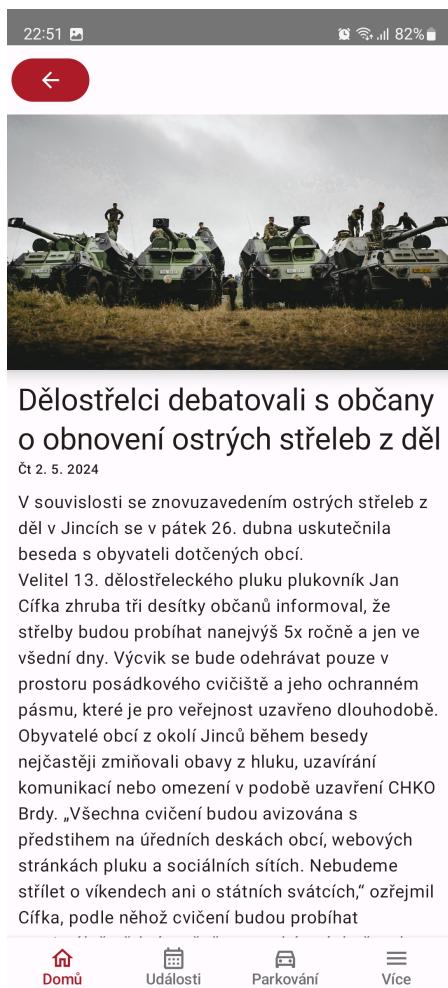
Snímky obrazovky



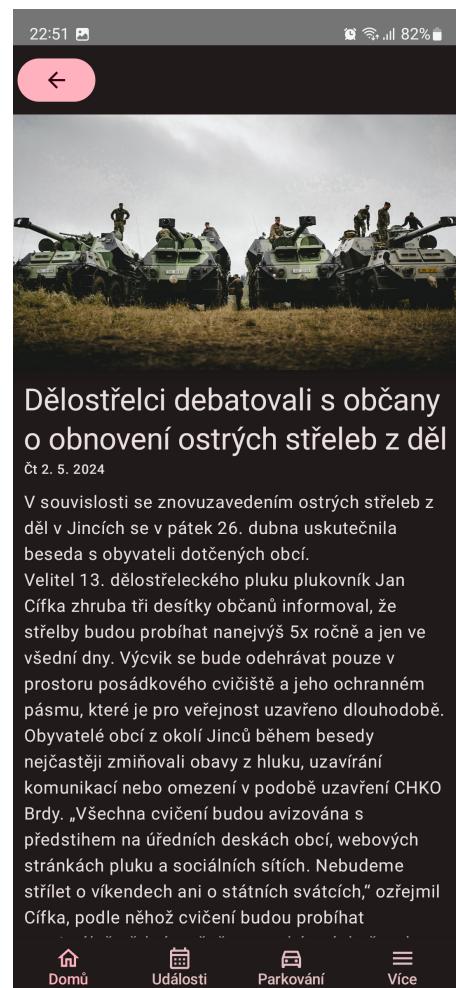
■ Obrázek A.1 Obrazovka Domů



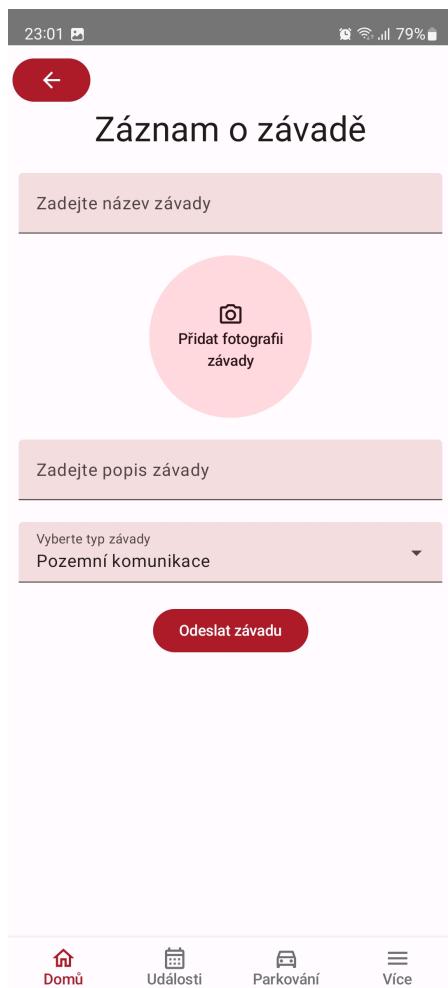
■ Obrázek A.2 Obrazovka Domů



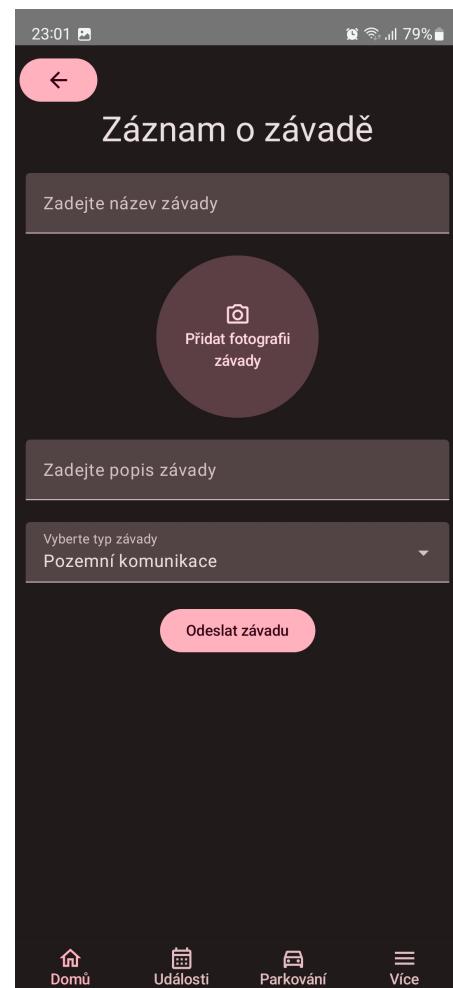
Obrázek A.3 Obrazovka Domů



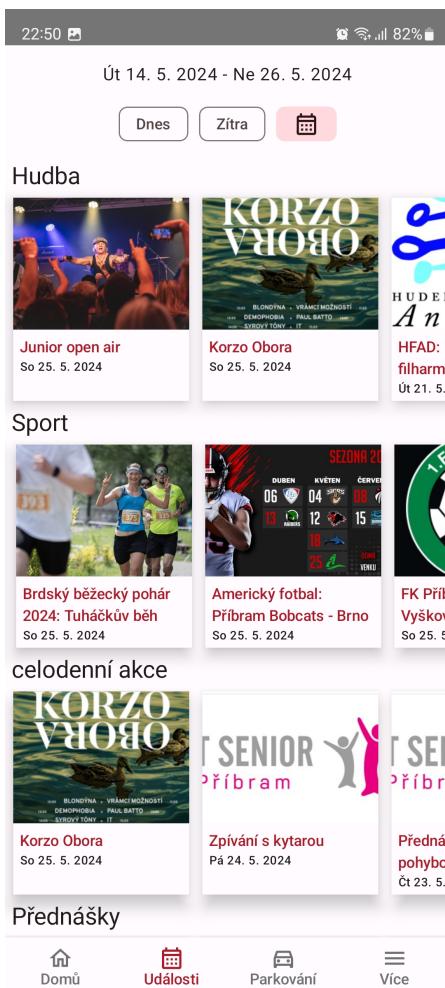
Obrázek A.4 Obrazovka Domů



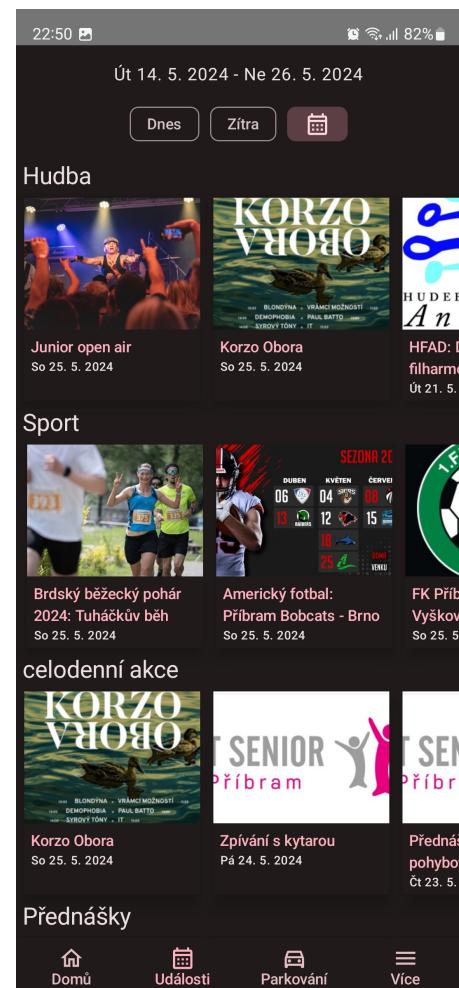
Obrázek A.5 Obrazovka *Domů*



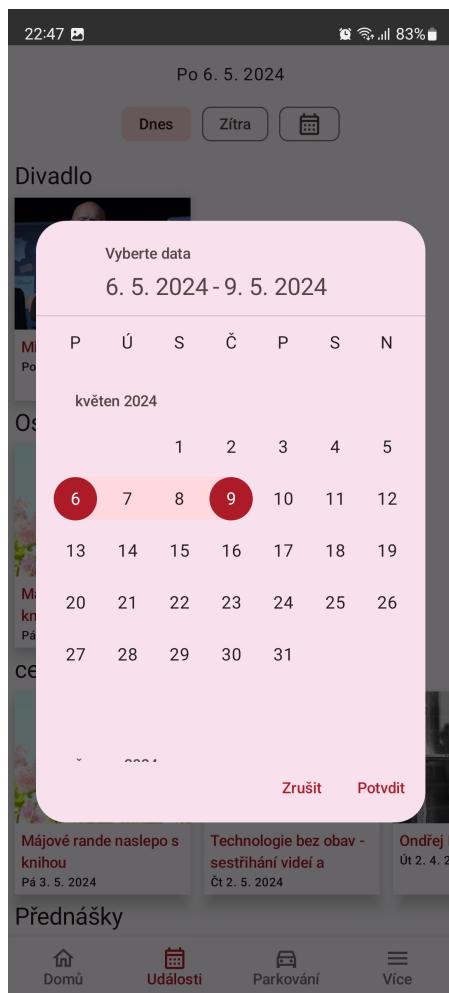
Obrázek A.6 Obrazovka *Domů*



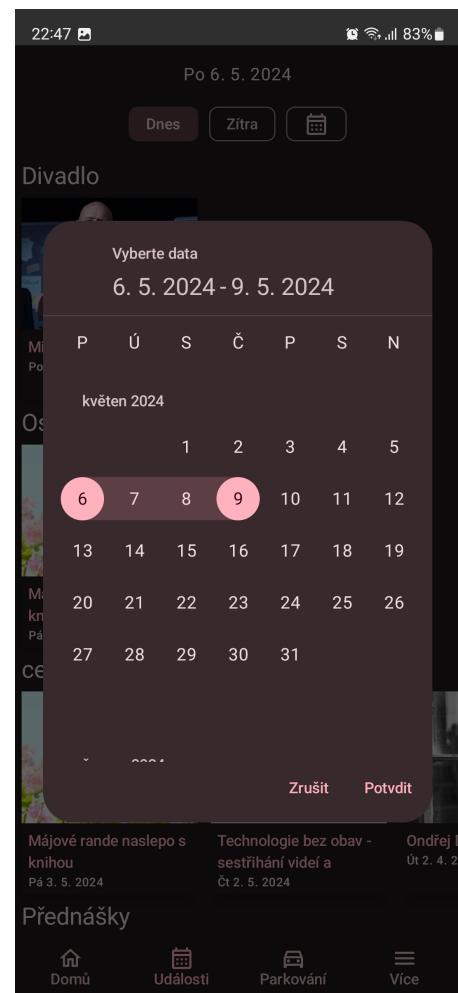
■ Obrázek A.7 Obrazovka *Události*



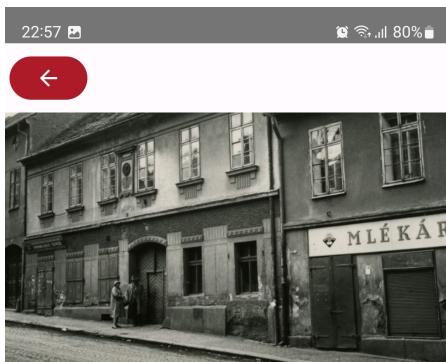
■ Obrázek A.8 Obrazovka *Události*



■ Obrázek A.9 Obrazovka *Události*



■ Obrázek A.10 Obrazovka *Události*



Vernisáž výstavy: Zkáza příbramského domu Jechů

Od: St 29. 5. 2024
Do: St 29. 5. 2024

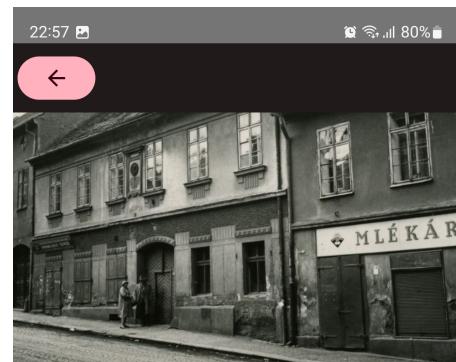
Výstava ilustruje život v Příbrami na přelomu 19. a 20. století na příkladu rodiny Jechů, obývající dům čp. 9 v Plzeňské ulici. Zároveň podává svědectví o zvůli moci a o křivdě, jež se stala lidem i městu v rámci asanace ulice v letech 1961–1962, a kterou mnozí Příbramáci dodnes mají v živé paměti.

Slavnostní zahájení se koná ve středu 29. 5. od 18 hodin.

<https://www.muzeum-pribram.cz/detail-akce/slavnostni-zahajeni-vystavy-zkaza-pribramskeho-domu-jechu/>



■ Obrázek A.11 Obrazovka *Události*



Vernisáž výstavy: Zkáza příbramského domu Jechů

Od: St 29. 5. 2024
Do: St 29. 5. 2024

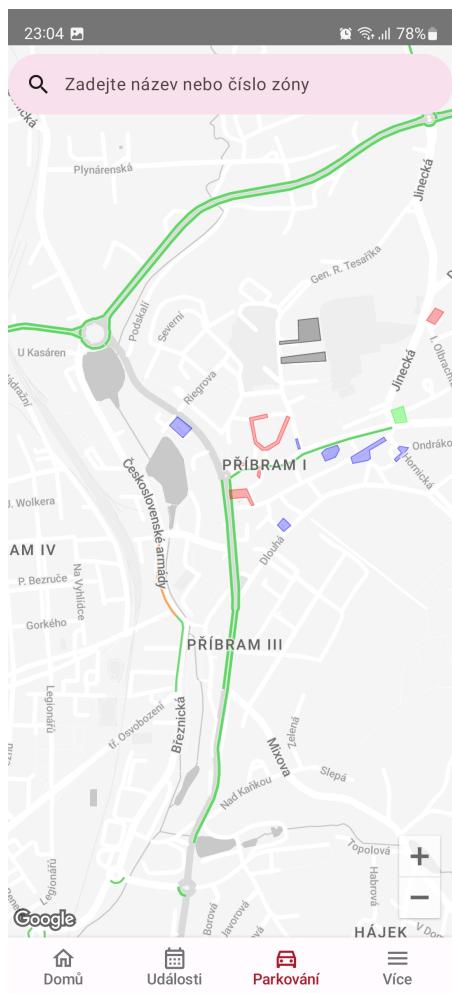
Výstava ilustruje život v Příbrami na přelomu 19. a 20. století na příkladu rodiny Jechů, obývající dům čp. 9 v Plzeňské ulici. Zároveň podává svědectví o zvůli moci a o křivdě, jež se stala lidem i městu v rámci asanace ulice v letech 1961–1962, a kterou mnozí Příbramáci dodnes mají v živé paměti.

Slavnostní zahájení se koná ve středu 29. 5. od 18 hodin.

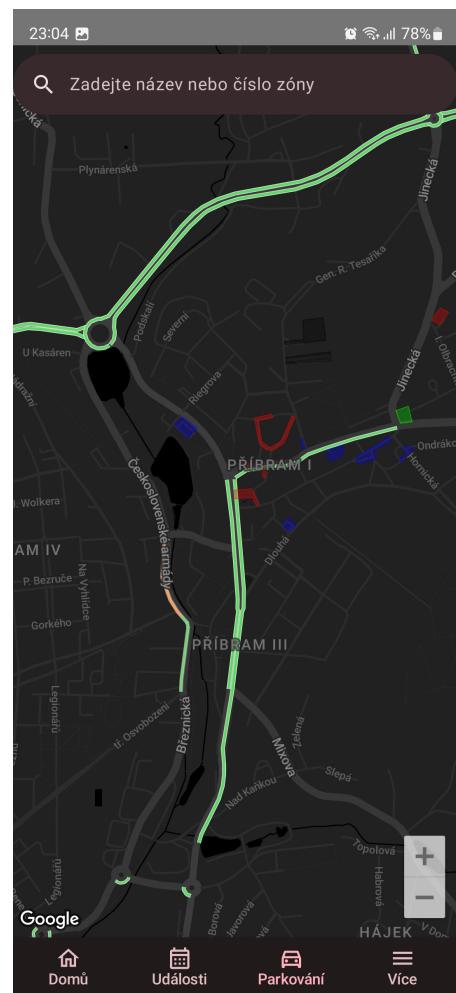
<https://www.muzeum-pribram.cz/detail-akce/slavnostni-zahajeni-vystavy-zkaza-pribramskeho-domu-jechu/>



■ Obrázek A.12 Obrazovka *Události*



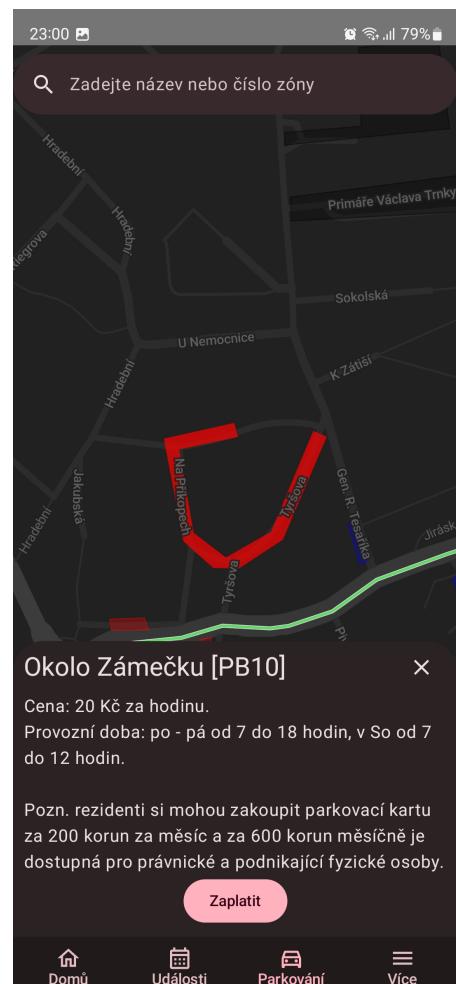
Obrázek A.13 Obrazovka Parkování



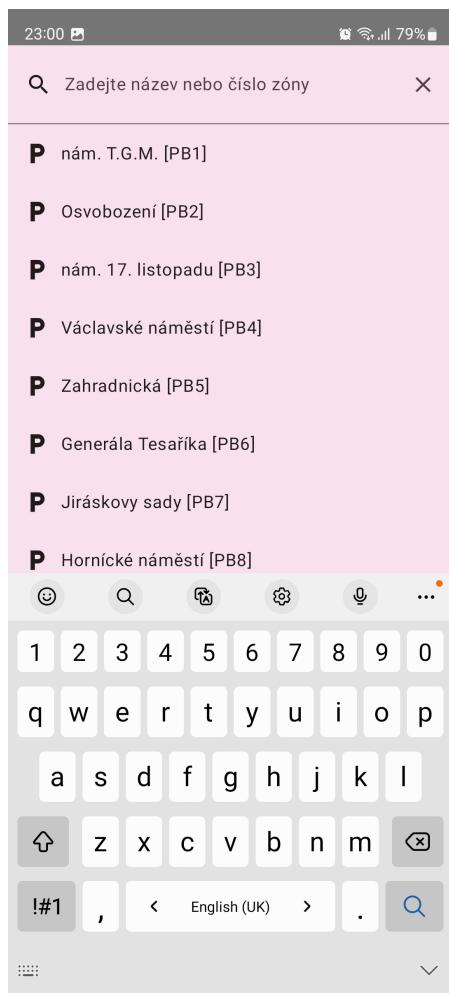
Obrázek A.14 Obrazovka Parkování



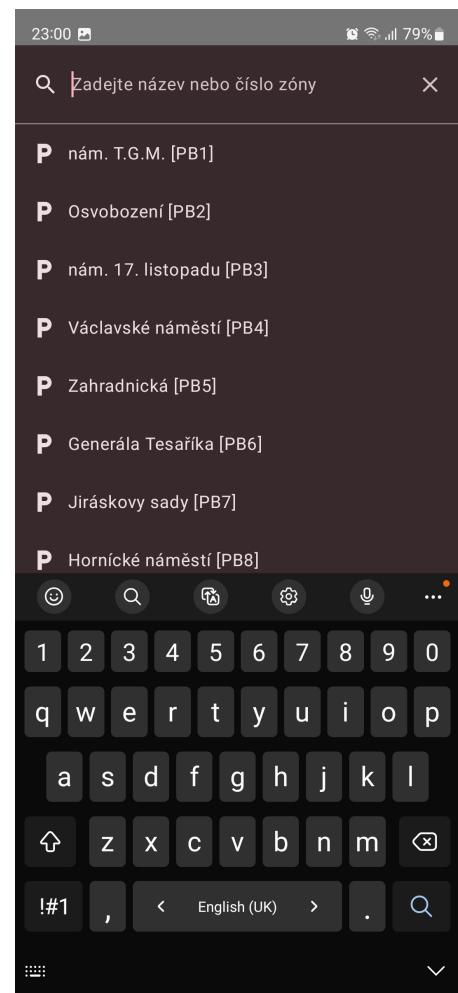
Obrázek A.15 Obrazovka Parkování



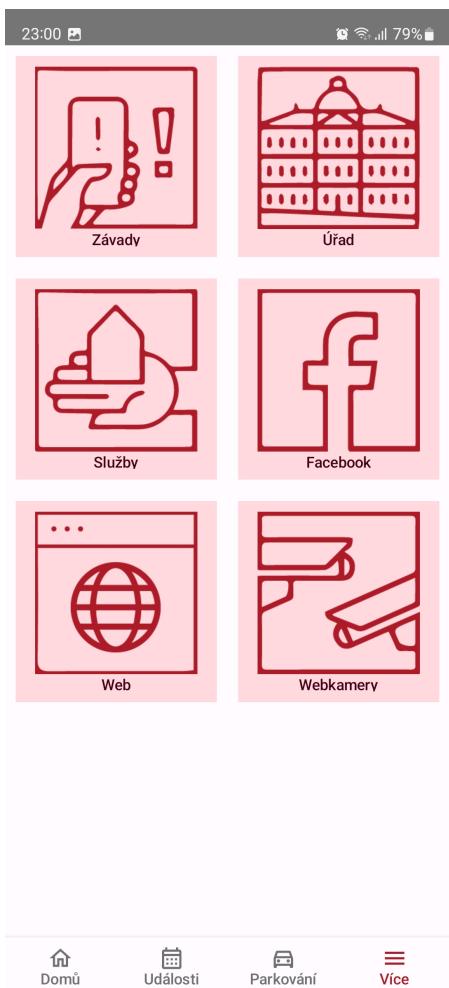
Obrázek A.16 Obrazovka Parkování



Obrázek A.17 Obrazovka Parkování



Obrázek A.18 Obrazovka Parkování



Obrázek A.19 Obrazovka *Více*



Obrázek A.20 Obrazovka *Více*

Bibliografie

1. STATISTA. *Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022*. 2022. Dostupné také z: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>. Accessed: 15.1.2024.
2. SHOUTEM. *A brief history of React Native*. 2016. Dostupné také z: <https://medium.com/react-native-development/a-brief-history-of-react-native-aae11f4ca39>. Accessed: 28.12.2023.
3. *Core Components and Native Components*. 2023. Dostupné také z: <https://reactnative.dev/docs/intro-react-native-components>. Accessed: 28.12.2023.
4. *React Fundamentals - JSX*. 2023. Dostupné také z: <https://reactnative.dev/docs/intro-react#jsx>. Accessed: 28.12.2023.
5. STEINBERGER, PETER. *The new shiny*. 2021. Dostupné také z: <https://increment.com/mobile/the-shift-to-declarative-ui/>. Accessed: 28.12.2023.
6. *Fast Refresh*. 2023. Dostupné také z: <https://reactnative.dev/docs/fast-refresh>. Accessed: 28.12.2023.
7. *React Fundamentals*. 2023. Dostupné také z: <https://reactnative.dev/docs/intro-react>. Accessed: 28.12.2023.
8. *React Fundamentals*. 2023. Dostupné také z: <https://reactnative.dev/community/overview>. Accessed: 28.12.2023.
9. *Expo*. 2023. Dostupné také z: <https://expo.dev/>. Accessed: 28.12.2023.
10. *About the New Architecture*. 2024. Dostupné také z: <https://reactnative.dev/docs/the-new-architecture/landing-page>. Accessed: 20.3.2024.
11. META PLATFORMS, Inc. *Cross Platform Implementation*. 2024. Dostupné také z: <https://reactnative.dev/architecture/xplat-implementation>. Accessed: 15.3.2024.
12. *Render, Commit, and Mount*. 2023. Dostupné také z: <https://reactnative.dev/architecture/render-pipeline>. Accessed: 28.12.2023.
13. TEAM, Flutter. *Flutter FAQ*. 2023. Dostupné také z: <https://docs.flutter.dev/resources/faq>. Accessed: 28.12.2023.
14. TEAM, Flutter. *Reactive user interfaces*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#reactive-user-interfaces>. Accessed: 28.2.2024.
15. TEAM, Flutter. *Widgets*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#widgets>. Accessed: 28.2.2024.

16. TEAM, Flutter. *Hot Reload*. 2024. Dostupné také z: <https://docs.flutter.dev/tools/hot-reload>. Accessed: 28.2.2024.
17. TEAM, Flutter. *Flutter architectural overview*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview>. Accessed: 28.12.2023.
18. TEAM, Flutter. *Flutter*. 2024. Dostupné také z: <https://flutter.dev>. Accessed: 28.2.2024.
19. TEAM, Flutter. *Widget catalog*. 2024. Dostupné také z: <https://docs.flutter.dev/ui/widgets>. Accessed: 28.2.2024.
20. Dart. 2024. Dostupné také z: <https://dart.dev/>. Accessed: 28.2.2024.
21. TEAM, Flutter. *Flutter's rendering model*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#flutters-rendering-model>. Accessed: 28.12.2023.
22. JETBRAINS. *Compose Multiplatform*. 2024. Dostupné také z: <https://www.jetbrains.com/lp/compose-multiplatform/>. Accessed: 10.1.2024.
23. *Build better apps faster with Jetpack Compose*. 2024. Dostupné také z: <https://developer.android.com/jetpack/compose>. Accessed: 13.1.2024.
24. *Kotlin Multiplatform use cases*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform.html#android-and-ios-applications>. Accessed: 1.2.2024.
25. JETBRAINS. *Kotlin Multiplatform*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform.html>. Accessed: 1.2.2024.
26. AIGNER, Sebastian. *Compose Multiplatform for iOS Is in Alpha*. 2024. Dostupné také z: <https://blog.jetbrains.com/kotlin/2023/05/compose-multiplatform-for-ios-is-in-alpha/>. Accessed: 15.3.2024.
27. *Jetpack Compose phases*. 2024. Dostupné také z: <https://developer.android.com/develop/ui/compose/phases>. Accessed: 13.1.2024.
28. INC., Alphabet. *Semantics*. 2024. Dostupné také z: <https://developer.android.com/develop/ui/compose/testing/semantics>. Accessed: 15.3.2024.
29. PETROVA, Ekaterina. *Kotlin Multiplatform Is Stable and Production-Ready*. 2023. Dostupné také z: <https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/>. Accessed: 1.2.2023.
30. *Expected and actual declarations*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-expect-actual.html>. Accessed: 15.3.2024.
31. *Common code*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-discover-project.html#common-code>. Accessed: 2.4.2024.
32. *Kotlin Native*. 2024. Dostupné také z: <https://kotlinlang.org/docs/native-overview.html>. Accessed: 2.4.2024.
33. *Hierarchical project structure*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-hierarchy.html>. Accessed: 2.4.2024.
34. JETBRAINS. *Advanced concepts of the multiplatform project structure*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-advanced-project-structure.html>. Accessed: 15.3.2024.
35. *Forbes Mobile App Shifts To Kotlin Multiplatform*. 2023. Dostupné také z: <https://www.forbes.com/sites/forbes-engineering/2023/11/13/forbes-mobile-app-shifts-to-kotlin-multiplatform/?sh=7ed3ba6d53ca>. Accessed: 1.2.2024.
36. *Mobile multiplatform development at McDonald's*. 2023. Dostupné také z: <https://medium.com/mcdonalds-technical-blog/mobile-multiplatform-development-at-mcdonalds-3b72c8d44ebc>. Accessed: 1.2.2024.

37. ANISIMOV, Alex. *Results of the First Kotlin Multiplatform Survey*. 2024. Dostupné také z: <https://blog.jetbrains.com/kotlin/2021/01/results-of-the-first-kotlin-multiplatform-survey/>. Accessed: 15.3.2024.
38. RAS, Jacob. *Android & iOS native vs. Flutter vs. Compose Multiplatform*. 2024. Dostupné také z: <https://www.jacobras.nl/2023/09/android-ios-native-flutter-compose-kmp/>. Accessed: 15.3.2024.
39. TEAM, Flutter. *How big is the Flutter engine?* 2024. Dostupné také z: <https://docs.flutter.dev/resources/faq#how-big-is-the-flutter-engine>. Accessed: 28.2.2024.
40. TEAM, Flutter. *Load sequence, performance, and memory*. 2024. Dostupné také z: <https://docs.flutter.dev/add-to-app/performance#memory-and-latency>. Accessed: 28.2.2024.
41. *Kotlin Multiplatform Development Roadmap for 2024*. 2023. Dostupné také z: <https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-development-roadmap-for-2024/>. Accessed: 1.2.2024.
42. STATISTA. *Market share of mobile operating systems worldwide from 2009 to 2023, by quarter*. 2023. Dostupné také z: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>. Accessed: 30.12.2024.
43. FIGMA. *What is a use case*. 2024. Dostupné také z: <https://www.figma.com/resource-library/what-is-a-use-case/>. Accessed: 15.3.2024.
44. SAP. *Separation of Concerns*. 2024. Dostupné také z: https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/abenseperation_concerns_guidl.htm. Accessed: 15.3.2024.
45. INC., Alphabet. *Guide to app architecture*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture>. Accessed: 15.3.2024.
46. INC., Alphabet. *State holders and UI State*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture/ui-layer/stateholders>. Accessed: 15.3.2024.
47. JETBRAINS. *ViewModel overview*. 2024. Dostupné také z: <https://developer.android.com/topic/libraries/architecture/viewmodel>. Accessed: 30.12.2024.
48. FLUTTER. *Start thinking declaratively*. 2024. Dostupné také z: <https://docs.flutter.dev/data-and-backend/state-mgmt/declarative>. Accessed: 15.3.2024.
49. INC., Alphabet. *UI events*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture/ui-layer/events>. Accessed: 15.3.2024.
50. GOOGLE. *Domain layer*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture/domain-layer>. Accessed: 30.12.2024.
51. INC., Alphabet. *Data layer*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture/data-layer>. Accessed: 15.3.2024.
52. ORACLE. *Data Access Object*. 2024. Dostupné také z: <https://www.oracle.com/java/technologies/data-access-object.html>. Accessed: 15.3.2024.
53. BAELDUNG. *The DTO Pattern (Data Transfer Object)*. 2024. Dostupné také z: <https://www.baeldung.com/java-dto-pattern>. Accessed: 15.3.2024.
54. BRANDI, Denis. *The “Real” Repository Pattern in Android*. 2024. Dostupné také z: <https://proandroiddev.com/the-real-repository-pattern-in-android-efba8662b754>. Accessed: 15.3.2024.
55. INC., Alphabet. *Color roles*. 2024. Dostupné také z: <https://m3.material.io/styles/color/roles>. Accessed: 15.3.2024.

56. INC., Alphabet. *Typography*. 2024. Dostupné také z: <https://m3.material.io/styles/typography/type-scale-tokens>. Accessed: 15.3.2024.
57. INC., Alphabet. *Add a Styled Map*. 2024. Dostupné také z: <https://developers.google.com/maps/documentation/android-sdk/styling>. Accessed: 15.3.2024.
58. JETBRAINS. *Navigation and routing*. 2023. Dostupné také z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html>. Accessed: 15.3.2024.
59. CAFÉ, Adriel. *Overview*. 2024. Dostupné také z: <https://voyager.adriel.cafe/>. Accessed: 15.3.2024.
60. CAFÉ, Adriel. *Tab navigation*. 2024. Dostupné také z: <https://voyager.adriel.cafe/navigation/tab-navigation>. Accessed: 15.3.2024.
61. CAFÉ, Adriel. *ScreenModel*. 2024. Dostupné také z: <https://voyager.adriel.cafe/screenmodel>. Accessed: 15.3.2024.
62. INC., Alphabet. *Handle ViewModel events*. 2024. Dostupné také z: <https://developer.android.com/topic/architecture/ui-layer/events#handle-viewmodel-events>. Accessed: 15.3.2024.
63. JETBRAINS. *What's new in Compose Multiplatform 1.6.0*. 2024. Dostupné také z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/whats-new-compose-1-6-0.html#across-platforms>. Accessed: 15.3.2024.
64. INC., Alphabet. *Testing your Compose layout*. 2024. Dostupné také z: <https://developer.android.com/develop/ui/compose/testing>. Accessed: 15.3.2024.
65. JETBRAINS. *@Preview annotation for Fleet*. 2024. Dostupné také z: <https://blog.jetbrains.com/kotlin/2024/02/compose-multiplatform-1-6-0-release/#preview-annotation-for-fleet>. Accessed: 15.3.2024.
66. JETBRAINS. *Writing and running tests with Compose Multiplatform*. 2024. Dostupné také z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-test.html#62f6e1cb>. Accessed: 30.12.2024.