



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
6. 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	1
1.2 Stručný přehled obsahu práce	1
1.3 Definice multiplatformního vývoje UI	1
1.4 Uplatnění multiplatformního vývoje a sdíleného UI	2
1.5 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	17
2.2.2 Náročnost implementace	18
2.3 Limitace Compose Multiplatform oproti nativnímu řešení	19
3 Návrh	20
3.1 Výběr aplikace	20
3.2 Případy užití	21
3.3 Funkční 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 Tvorba UI obecně	37
4.2.1 Navigace	37
4.2.2 Design systém	37
4.3 Použité technologie	41
4.3.1 Vrstva uživatelského rozhraní	41
4.3.2 Vrstva aplikační logiky	43

4.4	Logika uživatelského rozhraní	45
4.4.1	Stav uživatelského rozhraní	46
4.4.2	46
4.4.3	ViewModel	47
4.5	Tvorba UI obrazovek	47
4.5.1	Domovská obrazovka	48
4.5.2	Obrazovka <i>Události</i>	49
4.5.3	Obrazovka <i>Parkování</i>	49
4.6	Řešení problémů spojených s multiplatformním vývojem	51
5	Testování	52
5.1	Uživatelské rozhraní	52
5.2	Výkon aplikace	52
5.3	Přístupnost aplikace	52
5.4	Kompatibilita	52
5.5	Možnosti testování UI v Compose Multiplatform	52
5.6	Testovací případy	53
5.7	Implementace testů	56
5.8	Zhodnocení výsledků testování	56
5.9	Zhodnocení použitelnosti	56
6	Závěr	57
6.1	Evaluace vlastností Compose Multiplatform ve srovnání s cíli práce	57
6.2	Zkušenosti z implementace na reálné aplikaci	57
6.3	Závěr o použitelnosti frameworku	57
6.4	Shrnutí dosažených výsledků	58
6.5	Zhodnocení splnění cílů práce	59
6.6	Návrhy pro budoucí vývoj a výzkum	59
A	Snímky obrazovky	60
	Obsah příloh	69

Seznam obrázků

2.1	React Native	5
2.2	React Native vykreslovací fáze	6
2.3	Flutter architectural layers	8
2.4	Flutter build proces	9
2.5	Compose Multiplatform iOS	10
2.6	UI struktura a její sémantický strom	11
2.7	Možnosti implementace KMP	12
2.8	Hierarchická struktura KMP [30]	14
2.9	Strom závislostí	15
2.10	Množství kódu KMP vs native	16
2.11	KMP průzkum	16
2.12	APK/IPA size in megabytes	17
2.13	Časy nastartování jednotlivých aplikací	18
3.1	Use case diagram	23
3.2	Architektura UI vrstvy	26
3.3	Reprezentace vztahu UI a stavu aplikace	27
3.4	Architektura datové vrstvy	28
3.5	Provázanost jednotlivých částí datové vrstvy	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	Domácí obrazovka	60
A.2	Domácí obrazovka	60
A.3	Domácí obrazovka	61
A.4	Domácí obrazovka	61
A.5	Obrazovka událostí	62
A.6	Obrazovka událostí	62
A.7	Obrazovka událostí	63
A.8	Obrazovka událostí	63
A.9	Obrazovka parkování	64
A.10	Obrazovka parkování	64
A.11	Obrazovka parkování více	65
A.12	Obrazovka parkování více	65

Seznam tabulek

2.1	Porovnání rychlosti spuštění ukázkové aplikace na platformě Android	18
2.2	Porovnání rychlosti spuštění ukázkové aplikace na platformě iOS	18
3.1	Pokrytí případů užití funkčními požadavky aplikace	25

Seznam výpisů kódu

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	38
4.3	Lib integration	39
4.4	Version katalog	39
4.5	Zadefinování barev	40
4.6	Definice barevných motivů	40
4.7	Definice barevných motivů	41
4.8	Ukázka použití stylu písma	41
4.9	Ukázka použití navigace založené na záložkách	42
4.10	Coil	43
4.11	Konfigurace http klienta	43
4.12	Zaslání požadavku	44
4.13	DI databázového ovladače pomocí Koinu	44
4.14	SQL dotaz	45
4.15	SQL vygenerovaný dotaz	45
4.16	Nativní databázový ovladač	45
4.17	Event State katalog	46
4.18	Události uživatelského rozhraní	46
4.19	Použití stavu v aplikaci	47
4.20	Implementace ViewModelu	47
4.21	Popis UI widgetů pomocí jazyka Kotlin	48
4.22	Implementace posuvného řádku pomocí LazyRow	48
4.23	GoogleMap element	50
4.24	Motiv mapy ve formátu JSON	50
5.1	Integrace UI testů pomocí Gradle	53

5.2 Implementace UI 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 6. května 2024

Abstrakt

Fill in abstract of this thesis in Czech language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Klíčová slova enter, commma, separated, list, of, keywords, in, CZECH

Abstract

Fill in abstract of this thesis in English language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Keywords enter, commma, separated, list, of, keywords, in, ENGLISH

Seznam zkratek

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

Kapitola 1

Úvod

1.1 Cíle práce

Prvním důležitým cílem práce je analyzovat 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.

Další 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 Stručný přehled obsahu práce

V úvodu se práce zabývá otázkou 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 aplikovat multiplatformní UI a na jakých zařízeních.

Jaké jsou výhody a nevýhody multiplatformních aplikací v porovnaní s nativními aplikacemi.

Existujícími frameworky a jich porovnáním s Compose Multiplatform.

Detailně rozebrána architektura Compose Multiplatform jak framework funguje a jaké další technologie jsou potřeba k smysluplné implementaci tohoto frameworku.

Dále je se práce zabývá návrhem a následnou tvorbou multiplatformního UI pomocí frameworku Compose Multiplatform.

V jedné z posledních části se týká možností testování takto vytvořeného UI a následného testování UI na nainplementované aplikaci.

je věnován shrnutí celého procesu implementace a vyzdvižení naskytlých problémů a výhod oproti nativnímu případně jiným multiplatformním frameworkům

Obecně něco o multiplatformním vývoji.

1.3 Definice multiplatformního vývoje UI

Technologie sloužících 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 aut či klasické počítače 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 ne vždy má smysl multiplatformní přístup aplikovat a tomu kdy je vhodné aplikovat multiplatformní vývoj je věnována následující kapitola.

1.4 Uplatnění multiplatformního vývoje a sdíleného UI

Aktuálně největší uplatnění multiplatformního UI nabízí mobilní vývoj, kde je díky multiplatformním frameworkům možné vyvíjet aplikace pro platformy Android a iOS současně. Je tomu tak z důvodu, že mobilní vývoj je aktuálně jeden TODO také díky podobnostem majících na návrh multiplatformního UI vliv jako je například velikost obrazovky.

I přesto, že multiplatformní frameworky umožňují implementaci multiplatformním UI na většinu nejvíce používaných platforem jako je Android, iOS, Android TV, tvOS, Wear OS, watchOS nebo Android Automotive, tak né vždy je vhodné těchto možností využít. Z pohledu UI mají ty platformy často jiné velikosti obrazovek, jiné možnosti ovládaní a proto, je často 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 omezeních a požadavků výrazně lišit.

1.5 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. Jednodušší dosažení konzistentního vzhledu na různých 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. 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 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 věnuje porovnáním 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 nejstaršího po nejmladší, což umožňuje lepší náhled na to, jak se jednotlivé frameworky vyvíjely v čase. Pro lepší ilustraci rozdílů mezi jednotlivými frameworky byla zvolena podobná struktura jednotlivých kapitol, čehož bylo možné docílit díky mnoha společnými 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 hackaton projektu a první jeho oficiálně publikovaná verze vyšla začátkem roku 2015. [2] Nyní se jedná o open-source framework, kde jeho hlavním cílem je 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] Takto zapsané UI lépe reflektovalo aktuální stav aplikace. Tento způsob zápisu začal na mobilních platformách růst 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 v 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 Expo framework, který zjednodušuje proces vývoje a umožňuje rychlé prototypování. [9]

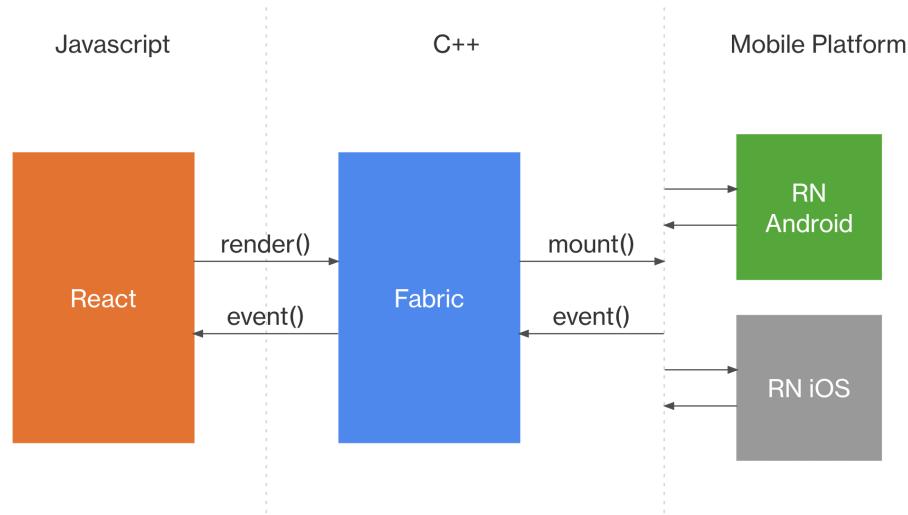
Architektura frameworku React Native

React native do roku 2022 využíval architekturu založenou na návrhového vzoru Bridge, který spojoval kód napsaný v JavaScriptu s nativní kódem určeným pro danou platformu. Tyto dva celky byly spuštěny na souběžných vláknech a komunikovaly spolu pomocí zaslání 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.

■ **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 />
```



■ **Obrázek 2.1** React Native

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.

Na ukázce kódu 2.1 jsou použity 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: [11]

Render

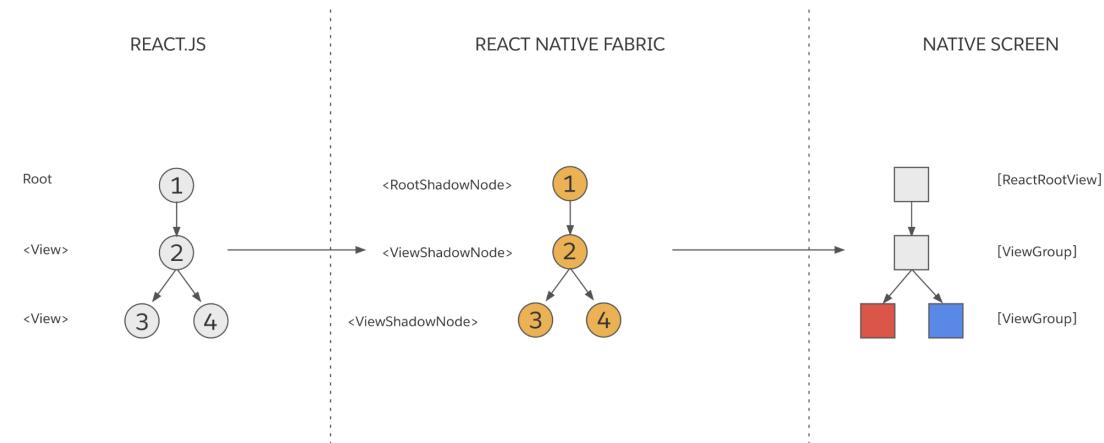
Během této fáze se z jednotlivých komponent (React Elementů) sestaví strom elementů v JavaScriptu (viz levá část obrázku 2.2) a nad tímto stromem se následně spustí rekurzivní redukce, při které dojde k vytvoření nového stromu takzvaného React Shadow Tree. (viz prostřední část obrázku 2.2) [11] Ten se skládá z jednotlivých React Shadow Nodes, které reprezentují objekty v C++ a tím přechází tato renderovací fáze do další fáze zvané commit.[11]

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. [11]

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. [11] Čili například na platformě Android se <*ViewShadowNode*> přetrasformuje na *android.view.ViewGroup*. [11]



■ Obrázek 2.2 React Native vykreslovací fáze

2.1.2 Flutter

Flutter je open-source softwarový toolkit pro vývoj uživatelských rozhraní (UI). [12] Za vývojem stojí společnost Google a je určený k vytváření nativně komplikovaných aplikací pro mobilní zařízení, web a desktop z jednoho zdrojového kódu. [12] 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. [13] Widgety jsou základními stavebními bloky Flutter aplikací, představující vše od strukturálních prvků po stylistické komponenty. [14]

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. [15] Teprve pro vydání jsou Flutter aplikace komplikovány přímo do strojového kódu, ať už jde o instrukce Intel x64 nebo ARM, případně do JavaScriptu pokud jsou cíleny na web. [16]

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. [17]

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. [18]

Programovací jazyk Dart

Aplikace vytvořené pomocí frameworku Flutter jsou psány v jazyce Dart, moderním objektově orientovaném programovacím jazyce vyvinutém společností Google. Dart je navržen pro optimální výkon a produktivitu, což ho činí vhodným pro mobilní a webový vývoj. [19]

Způsob renderovaní UI

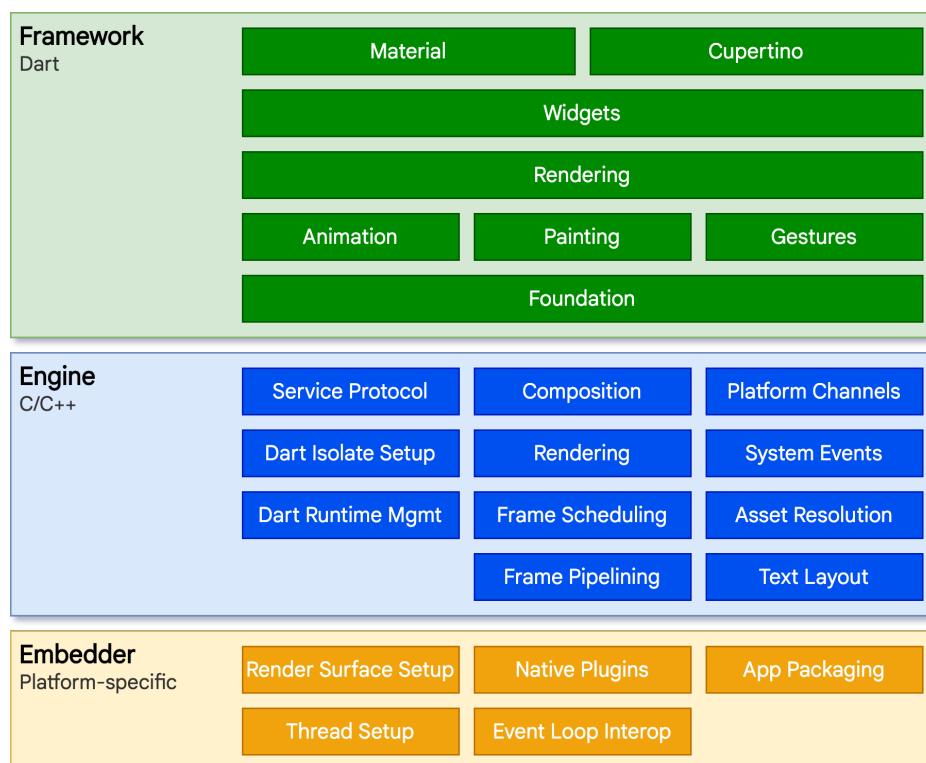
Na rozdíl od Reactu Native Flutter nepoužívá platformě specifické komponenty koncových zařízení, ale veškeré UI komponenty (widgety) renderuje pomocí vlastního renderovacího enginu. [20]

Architektura frameworku Flutter

Jak je vidět na obrázku 2.3, tak architektura Flutteru je rozdělena do tří hlavních vrstev. [16] 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 vrstva starající se o vykreslování grafiky, kdykoli kdy je potřeba vykreslit nový snímek. [16] 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. [16]

■ **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'),
    ],
  ),
);
```



■ **Obrázek 2.3** Flutter architectural layers

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

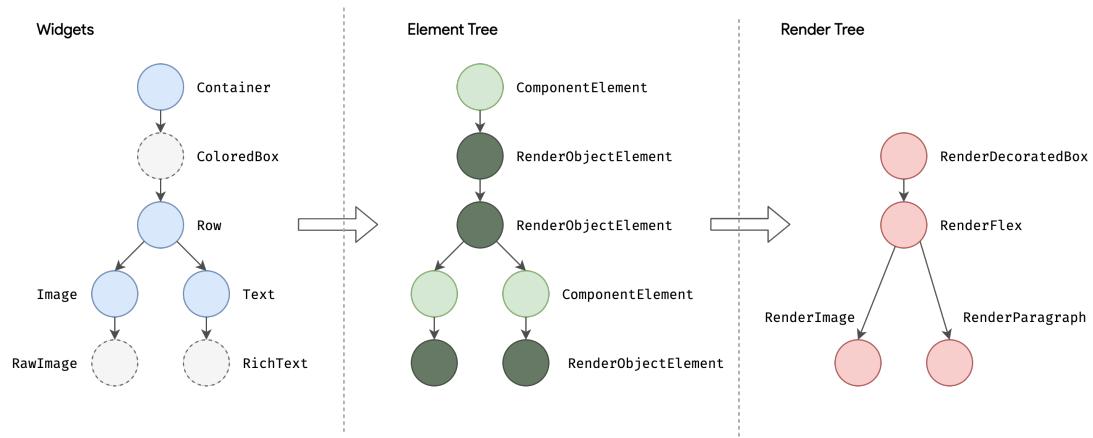
Z pohledu UI je důležitým prvkem právě Flutter framework, který zároveň definuje jak spolu jednotlivé widgety interagují.

Widget je ve Flutteru základní stavební blok pro tvorbu uživatelského rozhraní. [14] V následující ukázce kódu 2.2 je pro příklad použito několik základních widgetů jako je *Image* nebo *Text* a taktéž layout widget zvaný *Container* a *Row* pro organizaci a rozložení vnořených widgetů na obrazovce.

Když Flutter potřebuje vykreslit tento blok, zavolá metodu *build()* a ta vrátí podstrom wi-

dgetů, které následně vykreslí uživatelské rozhraní na základě aktuálního stavu aplikace. [16]

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 jeden element a každý prvek představuje určitou instanci widgetu v daném umístění stromové hierarchie. [16]



■ Obrázek 2.4 Flutter build proces

2.1.3 Compose Multiplatform

Compose Multiplatform je framework sloužící k tvorbě uživatelských rozhraní použitelných na vícero platformách za jehož vývojem stojí společnost JetBrains. [21] Je založen na toolkitu zvaném Jetpack Compose, který je aktuálně doporučovaný k tvorbě nativních uživatelských rozhraních na platformě Android. [22]

Podporuje platformy jako Android, iOS (Alpha), Windows, MacOS, Linux a Web (experimentální). [21]

Klíčové vlastnosti Compose Multiplatform

Deklarativní zápis UI

Používá deklarativní syntaxi pro popis uživatelského rozhraní. [23]

Jednotný kód pro různé platformy

Umožňuje sdílet kód pro Android, iOS, web i desktop. [21]

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. [24]

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 jsou psány v jazyce Kotlin pro bezproblémovou integraci se serverovou částí.

Podpora od JetBrains

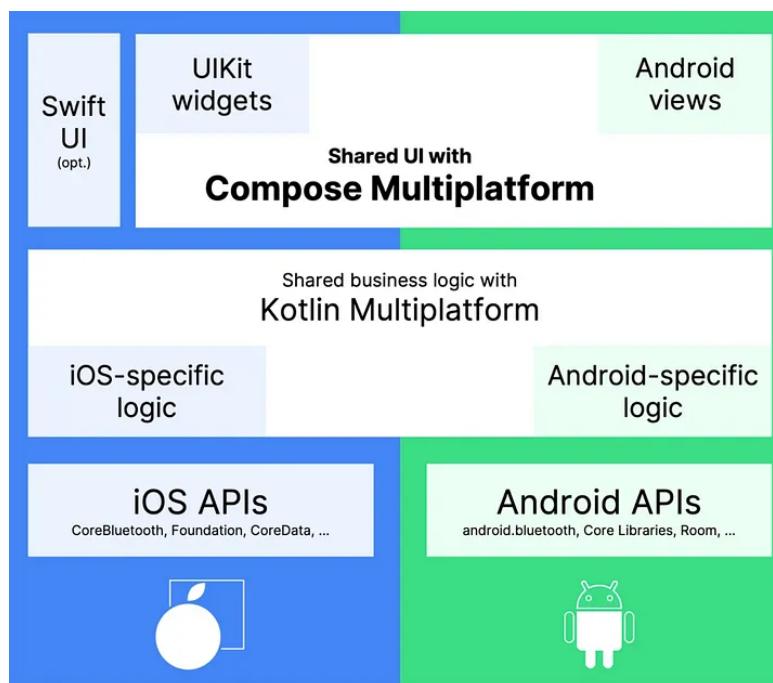
Poskytuje stabilní podporu od vývojářského týmu JetBrains.

Architektura frameworku Compose Multiplatform

Jelikož je framework Compose Multiplatform z velké části založen na frameworku Jetpack Compose, tak v této kapitole bude probírána právě architektura toho frameworku, která bude doplněna o drobné rozdíly mezi těmito dvěma frameworky.

V porovnání s ostatními dříve zmíněnými frameworky slouží tento framework pouze k tvorbě multiplatformního UI a z toto důvodu je převážně používán s toolkitem Kotlin Multiplatform, který dané aplikaci poskytuje i multiplatformní aplikační logiku. Jelikož se jedná o základní komponentu bez které by Compose Multiplatform nevznikl, tak 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 Compsose Multiplatfom s toolkitem Kotlin Multiplatfom (KMP). Dále vizualizuje vztah platformě specifických knihoven pro tvorbu UI jako 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, od kterých je postupně upouštěno a přechází se k deklarativním frameworkům jako je Jetpack Compose nebo SwiftUI.



■ Obrázek 2.5 Compose Multiplatform iOS

TODO

Co se týče samotného UI kódu, tak ten se skládá z jednotlivým 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.

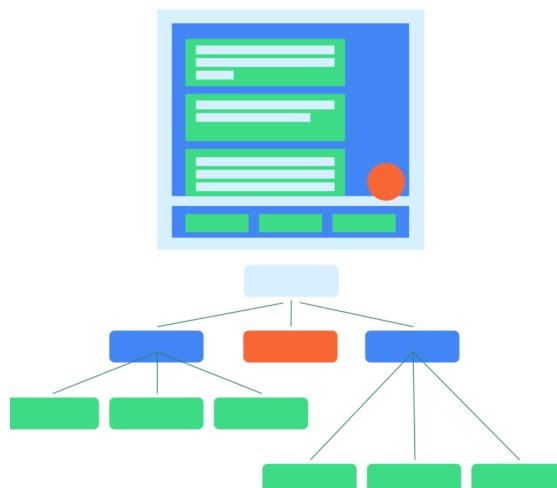
Takto strukturovaný kód může být následně frameworkem vykreslen na obrazovku k čemuž dochází v těchto třech následujících fázích:

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

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

Composition

Během této fáze Jetpack Compose vytvoří stromovou strukturu reprezentující UI komponenty, které mají být vykresleny.[25] Tato stromová struktura je tvořena na základě do sebe zanořených Composable funkcí, které představují jednotlivé prvku uživatelského rozhraní. Lepé tutu skutečnost reprezentuje obrázek 2.6, na kterém je vidět, že nejspodnější vrstvu UI, reprezentuje kořenový uzel sémantického stromu a následující vnořené prvky jsou jeho potomky.



■ **Obrázek 2.6** UI struktura a její sémantický strom

Layout

Během této fáze se jednotlivým 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 rozhodně o vlastní velikosti. Nakonec umístí každého potomka relativně ke své pozici.[25]

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. [25]

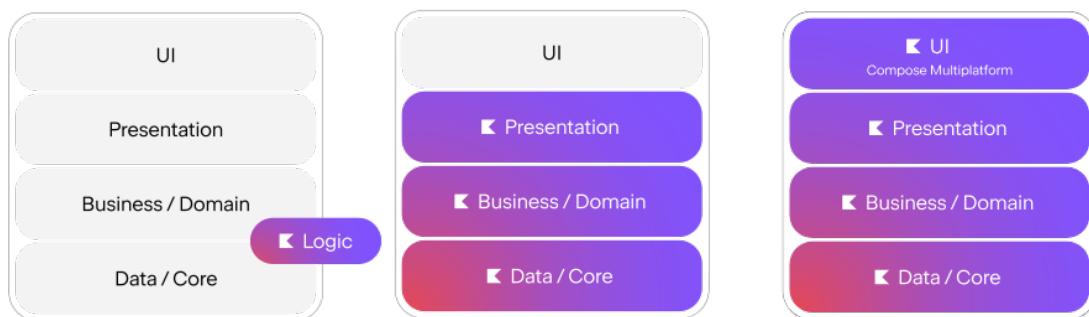
Drawing

V rámci poslední fáze dochází s samotnému vykreslení komponent na obrazovku koncového zařízení. [25] Compose během této fáze znova postupně prochází strom UI elementů od kořene až do listů a každý element tohoto stromu vykreslí sám sebe na pozici určenou během Layout fáze. [25]

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. [23] Důležitým rozdílem oproti ostatním multiplatformním frameworkům je právě ta možnost, kterou KMP případně Compose Multiplatform vývojářům poskytuje.

Jelikož se jedná o SDK, tak umožňuje vývojářům implementovat multiplatformní funkcionality postupně, bez nutnosti implementovat v Kotlinu celé vrstvy aplikací. Díky tomu dává vývojářům možnost sdílet napříč platformami ty části kódu, které mají největší smysl implementovat pro veškeré platformy a zbylé části kódu psát v nativním jazyce pro danou platformu. [24] 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

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 víro platforem a postupem času přesouvat do multiplatformní části aplikace například celé vrstvy starající se o prezentační, datovou nebo jinou aplikační logiku. [23]

Aby ale k takové migraci mohlo dojít, tak je zapotřebí aby pro technologie používané v původní nativní aplikaci existovali 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. [26]

Právě počet a vyspělost téhoto knihoven se podle společnosti, které již KMP používají jeví jako zatím jedna z nejslabších stránek KMP (viz kapitola 2.1.3.1).

V případě, že žádná vhodná knihovna není k dispozici, tak je zde stále možnost čehož je možné docílit díky funkcionalitě jazyka Kotlin zvané Expected a actual deklarace.

Expected a actual deklarace

Deklarace *expected* a *actual* umožňují přistupovat k platformě specifickým API z Kotlin Multiplatform modulů. [27] 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 v rámci společného modulu označit klíčovým slovem `expect`. [27] 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`. [27]

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. [27]

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 je 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`. [28]

Kotlin komplilátor na základě zdrojových kódů umístěných v tomto adresáři generuje sadu binárních souborů specifických pro danou platformu. [28] Při komplikaci tak může z téhož kódu vygenerovat několik souborů, jako například soubory pro *Java Virtual Machine (JVM)*, anebo spustitelné soubory pro nativní platformu. [28]

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. [29]

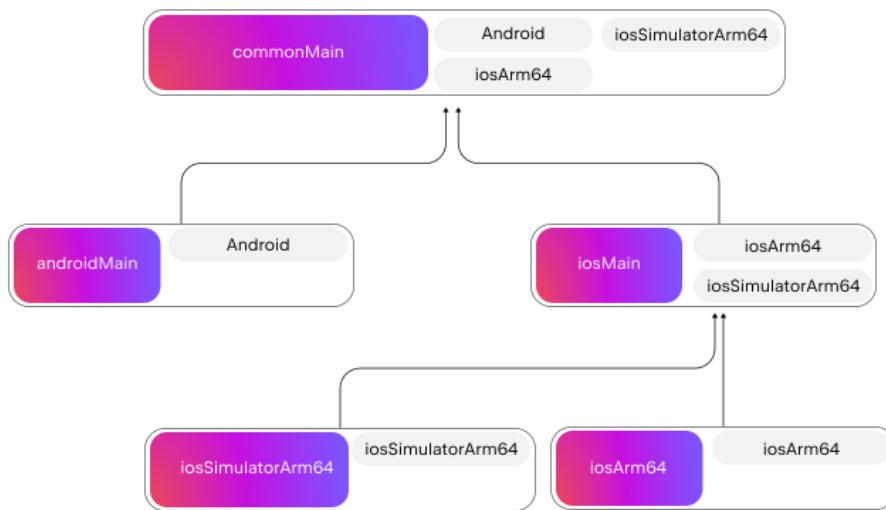
Pro ukázku toho, jaká zařízení jsou technologií Kotlin/Native podporovaná slouží následující obrázek 2.8



■ **Obrázek 2.8** Hierarchická struktura KMP [30]

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 i těchto platformě specifických kódů mohou být součástí společného kódu. Nesdílení toho kódu s ostatními platformami by tak zapříčinilo duplikaci velké části kódu.

Aby se tomuto zabránilo je vhodné využít 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

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

A dále na adresáře pro vybrané platformy jako například `AndroidMain`, `iosMain`, které již mohou být závislé na pouze platformě specifický knihovnách.

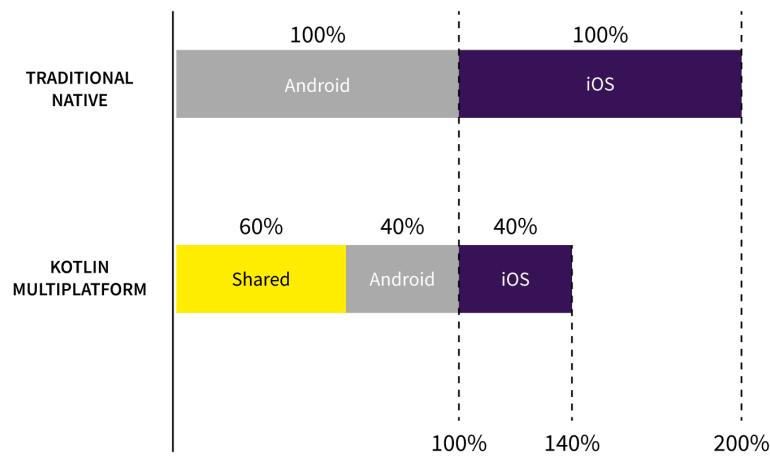
Aktuální použití KMP v praxi

I přesto, že je KMP ve stabilní verzi teprve od listopadu 2023 [26], tak tuto technologii používá již několik světově známých firem v produkčním nasazení jako je například Forbes či McDonald's. Forbes uvádí, že právě díky KMP dokáží sdílet až 80 % aplikační logiky napříč platformami Android a iOS. [31] Druhý ze jmenovaných McDonald's jako hlavní výhody uvádí jednodušší testování [32]

Naopak mezi největší výzvy, 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). [31] [32]

Pro porovnání s

"Od této doby jsme využívají a v produkci provozujeme několik mobilních aplikací. Ze zkušenosti vidíme, že při jejich vývoji dokážeme sdílet cca 60–70 % kódu na platformu. V případě vývoje na dvě platformy to znamená, že v součtu dokážeme ušetřit minimálně třetinu nákladů na vývoj, plus s tím související další náklady na věci typu testování (v tomto případě snížení až na polovinu)."



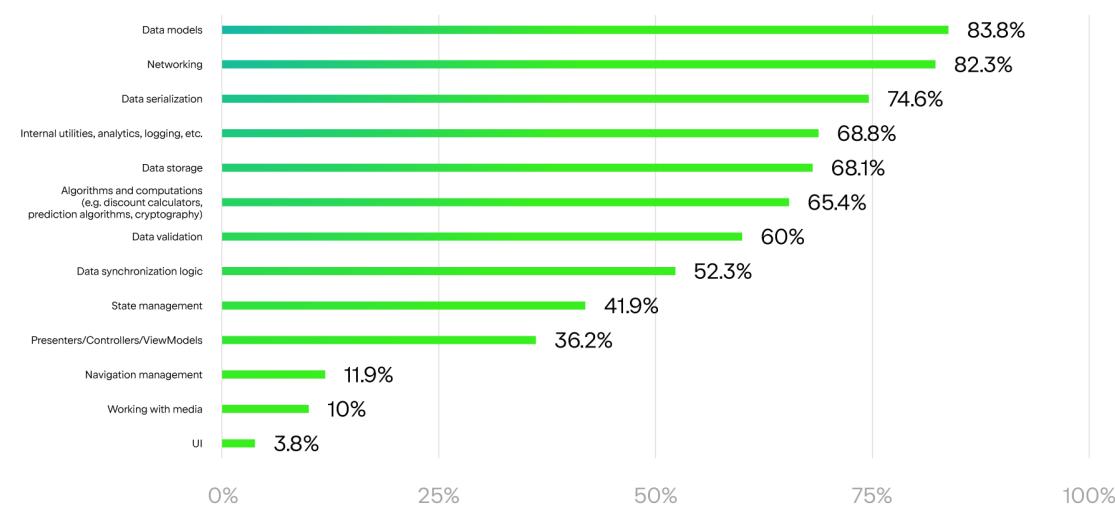
Obrázek 2.10 Množství kódu KMP vs native

Pro upřesnění je ještě potřeba zmínit, že veškeré vyjmenované společnosti, využili 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 2.11, 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 2.11 KMP průzkum

2.2 Porovnání vybraných multiplatformních frameworků

V rámci této kapitoly budou porovnány vybrané multiplatformní frameworky

2.2.1 Porovnání výkonu

Mezi klíčové parametry, kvůli kterým většina firem upřednostňuje vývoj nativních aplikací, patří především výkon nativních aplikací. 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, která obsahovala jednu obrazovku, načetla obrázky z veřejného API a zobrazila je v horizontálním seznamu. Na každý obrázek šlo kliknout a zobrazit jej přiblžený pod seznamem.

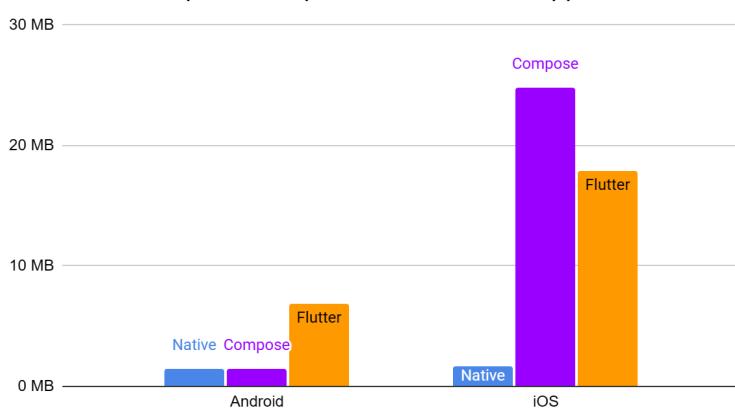
Veškeré testy proběhly na zařízeních Pixel 4a a iPhone 12 Mini a byly opakovány celkem pětkrát.

Velikost aplikace

Na obrázku 2.12 je vidět, že velikost aplikace založené na Compose Multiplatform je identická s velikostí nativní aplikace pro Android. Je tomu tak z toho důvodu, že výsledná aplikace pro Android neobsahuje kód pro jiné platformy. [<empty citation>] Kdežto u velikosti iOS aplikace je situace výrazně jiná. 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 ne a proto tam musí být spolu s aplikací dodána. [<empty citation>]

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 menší než Compose Multiplatform aplikace na platformě iOS. Součástí aplikace Flutter ještě vlastní engine, který je součástí aplikace a zvětšuje tak velikost aplikace asi o 3–4 MB pro Android a 10 MB pro iOS. [33]

Native vs. Compose Multiplatform vs. Flutter: App Size



■ Obrázek 2.12 APK/IPA size in megabytes

Rychlosť spuštēní aplikace

Při porovnání rychlosti spuštēní Compose Multiplatform aplikace na platformě Android není mezi rychlostmi téměř žádný rozdíl stejně tak jako tomu při porovnání velikosti aplikací v předchozí kapitole. O něco delší dobu spuštēní měla aplikace napsaná pomocí frameworku Flutter, která se spouštěla v průměru o 221 ms déle než Compose Multiplatform aplikace. Toto zpomalení je s nejvyšší pravděpodobností způsobeno dobou spuštēní Flutter Enginu, což by korespondovalo s oficiální Flutter dokumentací. [34]

U posledního porovnání na platformě iOS se doba spuštēní aplikací založených na Compose Multiplatform a frameworku Flutter o tolik nelišila od nativní aplikace.

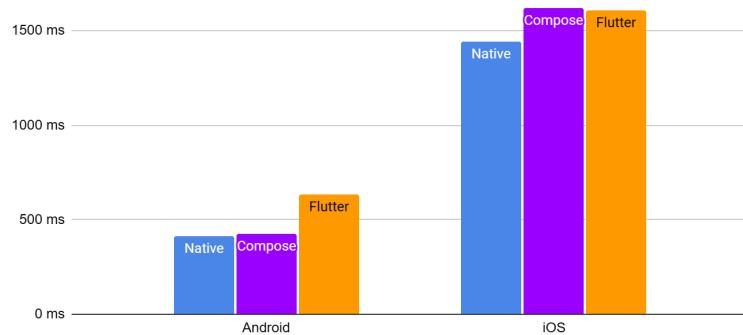
Tabulka 2.1 Porovnání rychlosti spuštēní ukázkové aplikace na platformě Android

	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 spuštēní ukázkové aplikace na platformě iOS

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

Native vs. Compose Multiplatform vs. Flutter: Startup time



Obrázek 2.13 Časy nastartování jednotlivých aplikací

2.2.2 Náročnost implementace

I přesto, že je náročnost implementace do jisté míry subjektivní záležitostí, tak existuje několik aspektů, které mohou k porovnání náročnosti implementace Compose Multiplatform s ostatními frameworky posloužit.

TODO

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 a nebo použití nativně vypadajících 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, tak aktuálně se nejvíce řeší problém s rychlostí vykreslování.

Většina výše zmíněných problémů se již řeší a je zmíněna v Kotlin Multiplatform vývojářském plánu pro rok 2024. [35] Konkrétně se jedná o veškeré zmíněné o problémy kromě zařazení HarmonyOS mezi podporované platformy. V tomto plánu je mimo jiné uvedeno, že aktuální prioritu číslo jedna je dostání frameworku Compose Multiplatform na platformě iOS do verze Beta. [35]

Kapitola 3

Návrh

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

Po vybrání typu aplikace následuje podkapitola věnující návrhu vhodných případů užití, které je vhodné specifikovat jednak aby bylo zřejmé jaké uživatelé budou danou aplikaci používat, tak z pohledu na jakých zařízení případně platformách ji budou využívat. Zároveň vhodné skloubení případů užití s grafickou podobou UI, tak aby nejčastější uživateli cílem bylo možné splnit co nejmenší dávkou úsilí, je pro správný návrh UI klíčové. Což obecně vede k intenzivnějšímu používání a oblibě aplikace.

Další fáze návrhu je věnována vytyčení funkčních požadavků ...

Dále je v rámci této kapitoly navržena architektura aplikace, která je pro následné navržení a správné fungování UI nezbytná.

Podkapitola *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 samotné návrhu uživatelského rozhraní aplikace a to nejprve za použití takzvaných wireframe modelů, následně návrhu designu 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 patřilo zaměření se na kritické problémy, které aktuálně multiplatformní vývojem provází. Mezi takové problémy patří například využití funkcí, které jsou silně spjaty s hardwarem koncových zařízení jako je 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 by jim například umožňovala vyhledat a zaplatit parkovné. TODO rozšířit uvodní popis aplikace pro výběr analyzu co by takova apk měla obsahovat

Na základě toho popisu byli vybrány následující případy užití.

3.2 Případy užití

Případy užití (use cases) jsou scénáře popisující, jakým způsobem by případní uživatelé aplikace mohli využívat určité funkce nebo vlastnosti aplikace k dosažení svých cílů. [<empty citation>] 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, co uživatel chce dosáhnout.
- Aktor: Uživatel nebo systém, který spouští případ užití.
- Popis: Podrobný popis scénáře, který obsahuje kroky, které uživatel vykonává, a odpovědi systému na tyto kroky.
- Předpoklady: Podmínky nebo situace, které musí být splněny, aby mohl být případ užití spuštěn.
- Výsledek: Očekávaný výsledek akce provedené uživatelem.

UC1 Zobrazení aktuálních novinek

Tento případ užití popisuje, jak mohou občané prostřednictvím aplikace získat přístup k aktuálním novinkám a důležitým oznámením ze svého města nebo obce. Tato funkce umožní občanům zůstat informováni o dění ve svém okolí.

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

UC2 Zobrazení aktuálních událostí

Tento případ užití popisuje jak mohou občané prostřednictvím aplikace získat přehled o aktuálně připravovaných akcích, událostech a kulturních aktivitách v jejich městě nebo obci.

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

UC6 Filtrování zobrazených událostí za základě času

Tento případ užití popisuje jak mohou občané filtrovat zobrazené události podle na základě datu konání daných událostí.

1. Systém uživateli zobrazí seznam událostí z webových stránek města, divadel, kin a dalších informačních zdrojů s nejbližším datem konání, který je rozdělen na základě kategorií jako například kino, divadlo, hudba atd..
2. Uživatel si vybere konkrétní den pro který chce zobrazit konané události.
3. Systém uživateli vrátí výpis pro vybraný den a události rozdělí na základě kategorií, do kterých vyfiltrované události spadají.

UC3 Zobrazení kontakty na místní úřady

Aplikace může poskytovat možnost rychlého a snadného získání kontaktu na místní úřady nebo správními orgány.

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

UC4 Nahlášení závady ve městě

Aplikace dá občanům možnost hlásit závady, které je potřeba ve městě opravit.

1. Systém uživateli zobrazí formulář pro 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
4. TODO rozdvojka spatne dobre vyplnene

UC5 Propojení s YouTube kanálem města

Aplikace dá občanům možnost konzumovat aktuální novinky prostřednictvím videí z městského YouTube kanálu. Díky intuitívnímu propojení...

- 1.
- 2.

UC7 Vyhledání parkovacích zón

Aplikace umožňuje občanům města jednoduše a rychle vyhledat veškeré parkovací zóny ve městě včetně informace o provozní době parkovací zóny.

1. Systém uživateli zobrazí mapu parkovacích zón ve jeho obci.
2. Uživatel vybere parkovací zóny pro kterou chce zobrazit detailní informace.
3. Systém vrátí uživateli detailní informace týkající se vybrané parkovací zóny.

UC8 Platba parkovného

podTask UC7 Aplikace umožní uživatelům snadno a rychle zaplatit parkovné prostřednictvím svého mobilního zařízení. Uživatelé mohou zadat informace o svém vozidle, délce parkování a provedení platby online, což jim ušetří čas a nepohodlí spojené s hledáním parkovacího automatu nebo mincí.

1. Systém uživateli zobrazí mapu parkovacích zón ve jeho obci.
2. Uživatel vybere parkovací zóny pro kterou chce zaplatit parkování.
3. Systém vrátí uživateli detailní informace týkající se vybrané parkovací zóny s odkazem na zaplacení.
4. Uživatel na detailu vybrané zóny vybere akci zaplatit parkovné.
5. Systém uživatele přesměruje na platební systém používaný daným městem pro platbu parkovného s předvyplněnou informací o vybrané parkovací zóně.

UC7 Zobrazení dopravy na mapě města

Tento případ užití popisuje jak mohou občané města jednoduše a rychle zobrazit vytíženost silnice ve městě.

1. Systém uživateli zobrazí mapu vytíženost silnice ve městě.

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

Co se týče vzájemných vazeb, tak zde je věno

Use case diagram



■ Obrázek 3.1 Use case diagram

Aktéři

Hlavními uživateli systému jsou především občané příslušného města, kteří budou aplikaci využívat za účelem splnění cílů z předchozí podkapitoly.

Dalším aktérem toto systému je informační systém (IS) města, který aplikaci poskytuje potřebná data.

TODO jaci uživatel budou aplikaci používat

Případy užití jsou klíčovým nástrojem pro pochopení potřeb uživatelů a návrhu funkcionality aplikace tak, aby co nejlépe odpovídaly těmto potřebám. Jsou také důležitým zdrojem pro testování a validaci funkčnosti aplikace, protože je na jejich základě možné ověřit, zda systém správně reaguje na očekávané uživatelské interakce.

Na jejichž základě je proto založeno koncové testování celé aplikace čehož je využito k tvorbě UI testů v kapitole *Testování 5*.

Na základě takto sepsaných případech užití je nyní možné sepsat funkční požadavky.

3.3 Funkční požadavky

Funkční požadavky definují konkrétní vlastnosti a chování aplikace, které zajistí, že bude aplikace pro občany měst či obcí užitečná, efektivní a snadno použitelná.

TODO priorita 5, slozitost viz grofetom

Zobrazení novinek

Možnost přidávat, editovat a mazat novinky a oznámení. Zobrazení aktuálních novinek na domovské obrazovce aplikace. Možnost filtrovat novinky podle kategorie nebo data.

Zobrazení událostí

Zobrazení kalendáře s událostmi, akcemi a důležitými daty. Možnost procházet události v kalendáři dle data, místa nebo typu události. Možnost přidávat nové události do kalendáře. Možnost filtrovat události podle kategorie nebo data.

Platební brána pro parkovné

Možnost provádět platby parkovného prostřednictvím aplikace. Zadání informací o vozidle, parkovacím místě a délce parkování. Zobrazení historie plateb a možnost platit opakováná parkování.

Možnost vyhledání a kontaktování místních úřadů, policie nebo zdravotního střediska

Zobrazení kontaktních informací a pracovní doby úřadů. Možnost zaslat otázku, stížnost nebo žádost přímo prostřednictvím aplikace.

Navigace a mapy

Integrace mapových služeb pro zobrazení umístění událostí a místních služeb. Možnost vyhledávání tras a navigace v rámci města nebo obce.

Podpora více jazyků

Možnost nastavení jazyka aplikace dle uživatelských preferencí. Zobrazení obsahu a textů v různých jazycích.

Feedback a hodnocení

Možnost zaslat zpětnou vazbu, hodnocení nebo recenzi na aplikaci. Zpětná vazba od uživatelů a možnost sledovat spokojenosť s aplikací.

Pokrytí případů užití

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

	F1	F2	F3	F4	F5	F6	F7	F8
UC1	*							
UC2	*							
UC3	*							
UC4		*						
UC5				*				
UC6			*					
UC7				*	*	*	*	
UC8					*	*	*	
UC9							*	
UC10							*	
UC11							*	

■ **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. [36]

TODO možné architektury pro vyvoj mobilních aplikací

Důležité principy

Separation of concerns

Separation of concerns je princip návrhu softwaru, který popisuje důležitost oddělení různých funkcionalit do samostatných, dobrě definovaných a izolovaných částí. 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. 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. 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 údržnosti softwarových aplikací.

Single source of truth

Princip Single Source of Truth (SSOT) je koncept v softwarovém inženýrství, který zdůrazňuje, že v aplikaci by měl existovat jeden zdroj dat, který obsahuje aktuální a spolehlivé informace. 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.

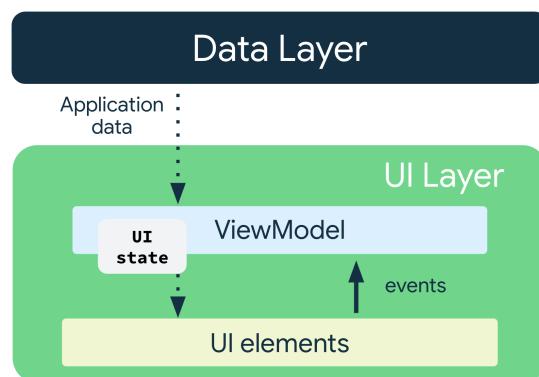
Vrstvy architektury

TODO Na základě těchto principů vznikla níže popsaná architektura, která je aktuálně doporučovaná pro vývoj mobilních Android aplikací.

Ta se rozděluje do následujících třech vrstev:

UI vrstva

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



■ Obrázek 3.2 Architektura UI vrstvy

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). ViewModel také poskytuje metody a události pro interakci s daty, jako je načítání dat ze zdroje, 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 částí UI vrstvy, který popisuje aktuální stav a chování uživatelského rozhraní v daném okamžiku. 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 interaktivity 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, 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, která zjednodušeně popisuje to, že na základě stavu je tvořeno výsledné uživatelské rozhraní.



■ **Obrázek 3.3** Reprezentace vztahu UI a stavu aplikace

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 tvorící vizuální část aplikace a umožňují uživatelům interagovat s aplikací. 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 jsou 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. Patří k nim například kliknutí myší, dotyk na obrazovce, stisknutí klávesy nebo jakékoli jiné akce prováděné uživatelem, které vyvolávají reakci v UI aplikace. Tyto události jsou zpracovány aplikací pomocí metod jako je například metoda `onClick()` a mohou spustit různé akce, které by zároveň měli způsobit změny UI stavu.

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. Často se tímto způsobem "odlehčuje" ViewModeley, které by jinak obsahovali 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 zjednoduší

testování, údržbu a rozšíření aplikace a umožňuje snadnou změnu technologií nebo platformy bez vlivu na doménovou logiku.

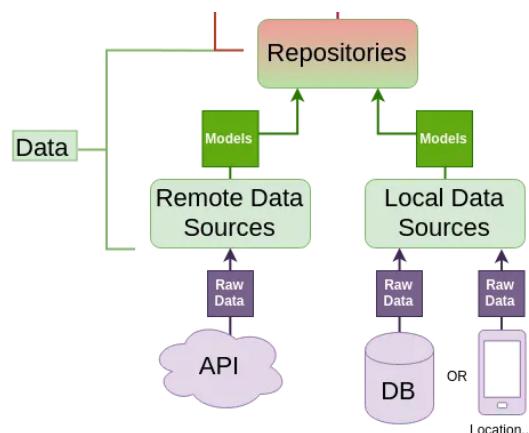
Datová vrstva

Datová vrstva se zabývá persistencí dat a komunikací 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.

K tomu aby aplikace fungovala, tak jak bylo popsáno v úvodu této kapitoly, je zapotřebí aby datová vrstva této aplikace obsahovala mechanismy pro získání dat z informačních systémů města. Jelikož se na základě provedené analýzy data v informačních systémech nachází na ... ve formátu XML,

byla proto navržena služba získávající tato data ze vzdáleného datového zdroje (IS města) tak jak se tomu v levé části obrázku 3.4.

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



■ Obrázek 3.4 Architektura datové vrstvy

Mezi důležité prvky datové vrstvy patří:

DAOs

DAOs jsou zodpovědné za přímý přístup k datovým úložištěm a provádění operací jako čtení, zápis, aktualizace a mazání dat. Tyto objekty poskytují abstrakci nad konkrétními technologiemi datových úložišť a umožňují snadnou změnu úložišť bez vlivu na ostatní části aplikace.

DTOs

DTOs 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.

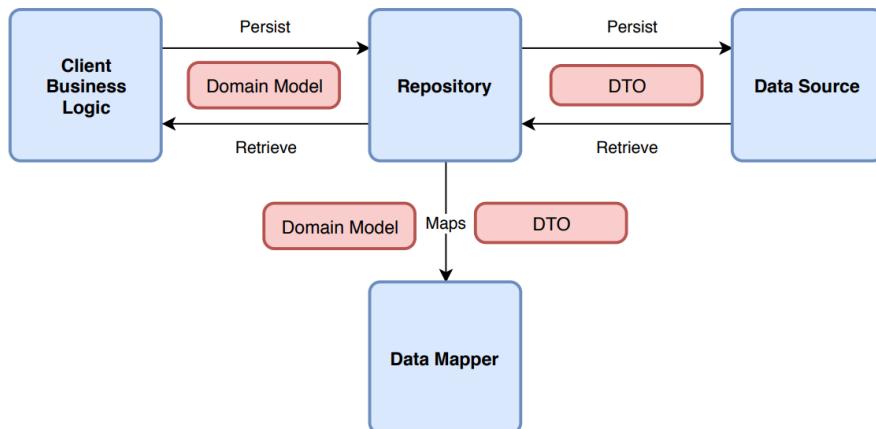
Datové entity

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

Datová úložiště

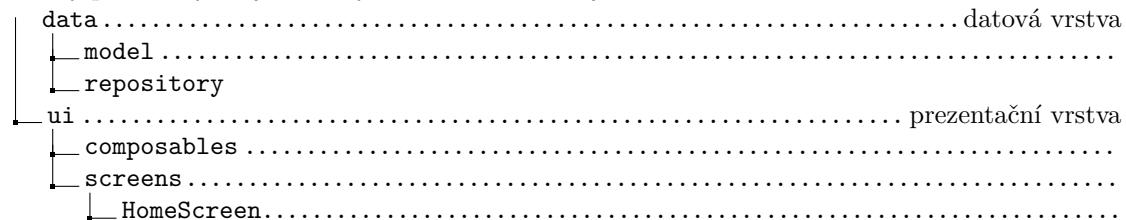
Datové úložiště jsou fyzická místa, kde jsou data uložena. Mohou to být relační databáze, NoSQL databáze, soubory na disku nebo externí API. Datová vrstva zajišťuje, aby bylo možné efektivně pracovat s těmito úložišti a aby byla data bezpečně uložena a získána.

Tyto vrstvy jsou spolu propojeny následujícím způsobem.



■ **Obrázek 3.5** Provázanost jednotlivých částí datové vrstvy

Po shrnutí dosavadně zmíněných poznatků týkajících se architektury aplikace byly jednotlivé vrstvy převedeny do jednotlivých balíků s následující strukturou:



3.5 Uživatelské rozhraní

UI bylo navrhováno tak aby vyhovovalo

v Android dokumentaci jejíž části se zaměřují na návrh od jednotlivých tlačítek, textů až po to jak celé UI působí z pohledu uspořádání.

TODO pravidla stravného navrhu UI

3.5.1 Drátěné modely

K tomu aby bylo výsledné UI pro uživatele přívětivé bylo přistupováno tak aby co nejvíce s pravidly dobrého návrhu UI.

Mezi takoví pravidla patří: TODO

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

Zároveň byla snaha o

Domovská obrazovka

Jak již bylo zmíněno v úvodu této kapitoly, tak domovská obrazovka obsahuje většinu nejdůležitější a

Na základně...byla pro události vybrána možnost zobrazení jednotlivých událostí v posuvném řádku a to především díky možnosti zobrazit veškeré potřebné události kompaktně na malém prostoru, ale přesto uživateli nabídne možnost danou události rozkliknout a dohledat zbylé informace. Z ... ?? totiž vyplývá, že akce poutají především svým vizuálním největší část informace je uživateli předána právě pomocí titulního obrázku, názvu a času konání.

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

Mezi další UI prvky na domovské obrazovce patří tlačítka v její horní části, která uživateli umožňují rychlý proklik na YouTube kanál města, nahlášení závady, zobrazení úředních hodin a kontaktů.

Obrazovka událostí

V rámci obrazovky "Události" je použit stejný systém pro zobrazené události pomocí skrolovacího řádku jako je tomu na domovské obrazovce.

Oproti domovské stránce jsou zde však zobrazeny veškeré události, které se ve městě budou konat. Pro lepší přehled jsou události rozděleny do několika málo kategorií.

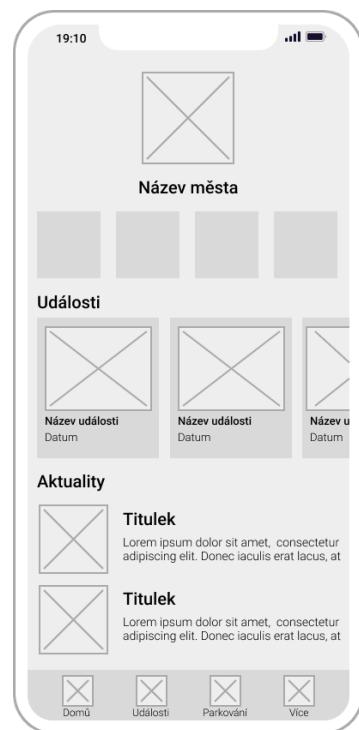
V horní části aplikace je uživateli dána možnost vybrat den pro který chce zobrazení plánované události, případně zvolit požadované rozmezí.

Byla navržena tak, aby uživatelé mohli jednoduše filtrovat zobrazované události primárně podle času.

Filtrování podle kategorie bylo nejprve navrženo ale následně z něho bylo umístěno kvůli malému počtu kategorií.

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

veškeré kategorie se zobrazí na maximálně 3 stránkách což se ukázalo



Obrázek 3.6 Drátěný model obrazovky *Domů*



Obrázek 3.7 Drátěný model obrazovky *Události*

Obrazovka parkování

Obrazovka parkování slouží uživateli především k vyhledání parkovací zóny, získání informací o ceně parkování případně k platbě parkovného.

Aby bylo docíleno co nejjednodušší manipulace s mapovými podklady byl prostor obrazovky navržen tak aby hlavní část byla věnována právě mapovému podkladu na němž jsou vyznačeny parkovací zóny.

Pro usnadnění hledání parkovacích zón dává aplikace uživateli možnost vyhledání požadované parkovací zóny pomocí vyhledávací pole, které uživateli 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 nějakou z nabízených parkovacích zón. Nezmenšuje tak uživateli 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 dá uživateli možnost zaplatit parkovné.

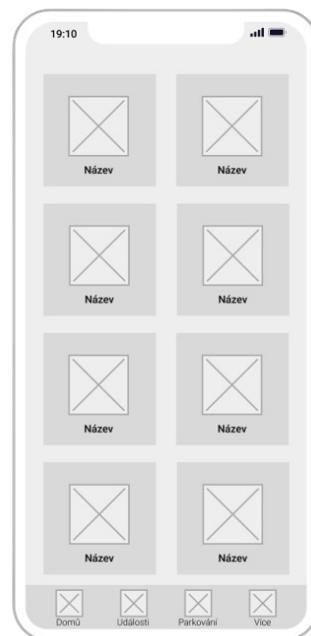
Poslední funkci, kterou tato obrazovka nabízí je zobrazení aktuálního provozu ve městě.

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 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.

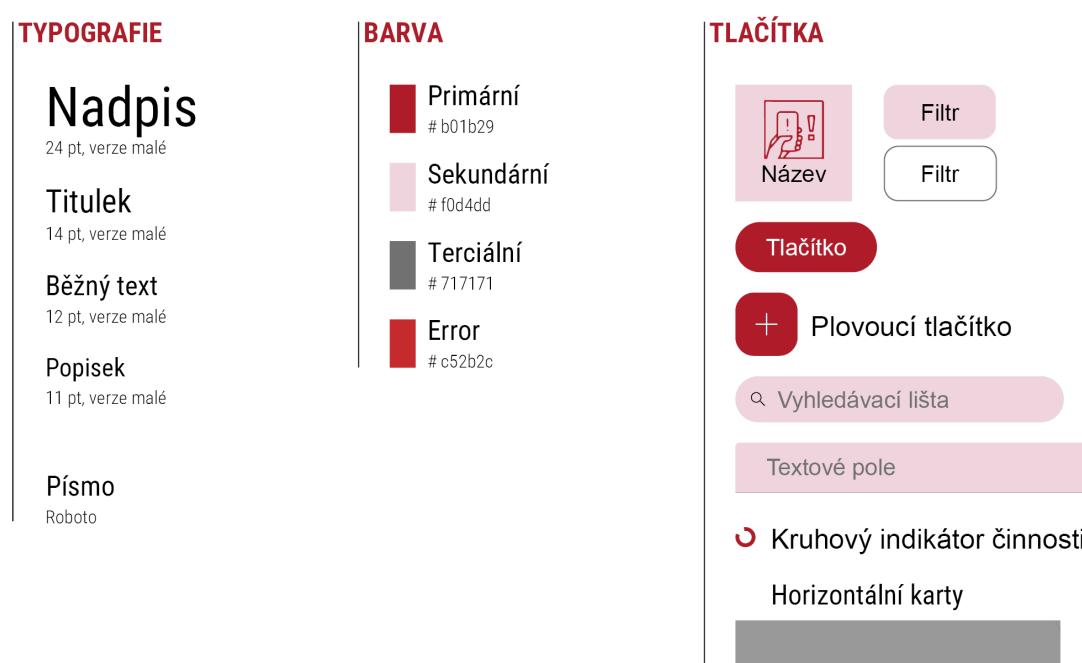
V rámci vybraného města Příbrami není takový systém nikde explicitně zadefinovaný a proto byl odvozen z designu webových stránek města, pro navození maximálního pocitu jednoty s dosavadními informačními systémy města.

Zároveň byl zkombinován s aktuálně nejnovějším design systémem od společnosti Google zvaným Material 3, který se hodí nejen pro použití na mobilních platformách.

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 prvky použitelné na mobilních platformách a zbylé části mohou být případně v budoucnu rozšířeny o další prvky používané na jiných platformách.

V tomto případě bylo navrženo pár základních UI prvků, které budou použity v rámci této 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 navrženém designu systému 3.5.2 je již samotný návrh grafické podoby UI zjednodušen na nezbytné minimum.

Vytvořené mockup modely jsou defacto výsledkem sloučení drátěných modelů s navrženým designem systémem.

při návrhu grafické podoby UI byla proto primárně soustředěn na význam jednotlivých UI komponent a na základě jejich významu byla přiřazovány odpovídající barvy.

Navržené UI se 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 pohledu přístupnosti je UI aplikace navrženo tak, aby jednotlivé prvky UI měly vhodně zvolenou barvu z pohledu správného kontrastu vůči podkladu a nebo 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 nevidomí nebo jinak zrakově postiženým lidem používat aplikaci.

Barvy

I přesto, že základní barvy byly zvoleny již v při navrhování designu systému, tak během fáze návrhu mockup modelu byl zároveň kladen důraz na jejich vhodné použití z pohledu sémantiky barev nebo důležitosti barvených komponenta.

V základu *Material Theme Builder* umožňuje definovat 3 základní barvy (primární, sekundární, ternární), které mají jasně vymezené role. Primární barva je v případě implementovaného UI například použita pro potvrzovací akci "zplacení parkovného". Sekundární barva byla implementována například na filtrační tlačítka na obrazovce "Události", jelikož hlavním předmětem pozornosti na dané obrazovce jsou zobrazované akce a ne filtrační možnosti.

Rozložení

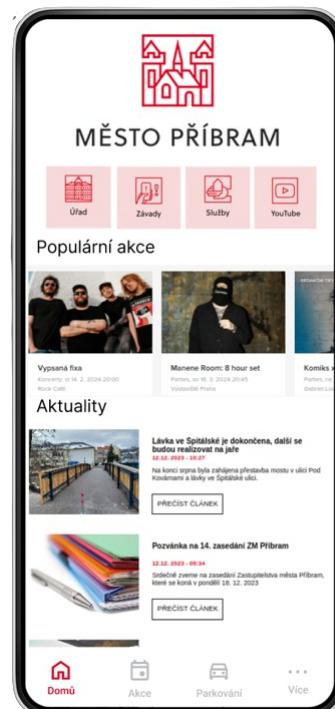
Co se týká rozložení UI komponent na obrazovky, tak zde bylo respektováno několik pravidel jako například liniového zarovnání UI komponent, tak aby nedocházelo k narušení čitelnosti.

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

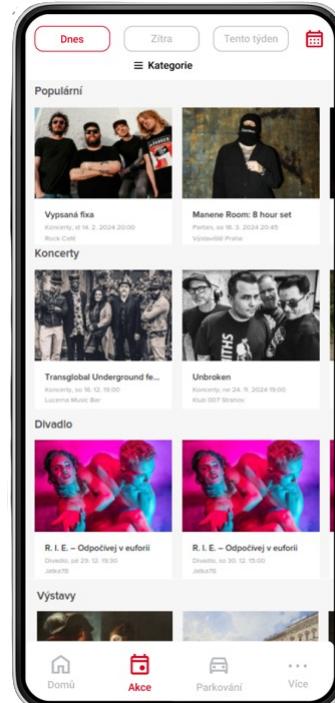
Pro obrázky byly voleny běžné poměry jako například 4:3, tak aby nedocházelo k jejich zbytečnému ořezávání.

Zároveň

Sytost barev byla zvolena na základě

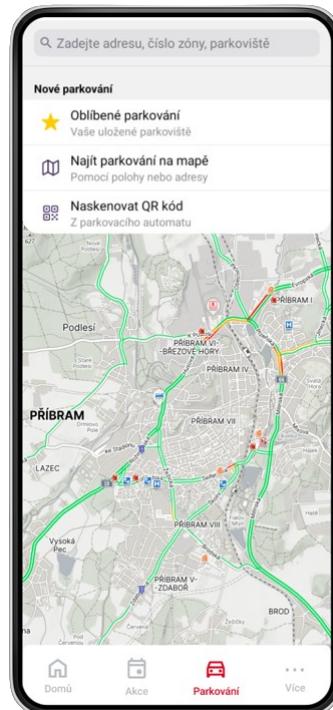


■ Obrázek 3.11 Obrazovka Domů

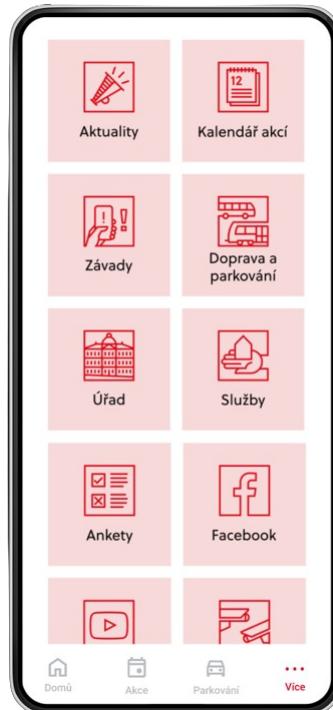


■ Obrázek 3.12 Obrazovka Události

TEXT DRUHE STANKY



■ Obrázek 3.13 Obrazovka *Parkování*



■ Obrázek 3.14 Obrazovka *Více*

Kapitola 4

Implementace

Stejně tak jako v případě vývoje Android aplikací, tak k sestavování multiplatformních aplikací založených na Compose Multiplatform se používá nástroj zvaný Gradle.

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 technologie KMP. O sestavování takto založeného projektu se stará nástroj Gradle.

4.1.1 Gradle

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

Gradle používá deklarativní doménově specifické jazyky (DSL), díky kterým umožňuje 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 používána 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).

Dále je zapotřebí vybrat platformy, pro které má být aplikace sestavována a podle nich vytvořit příslušné balíčky viz kapitola 2.1.3.1. V případě implementované aplikace se jedná o platformy Android a iOS, které bylo nutné specifikovat v projektovém souboru `build.gradle.kts` (viz řádky 2 - 30 na výpisu kódu 4.2). Podle těchto platforem lze následně v příslušném `build.gradle.kts`

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

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

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 jedna o multiplatformní (`commonMain`) nebo nativní závislost (`androidMain`, `iosMain`, atd.).

Pro ukázku jak taková multiplatformní závislost může vypadat slouží řádky 32 - 37 na ukázce kódu 4.2, pomocí kterých je do projektu přidána knihovna umožňující implementaci multiplatformní navigace.

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 napsána v rámci sdílené části aplikace.

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

Stejným způsobem je potřeba integrovat tuto funkci i do ostatních spouštěcích souborů.

Jelikož časem projekt obsahuje je vhodné pro správu verzí použít takzvaný version katalog.

Gradle version catalog

Gradle version catalog je obyčejný soubor ve formátu TOML, který umožňuje snadněji přidávat a spravovat závislosti a pluginy ve celém 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.

4.2 Tvorba UI obecně

4.2.1 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 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. [37] Výčet aktuálně doporučených knihoven pro implementaci multiplatformní navigace je uveden v oficiální Kotlin multiplatform dokumentaci a zahrnuje 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.

;

4.2.2 Design systém

Při implementaci design systému pomocí Compose Multiplatform byla snaha co nejpřesněji napodobit navržený design systém z kapitoly 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.

■ Výpis kódu 4.2 Lib integration

```
kotlin {  
    androidTarget {  
        compilations.all {  
            kotlinOptions {  
                jvmTarget = "11"  
            }  
        }  
        @OptIn(ExperimentalKotlinGradlePluginApi::class)  
        instrumentedTestVariant {  
            sourceSetTree.set(KotlinSourceSetTree.test)  
  
            dependencies {  
                implementation("androidx.compose.ui:ui-test-junit4-android:1.6.4")  
                debugImplementation("androidx.compose.ui:ui-test-manifest:1.6.4")  
            }  
        }  
    }  
    jvm("desktop")  
  
    listOf(  
        iosX64(),  
        iosArm64(),  
        iosSimulatorArm64()  
    ).forEach { iosTarget ->  
        iosTarget.binaries.framework {  
            basePath = "ComposeApp"  
            isStatic = true  
        }  
    }  
  
    sourceSets {  
        commonMain.dependencies {  
            implementation(libs.voyager.navigator)  
            ...  
        }  
    }  
}
```

■ Výpis kódu 4.3 Lib integration

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

■ 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" }
```

■ 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)
...
```

■ 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,
    secondary = md_theme_light_secondary,
    ...
)
private val DarkColors = darkColorScheme(
    primary = md_theme_dark_primary,
    onPrimary = md_theme_dark_onPrimary,
    primaryContainer = md_theme_dark_primaryContainer,
    ...
)
```

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 zadefinovat následujícím způsobem:

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

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

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
    )
}
```

■ Výpis kódu 4.8 Ukázka použití stylu písma

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

Typografie

Vybraný Material Design definuje celkem 15 stylů písma, kde každé z nich má předem definovaný způsob použití. [38] Jelikož se jedná o styly písma přímo definované 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 definovaný v rámci kapitoly 3.5.2

4.3 Použité technologie

Následující podkapitoly 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 byly zmíněny některé důležité technologie na kterých je postavena aplikační logika aplikace.

4.3.1 Vrstva uživatelského rozhraní

Voyager

Voyager je multiplatformní navigační knihovna, která umožnuje se pohybovat mezi různými obrazovkami a destinacemi v multiplatformních mobilních aplikacích. [39] Jejím hlavním cílem je poskytnout jednotné rozhraní pro navigaci na různých mobilních platformách, jako jsou Android

■ **Výpis kódu 4.9** 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() },
    )
}
```

a iOS, a zároveň maximalizovat sdílený kód a snižovat duplikaci kódu.

V rámci implementované aplikace byla navigace Voyager použita k implementaci základní navigace mezi jednotlivými obrazovkami. Příkladem takové navigace jsou veškeré možné přechody v rámci domovské obrazovky. Tato navigace využívá 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 a je znova obnoven 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. *Stack API* zároveň umožňuje použití několika funkcí jako jsou například `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 TODO

Způsob navigace založený na *Stack API* však není vhodný pro navigování mezi jednotlivými záložkami aplikace (lišta ve spodní části displaye), a proto je tato část navigace implementována pomocí takzvané *Tab navigation*, kterou knihovna Voyager, takéž 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. [40]

TODO

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

■ Výpis kódu 4.10 Coil

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

■ Výpis kódu 4.11 Konfigurace http klienta

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

potřebám projektu.

Coil 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.

4.3.2 Vrstva aplikační logiky

Kotlin

Coroutines

Ktor

Ktor je open-source framework vytvořený společností JetBrains pro tvorbu jak serverových, tak klientských aplikací pomocí programovacího jazyka Kotlin. Ktor byl vyžit 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, díky kterému je Ktor možné použít na platformách Android, JVM a všech platformách, které podporuje Kotlin/Native.

Po nakonfigurování http klienta je již možné zasílat běžné http dotazy pomocí metod `post`, `get`, `put` nebo `delete`.

■ **Výpis kódu 4.12** Zaslání požadavku

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

■ **Výpis kódu 4.13** DI databázového ovladače pomocí Koinu

```
class MainActivity : ComponentActivity() {

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

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

Koin

Koin je moderní framework pro vkládání závislostí v aplikacích napsaných v jazyce Kotlin. Jedná se o lehkou a snadno použitelnou alternativu k jiným známým knihovnám, jako je například Hilt pro platformu Android. Koin se zaměřuje na jednoduchost použití a minimalismus, což usnadňuje vývoj aplikací a zároveň zlepšuje čitelnost a údržbu kódu.

Hlavním principem Koinu je deklarativní přístup k definici závislostí pomocí tzv. modulů. V modulu se specifikují všechny závislosti a jejich vytváření, a to pomocí jednoduchých funkcí nebo tzv. singletónů. Tento přístup umožňuje snadnou konfiguraci závislostí a poskytuje vývojářům transparentní pohled na strukturu aplikace.

V rámci této aplikace byl Koin použit například pro vložení databázového ovladače za pomocí expect actual deklarace 2.1.3.1. Využití tohoto zápisu bylo vhodné použít především díky tomu, že převážná většina databázové logiky je napsána v rámci sdílené logiky a v rámci nativní logiky, byl tak implementován pouze databázový ovladač.

TODO nejen driver screenmodel, repository

SQLDelight

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

Hlavní funkcionality knihovny SQLDelight je automatická generace Kotlin tříd na základě definovaných SQL schémat. To znamená, že vývojáři mohou psát SQL dotazy pomocí standardní SQL syntaxe a SQLDelight se postará o vygenerování příslušných tříd 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 `selectAllNews:` ve výpisu kódu 4.14) pomocí kterého je vygenerována příslušná Kotlin funkce.

Vygenerové metody (dotazy) lze následně nalézt ve složce `build/generated/sqlodelight/...`

■ **Výpis kódu 4.14** SQL dotaz

```
selectNewsById:
SELECT *
FROM News
WHERE id = :id;
```

■ **Výpis kódu 4.15** SQL vygenerovaný dotaz

```
private inner class SelectNewsByIdQuery<out T : Any>(
    public val id: Long,
    mapper: (SqlCursor) -> T,
) : Query<T>(mapper) {
    override fun addListener(listener: Query.Listener) {
        driver.addListener("News", listener = listener)
    } TODO
}
```

a pro dotaz z výpisu kódu 4.14 vypadá následovně:

SQLDelight také nabízí silnou typovou kontrolu při práci s daty v databázi. To znamená, že chyby v SQL dotazech jsou odhaleny již při komplikaci kódu, což usnadňuje odhalování chyb a zvyšuje stabilitu aplikace.

I přesto, že většina kódu může být psána v balíčku sdíleném mezi všemi platformami, je zde i část z předchozí ukázky ??, kterou je pro platformy Android, JVM, a Native potřeba naimplementovat nativně. Konkrétně se jedná o ovladače k jejichž implementaci jsou potřeba nativní knihovny a musí být proto implementovány v balíčcích jednotlivých platform.

TODO

4.4 Logika uživatelského rozhraní

V rámci této kapitoly je ukázáno jakým způsobem byla implementována logika chování UI navržená v kapitole *Vrstvy architektury 3.4*.

■ **Výpis kódu 4.16** Nativní databázový ovladač

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

■ Výpis kódu 4.17 Event State katalog

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

■ Výpis kódu 4.18 Události uživatelského rozhraní

```
val screenModel = getScreenModel<HomeScreenModel>()
val state by screenModel.state.collectAsState()

val news = state.news.collectAsState()
LazyColumn {
    items(news.value) { news ->
        NewsItem(news = news, onItemClick = {
            navigator.push(NewsDetailScreen(news))
        })
    }
}
```

4.4.1 Stav uživatelského rozhraní

Tato podkapitola slouží k přiblížení toho, jak takzvaný *UI State*, kterému byl věnován paragraf *UI state* v kapitole *Vrstvy architektury 3.4* je implementován v rámci ukázkové aplikace.

Z pohledu UI se jedná o důležitou část aplikace o jejíž aktualizaci se stará již zmíněný ViewModel 4.4.3 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 ukázkou toho jak byla taková datová třída implementována byla vybrána obrazovka událostí (viz 4.17), 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 načítány nějaká data.

Takto zadefinované stavy je následně možné používat v rámci jednotlivých obrazovek a ty jsou při jakékoli změně tohoto stavu znova sestaveny.

být v aplikaci použity například následujícím způsobem. Kde stav udržován

4.4.2

V rámci této kapitoly bude představeno jak implementovaná aplikace obsluhuje události přicházející z uživatelského rozhraní.

V rámci ViewModelů jsou obsluhovány události (viz kapitola 3.4), které přichází z UI elementů. V jednotlivých j

■ **Výpis kódu 4.19** Použití stavu v aplikaci

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

■ **Výpis kódu 4.20** Implementace ViewModelu

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

4.4.3 ViewModel

TODO V rámci implementované aplikace byl pro každou obrazovku implementován jeden ViewModel, který se stará o logiku každé obrazovky zvlášť. Tato kapitola bude proto shrnutím stěžejních částí jednotlivých ViewModelů tak

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

, která ovlivnila implementaci ViewModelů bylo použití navigace Voyager, která poskytuje potřebná rozhraní k

4.5 Tvorba UI obrazovek

TODO

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 představují stavební bloky celého UI mohou sloužit k reprezentaci jednotlivých tlačítek, textů vstupní polí, seznamů, ale také celým souborům těchto prvků jako jsou například celé obrazovky.

Na následujícím výpisu kódu 4.21 je proto ukázáno jak mohou být do sebe jednotlivé Com-

■ **Výpis kódu 4.21** Popis UI widgetů pomocí jazyka Kotlin

```
@Composable
fun LatestNewsScreen(
    viewModel: NewsViewModel = viewModel()
) {
    // Show UI elements based on the viewModel.uiState
}
```

■ **Výpis kódu 4.22** Implementace posuvného řádku pomocí `LazyRow`

```
@Composable
fun EventLazyRow(category: EventCategory, state: EventsState) {
    val scrollState = rememberLazyListState()
    val coroutineScope = rememberCoroutineScope()
    val navigator: Navigator = LocalNavigator.currentOrThrow

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

posable funkce zanořeny čímž tvoří takzvaný sémantický strom taktéž popsaný v kapitole *Architektura frameworku Compose Multiplatform 2.1.3*

Obsahem následujících podkapitol 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 drátěných modelů (viz kapitola *Tvorba drátových modelů 3.5.1*) a následně bylo stylizováno tak, aby odpovídalo navržené podobě UI z kapitoly *Mockup modely 3.5.3*

4.5.1 Domovská obrazovka

Uživatelské rozhraní domovské obrazovky je kompletně implementováno pomocí frameworku Compose Multiplatform a nebylo při jeho implementaci potřeba přistoupit k nativnímu řešení ani na platformě iOS nebo desktop.

K implementaci posuvného řádku, který slouží pro zobrazení událostí byla použita komponenta vlastní *Composable* komponenta zvaná `EventLazyRow`, která je založená na základní `LazyRow`, která umožňuje zobrazení velké

Na stejném principu jako `LazyRow`, funguje i komponenta `LazyColumn`.
I implementaci

LAZY ROW LAZY COLUMN , OBRAZKY POMOCI COIL, STYLOVANI POMOCI DESIGN SYSTEMU, VYUZITI DATABAZE, ukazka Event lazy row komponenty

Obě tyto komponenty navíc nabízí postupné načítání jejich obsahu, což nezpomaluje načítání obsahu v případně většího množství novinek nebo událostí.

4.5.2 Obrazovka *Události*

V rámci obrazovky "Události" je používána stejná komponenta pro zobrazování událostí jako byla použita na domovská 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. Následně pro každou kategorii 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 uživateli umožňuje zvolit požadované časové rozmezí.

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

Celá obrazovka je vzhledem k nepředvídatelnosti zobrazovaných událostí implementována jako rolovací seznam.

LAZY ROW , DATE PICKER, SPECIAL CHIPS

4.5.3 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 4.23) z balíčku `com.google.maps.android.compose`, která umožňuje vložení mapových podkladů na platformě Android a pro propojení této nativní části se zbytkem aplikace byla použita `expect/actual` deklarace.

Implementace motivu mapy

Aby byla implementace motivů z kapitoly ?? komplementní bylo zapotřebí přizpůsobit i mapové podklady

kreré se mění v závislosti na aktuální motivu zařízení.

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

■ Výpis kódu 4.23 GoogleMap element

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

■ Výpis kódu 4.24 Motiv mapy ve formátu JSON

```
{  
    "featureType": "road.highway",  
    "elementType": "labels.text.fill",  
    "stylers": [  
        {  
            "color": "#616161"  
        }  
    ]  
},  
{  
    "featureType": "road.local",  
    "elementType": "labels.text.fill",  
    "stylers": [  
        {  
            "color": "#9e9e9e"  
        }  
    ]  
},
```

data byla získána 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 objektů použit stejný nástroj jako pro převod veškerých dat přicházejících s z informačních systémů města.

Zobrazení parkovacích zón na mapě bylo

Pro vycentrování vybrané zobrazení vybrané parkovací zóny na obrazovce byla napsána funkce pro vycentrování mapy tak, aby se zvolená zóna pokaždé umísnila 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.

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

Dále byla tato obrazovka rozšířena o vyhledávací pole a výsuvný panel, které bylo již implementováno v rámci společné části aplikace.

, které uživateli dává možnost jednoduchého vyhledání parkovací zóny podle názvu nebo čísla parkovací zóny.

Dále na stejně obrazovce implementován 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.

4.6 Řešení problémů spojených s multiplatformním vývojem

Nedostatek knihoven Mezi jeden z hlavních problémů patří nedostatek vhodných multiplatformních knihoven a ten se projevil i v případě implementace mapových podkladů do aplikace kde jak již bylo zmíněno byla nakonec zvolena implementace nativních mapových podkladů za použití `expect` a `actual` deklarací (viz paragraf 2.1.3.1).

Preview Další problémem bylo nefunkční náhled UI, což se sice na výsledném uživatelském rozhraní nikterak neprojevilo, ale celý proces tvorby UI to značně prodloužilo.

Špatná nebo téměř žádná dokumentace k knihovnám třetích stran a v případě co se týče Compose Multiplatfrom, tak zde je téměř kompletně spoléháno na identic. Na druhou stranu Compose Multiplatform ještě není na všech platformách ve stabilní verzi a proto je možné, že po vydání stabilních verzí i pro jiné platformy se dokumentace od JetBrains rozšíří.

problémy s IDE

..... Kapitola 5

Testování

Tato kapitola se bude věnovat předvším UI teestování a to z několika růzých pohledů.

5.1 Uživatelské rozhraní

Od verze 1.6.0 Compse multiplatform umožnuje testování UI na všech platformách. [41]

UI testy se používají k ověření, že uživatelské rozhraní aplikace funguje podle očekávání. To zahrnuje testování prvků uživatelského rozhraní, uživatelských interakcí a navigačních toků, aby se zajistilo plynulé uživatelské prostředí.

Mezi základní typy UI testů se řadí UI testy testující kritické uživatelské interakce na jedné obrazovce a dále například navigační testy testující správnost fungování navigace v dané aplikaci.

5.2 Výkon aplikace

5.3 Přístupnost aplikace

5.4 Kompatibilita

Testování aplikace na různých zařízeních s různými velikostmi obrazovek, rozlišeními a hardwarovými konfiguracemi, aby se zajistilo, že funguje správně na široké škále zařízení.

5.5 Možnosti testování UI v Compose Multiplatform

Testování aplikací založených na frameworku Compose Multiplatform je stejně tak jako tvorba samotného UI založena na Jetpack Compose a využívá proto i stejných konceptů. Mezi tyto klíčové koncepty testovaní UI se řadí následující:

Testování sémantiky

V rámci testování

API rozhraní

Compose Multiplatform z toto důvodu také využívá tři hlavní principy jak testovat UI, které se v Jetpack compose označují jako *Finders*, *Assertions* a *Actions*. K tomu aby bylo možné UI komponenty testovat se používají funkce, které tyto komponenty na obrazovce detekují a

■ Výpis kódu 5.1 Integrace UI testů pomocí Gradle

```
kotlin {  
    //...  
    sourceSets {  
        val desktopTest by getting  
  
        // Adds common test dependencies  
        commonTest.dependencies {  
            implementation(kotlin("test"))  
  
            @OptIn(org.jetbrains.compose.ExperimentalComposeLibrary::class)  
            implementation(compose.uiTest)  
        }  
  
        // Adds the desktop test dependency  
        desktopTest.dependencies {  
            implementation(compose.desktop.currentOs)  
        }  
    }  
}
```

následně další funkce které nad nalezenými komponenty umožňují provést akce podobné těm, které provádí uživatel. Pro kontrolu správnosti provední těchto akcí se používají takzvané tvrzení (Assertions), které ověří zdali určité UI prvky mají požadované atributy.

Finders umožňují najít uzel nebo uzly ve stromu UI struktury pomocí tagů, vnořených textů, různých popisků a nebo pomocí *Matchers*.

Pomocí *Assertions*

A pomocí *Actions* je možné simulovat uživatelské integrace jako jsou kliknutí nebo jiná gesta.
[42]

Testování synchoronizace

Testování interoperability

5.6 Testovací případy

Testovací případ je sada kroků nebo akcí, které jsou prováděny při testování softwarového produktu s cílem ověřit, zda se chová podle očekávání a splňuje požadavky. Každý testovací případ obvykle obsahuje následující prvky:

- Popis: Stručný popis toho, co testovací případ testuje a jaké jsou očekávané výsledky.
- Předpoklady: Podmínky, které musí být splněny nebo konfigurace, která musí být provedena před spuštěním testu.
- Kroky: Konkrétní kroky, které musí být provedeny k provedení testu.
- Očekávané výsledky: Popis očekávaných výsledků testu po dokončení kroků.

- Aktuální výsledky: Skutečné výsledky testu, které jsou porovnány s očekávanými výsledky k určení úspěšnosti testu.

Na základě této struktury byly sepsány následující testovací případy:

Testovací případ 1: Zobrazení aktuálních novinek

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 je zobrazen.
 - Po kliknutí na konkrétní novinku jsou zobrazeny její detaily.

Testovací případ 2: Zobrazení aktuálních událostí

1. Kroky:
 - a. Navigace na stránku zobrazující aktuální události.
 - b. Ověření, že systém zobrazil seznam aktuálních událostí.
 - c. Kliknutí na konkrétní událost.
 - d. Ověření, že systém zobrazil detaily vybrané události.
2. Očekávaný výstup:
 - Seznam aktuálních událostí je zobrazen.
 - Po kliknutí na konkrétní událost jsou zobrazeny její detaily.

Testovací případ 3: Filtrování zobrazených událostí podle data

1. Kroky:
 - a. Navigace na stránku zobrazující události.
 - b. Ověření, že systém zobrazil seznam událostí.
 - c. Zvolení konkrétního data pro filtrování událostí.
 - d. Ověření, že systém zobrazil události pouze pro vybrané datum.
2. Očekávaný výstup:
 - Seznam událostí je filtrován podle zvoleného data.
 - Události jsou rozděleny do kategorií podle typu události.

■ Výpis kódu 5.2 Implementace UI testu

```
class ExampleTest {
    @OptIn(ExperimentalTestApi::class)
    @Test
    fun myTest() = runComposeUiTest {
        // Declares a mock UI to demonstrate API calls
        //
        // Replace with your own declarations to test the code of your project
        setContent {
            var text by remember { mutableStateOf("Hello") }
            Text(
                text = text,
                modifier = Modifier.testTag("text")
            )
            Button(
                onClick = { text = "Compose" },
                modifier = Modifier.testTag("button")
            ) {
                Text("Click me")
            }
        }

        // Tests the declared UI with assertions and actions of the Compose Multiplatform testing
        onNodeWithTag("text").assertTextEquals("Hello")
        onNodeWithTag("button").performClick()
        onNodeWithTag("text").assertTextEquals("Compose")
    }
}
```

5.7 Implementace testů

Testy jednotlivých UI komponent

End-to-end testy

5.8 Zhodnocení výsledků testování

5.9 Zhodnocení použitelnosti

android already in production with hell a lot of features now If you want to share code among these and have single code base, going with flutter is bad idea .. Since its take nativity out. So hence you want both the teams to have a common business logic and networking thing then KMM is the way to go

..... Kapitola 6

Závěr

6.1 Evaluace vlastností Compose Multiplatform ve srovnání s cíli práce

Vetšinu cílů, které byly stanoveny v zadání bylo možné pomocí multiplatformního frameworku Compose Multiplatform splnit a nebylo tak nutné přistupovat na kompromisy. Veškeré implem-

Nicméně při implementační náročnějších částech aplikace bylo již občas nutné přistoupit k použití nativních řešení a tím pádem tyto dva přístupy zkombinovat. Ve výsledku z pohledu UI, tak tato kombinace nevedla k zádným ústupkům Nicméně díky vhodně navrženým principům jako je Expect a actual deklarace (viz paragraf 2.1.3.1) nebylo skombinování těchto odlišných částí nikterak implementačně komplikované a ne výsledku ani nepřehledné.

Fakt že v některých případech bylo nutné přistoupit k nativnímu řešení nemusí být brán jakožto nevýhoda Compose Multiplatform oproti ostatním frameworkům. Na druhou stranu lze tatu možnost vnimat i jako výhodu, kterou ostatní frameworky neposkytují a dělá takz compose Multiplatform z tohoto pohledu flexibilnější framework.

prostor o rozšíření ostatních naticvních knihoven o mmmultipatformní API.

Čehož výsledky by v ideálním případě bylo na daném vývojáři zdali zvolí nativní či multipalt-formí cetu vývoje.

6.2 Zkušenosti z implementace na reálné aplikaci

Preview Nedostatek knihoven Špatná nebo též žádna dokumentace k knihovnám třetích stran a v případě co se týče Compose Multiplatfrom

6.3 Závěr o použitelnosti frameworku

Jelikož je závěrečné hodnocení této nově nastupující technologie stěžejní částí práce, tak aby bylo toto hodnocení co nejobjetivnější, tak bude rozvrženno do několika částí.

Nejprve bude framework porovnán na základě provedné analýzy a to jednak z pohledu architektury,

Co se pokrytí aktuálně používaných platforem týka, tak Compose Multiplatform pokrývá většinu z aktuálně nejžádanějších platforem pro vývoj multipaltformních aplikací a dá se tak očekávat, že po uvolnění stabilních verzí pro veškeré aktuálně nabízené platformy se jeho obliba ještě zvýší.

V oblasti výkonu se Compose Multiplatform docela dobře daří a je konkurenceschopný s ostatními multiplatformními frameworky. V případě větších projektů by však tento rozdíl mohl být výraznější.

Rychlosť vývoje je také významným hlediskem. Co se rychlosti vývoje týká, tak při tvorbě UI, tak v porovnání s jinými frameworky, jako je například Flutter, má Compose Multiplatform strukturu kódu s méně znaky.

V oblasti optimalizace má však Compose Multiplatform ještě poměrně velké nedostatky a to zejména v plynulosti provozu na platformě iOS.

Pokud jde o připravenost IDE, situace je obdobná. Při implementaci a to před opriti Flutteru nenebízí tolik možností upravy už jako například zanoření některých komponent do jiných, ale díky jednoduchosti zápisu UI jsou tyto akce o něco jednosušší.

A co se výběru IDE týka, tak z otestovaných IDE Android Studio jednoznačně předčilo Fleet. Nicméně tato situace se může, také rychle změnit.

V průběhu implementace bylo narazeno na několik problémů, které byli nahlášeny a postupně se na nich začínala pracovat.

Z pohledu komunity se s,

Z pohledu dokumentace jsou zde velikoé nedostatky, ale díky Jetpack Compose není potřeba a většina postupů jak jednotlivé části UI tvořit je dobře dokumentovaná společností Google.

Z pohledu KMP je tato situace horší I přesto, že je KMP již ve stabilní verzi, tak dokumentace oproti ostatním technologiím velmi zaostává.

Následně bude porovnán z pohledu implementace jak z pouhledu tvorby UI, tak z pohledu tvorby aplikační logiky,

Při závěrečném hodnocení frameworku Compose Multiplatform se dá říci, že je Compose Multiplatform vhodným nástrojem pro vývoj multiplatformních aplikací, ačkoli existují oblasti, ve kterých může být ještě zlepšen.

Co se týka náročnosti implementace aplikací na ostatních platformách, tak zde byla náročnost o něco zvýšena kvůli nemožnosti implementace iOS aplikace již od začátku. jelikož je ke kompliaci potřeba

Podarilo se potvrdit, že množství sdíleného kódu, které základem provedné analýzy bylo zjištěno je můžné sdílet. V rámci implementované aplikace bylo orecento sdíleného kódu ještě větší a to především díky sdílení UI pomocí Compose Multiplatform, ale také možná kvůli menšímu rozsahu aplikace oproti porovnávaným aplikacím.

6.4 Shrnutí dosažených výsledků

Podařilo se navrhnu aplikaci tak aby fungovala na více platformách.

Součástí návrhu byl také obecně použitelný designový systém, který může být použit k rozšíření nebo tvorbě nových aplikací v navrženém stylu.

Aplikace byla úspěšně přizpůsobena konkrétnímu městu a napojena na jeho infrastrukturu pro získávání dat o událostech, aktualitách a parkovacích zónách. Díky implementaci lokální databáze je aplikace použitelná i v offline režimu.

Díky navržené architektuře je možné aplikaci kdykoliv v budoucnu rozšířit o další funkce.

Poskytuje občanů rychlý přehled o aktálním dění ve městě a zároveň

Poskytuje moderní UI, které odpovídá moderním standartům společnosti Google pro vývoj mobilních aplikací. vsetne fukci jako je využití tmavého motivu.

Podařilo se naimplementovat multipaltformní aplikaci, která splňuje požadavky zadání a je spusitelná na mobilních platformách Android a iOS.

Zároveň shrnuje co největší množství kódu ve společné části, čím ukazuje možnosti frameworku a

zarověň ale ukazuje jak framework umožnuje implementaci navních částí,

Jedinečný design, pokocí čeho ukažu možnosti daklarativního zápisu UI pomocí Compose Multiplatform.

Z pohledu UI splňuje veškeré stanovené požadavky

S podařilo naimplementovat UI testy, které testují klíčové části uživatelského rozhraní.

Ukázalo se, že i v současné fázi vývoje je možné pomocí této technologie vytvořit použitelnou aplikaci.

6.5 Zhodnocení splnění cílů práce

6.6 Návrhy pro budoucí vývoj a výzkum

Z pohledu aplikace jsou zde velké možnosti pro budoucí vývoj at už z pohledu rychlosti UI, propojení s dalšími systémy města, implementaci další UI testu a unit testu otestování na vize zařízeních

A navrhnut tak pro města intuitivní aplikace, která by díky multipaltformím vlastnostem pokryla co největší množství zařízení a zároveň by byla snadněji udržovatelná.

Zároveň umožnuje snadnou integraci mapových prvků

nicméně

Časem se zajisté ukáží

mohou ukázat další možnosti využití tohoto frameworku na jiných platformách

uz je prubehu psaní této práce vzniklo několik nových funkcionalit které nebyly otevřeny jako například možnost implementovat výpovědi na platformě iOS stejně jako je tomu na Android zařízeních.

Příloha A

Snímky obrazovky

The image consists of two side-by-side screenshots of a mobile website for the city of Příbram.

Screenshot 1 (Left): This screenshot shows the homepage of the website. At the top, there is a header bar with the time (21:35), signal strength, battery level (97%), and a small camera icon. Below the header is the city's logo (a stylized building) and the word "PŘÍBRAM". There is a red horizontal bar with the text "Aktualizuj události" (Update events). Below this, there are four pink rectangular buttons with icons: "Úřad" (Building icon), "Závady" (Pencil icon), "Služby" (House icon), and "YouTube" (YouTube play button icon). The main content area is titled "Populární události" (Popular events) and features three event cards:

- Junior open air** (Photo of a stage performance) - So 25. 5. 2024
- Korzo Obora** (Photo of a person playing guitar) - So 25. 5. 2024
- HUDEČ An** (Logo featuring blue and yellow shapes) - HFAD: filharmonie Út 21. 5.

Below these cards is another section titled "Aktuality" (Actualities) with three items:

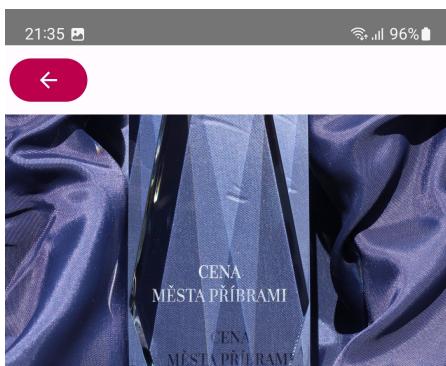
- Veřejná zakázka bude vypsána s nižší předpokládanou hodnotou** (Photo of pipes) - Pá 19. 4. 2024
- Pozvánka na 18. zasedání ZM Příbram** (Photo of a stack of papers) - Pá 19. 4. 2024
- Minutky z radnice - 19. 4. 2024** (Photo of a document cover) - Pá 19. 4. 2024

At the bottom of the page are navigation links: "Domů" (Home), "Akce" (Events), "Parkování" (Parking), and "Více" (More).

Screenshot 2 (Right): This screenshot shows a form for reporting a problem. At the top, it says "Záznam o závadě" (Report of a fault). It has a pink input field labeled "Zadejte název závady" (Enter the name of the fault). To the right is a large circular button with a camera icon and the text "Přidat fotografií závady" (Add photo of the fault). Below this is another pink input field labeled "Zadejte popis závady" (Enter a description of the fault). To the right of this is a dropdown menu labeled "Vyberte typ závady" (Select the type of fault) with the option "Pozemní komunikace" (Ground communication). At the bottom right is a purple button labeled "Odeslat závadu" (Send fault report).

■ Obrázek A.1 Domácí obrazovka

■ Obrázek A.2 Domácí obrazovka



Při čtvrtém ročníku putuje
ocenění do oblasti zdravotnictví

Čt 18. 4. 2024

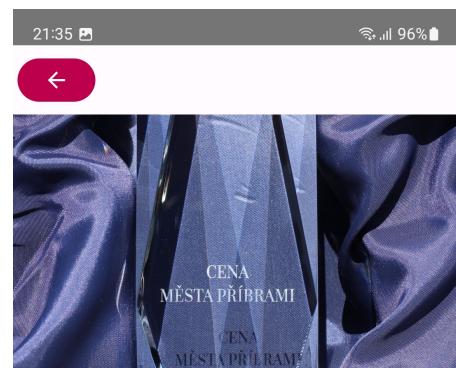
Již počtvrté se v Příbrami bude předávat Cena města Příbrami. Porota vybírala z dvacítky nominovaných, konečné slovo budou mít opět zastupitelé.

Do čtvrtého ročníku Ceny města Příbrami veřejnost nominovala dvacet osobností spjatých s naším městem napříč všemi sférami. Porota v letošním roce vybrala osobnost z oblasti zdravotnictví, v pondělí 22. dubna budou tuto nominaci schvalovat zastupitelé.

„V seznamu nominovaných jsou jak jména, která se každoročně opakují, tak jména, která se objevují nově. Letos byl seznam pestrý – jsou v něm osobnosti z oblasti sportu, zdravotnictví, kultury, školství, politiky a dalších. Máme tu čest zasedat v porotě, která diskutuje nad každým jménem v seznamu. V letošním

[Domů](#) [Akce](#) [Parkování](#) [Více](#)

■ Obrázek A.3 Domácí obrazovka



Při čtvrtém ročníku putuje
ocenění do oblasti zdravotnictví

Čt 18. 4. 2024

Již počtvrté se v Příbrami bude předávat Cena města Příbrami. Porota vybírala z dvacítky nominovaných, konečné slovo budou mít opět zastupitelé.

Do čtvrtého ročníku Ceny města Příbrami veřejnost nominovala dvacet osobností spjatých s naším městem napříč všemi sférami. Porota v letošním roce vybrala osobnost z oblasti zdravotnictví, v pondělí 22. dubna budou tuto nominaci schvalovat zastupitelé.

„V seznamu nominovaných jsou jak jména, která se každoročně opakují, tak jména, která se objevují nově. Letos byl seznam pestrý – jsou v něm osobnosti z oblasti sportu, zdravotnictví, kultury, školství, politiky a dalších. Máme tu čest zasedat v porotě, která diskutuje nad každým jménem v seznamu. V letošním

[Domů](#) [Akce](#) [Parkování](#) [Více](#)

■ Obrázek A.4 Domácí obrazovka

21:34 97%

Dnes Zítra

Od: Po 22. 4. 2024 do: Po 22. 4. 2024

Divadlo

Válka s mláky
Po 22. 4. 2024

Ostatní

Zasedání Zastupitelstva města Příbram
Po 22. 4. 2024

Uranový důl Bytíz - nová pobočka Hornického muzea Příbram
Út 3. 10. 2023
Do: Út 31. 12. 2024

celodenní akce

Zasedání Zastupitelstva města Příbram
Po 22. 4. 2024

Ondřej Dvořák: Akt ženy
Út 2. 4. 2024

Výstava
Čt 21. 3. :

Pro děti

DN 7EMF

Domů Akce Parkování Více

■ **Obrázek A.5** Obrazovka událostí

21:34 97%

←

Uranový důl Bytíz - nová pobočka Hornického muzea Příbram

Od: Út 3. 10. 2023
Do: Út 31. 12. 2024

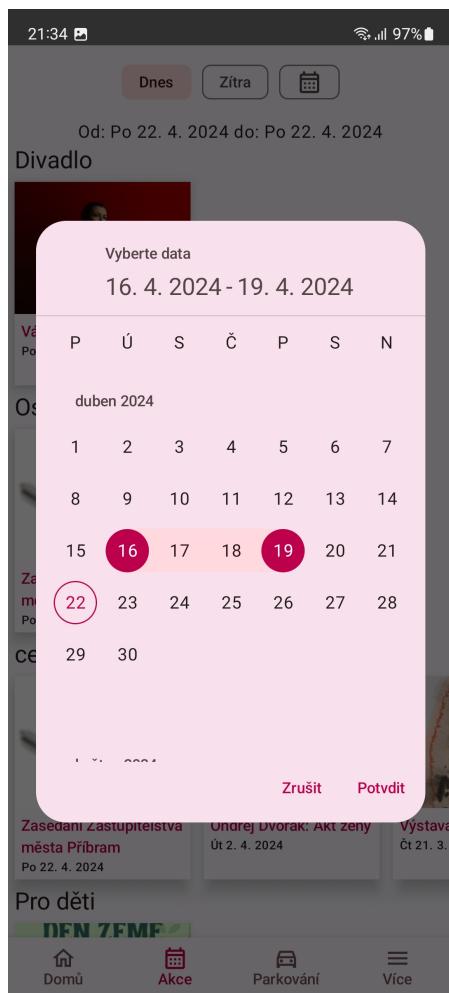
Hornické muzeum Příbram se řadí k největším hornickým muzeím v ČR i v Evropě. Spravuje více než 50 objektů, ke zhlédnutí nabízí přes 70 expozic a několik kilometrů historického podzemí zdejších dolů.

Provozuje též 5 poboček v okolí Příbrami. **Nejnovější z nich se stal Uranový důl Bytíz v areálu bývalé šachty č. 11 A z roku 1955**, kde se vedle těžní věže nachází strojovna, turbokompresorovna, elektrorozvodna, pozůstatky chladící věže a také hodnotné technické vybavení, jediné dochované v ČR.

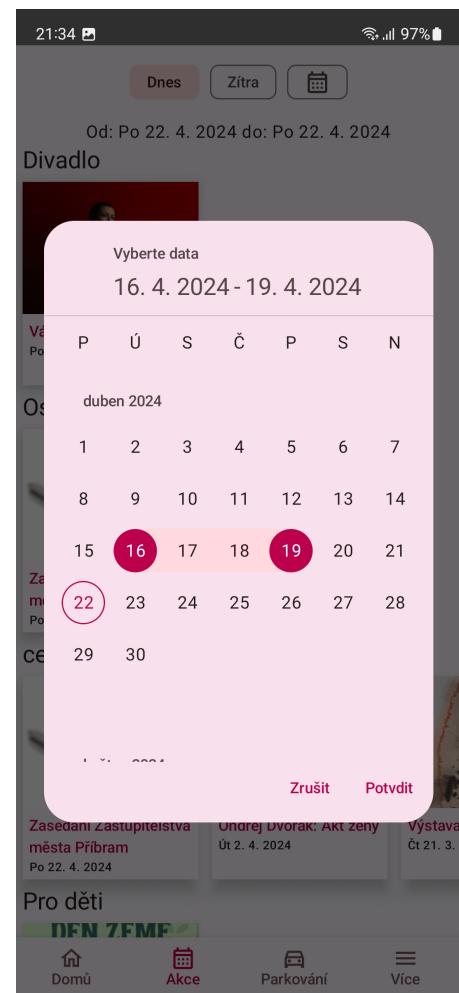
Od konce roku 2017 se muzeum snaží s podporou Středočeského kraje o **záchranu a proměnu tohoto jedinečného průmyslového areálu mezinárodního významu**, jehož objekty však byly ve velmi špatném stavu, **v areál návštěvnický**. Vzniknou tu expozice ve

Domů Akce Parkování Více

■ **Obrázek A.6** Obrazovka událostí



Obrázek A.7 Obrazovka událostí



Obrázek A.8 Obrazovka událostí



Obrázek A.9 Obrazovka parkování

Dvořákovo nábřeží [PB9]

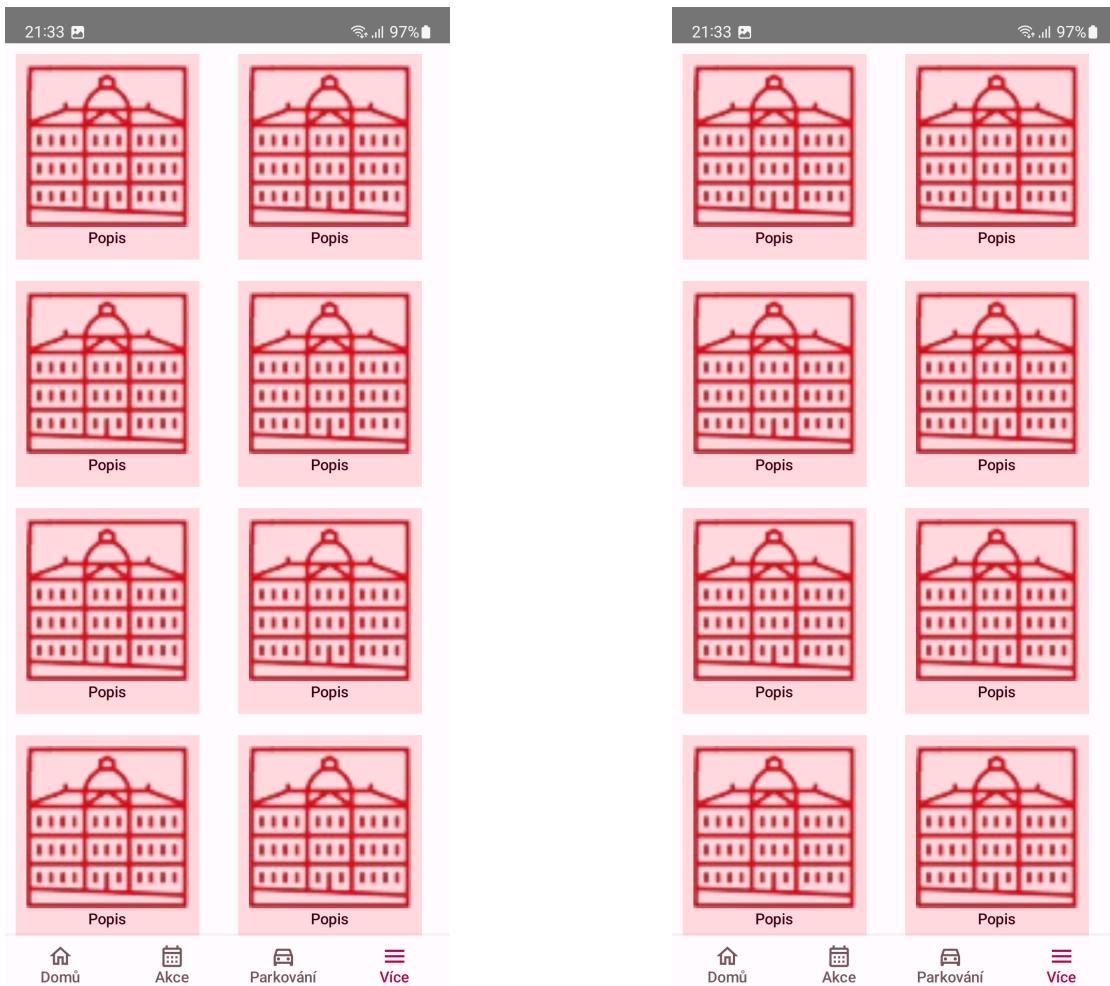
Platba: <http://parkovani.pribram.eu/PB9>

Cena: 20 Kč za hodinu.
Provozní doba: po - pá od 7 do 18 hodin, v So od 7 do 12 hodin.

Pozn. rezidenti si mohou zakoupit parkovací kartu za 200 korun za měsíc a za 600 korun měsíčně je dostupná pro právnické a podnikající fyzické osoby.

Zaplatit

Obrázek A.10 Obrazovka parkování



Obrázek A.11 Obrazovka parkování více

Sem přijde to, co nepatří do hlavní části.

Obrázek A.12 Obrazovka parkování 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. *Render, Commit, and Mount*. 2023. Dostupné také z: <https://reactnative.dev/architecture/render-pipeline>. Accessed: 28.12.2023.
12. TEAM, Flutter. *Flutter FAQ*. 2023. Dostupné také z: <https://docs.flutter.dev/resources/faq>. Accessed: 28.12.2023.
13. TEAM, Flutter. *Reactive user interfaces*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#reactive-user-interfaces>. Accessed: 28.2.2024.
14. TEAM, Flutter. *Widgets*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#widgets>. Accessed: 28.2.2024.
15. TEAM, Flutter. *Hot Reload*. 2024. Dostupné také z: <https://docs.flutter.dev/tools/hot-reload>. Accessed: 28.2.2024.

16. TEAM, Flutter. *Flutter architectural overview*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview>. Accessed: 28.12.2023.
17. TEAM, Flutter. *Flutter*. 2024. Dostupné také z: <https://flutter.dev>. Accessed: 28.2.2024.
18. TEAM, Flutter. *Widget catalog*. 2024. Dostupné také z: <https://docs.flutter.dev/ui/widgets>. Accessed: 28.2.2024.
19. Dart. 2024. Dostupné také z: <https://dart.dev/>. Accessed: 28.2.2024.
20. TEAM, Flutter. *Flutter's rendering model*. 2024. Dostupné také z: <https://docs.flutter.dev/resources/architectural-overview#flutters-rendering-model>. Accessed: 28.12.2023.
21. JETBRAINS. *Compose Multiplatform*. 2024. Dostupné také z: <https://www.jetbrains.com/lp/compose-multiplatform/>. Accessed: 10.1.2024.
22. *Build better apps faster with Jetpack Compose*. 2024. Dostupné také z: <https://developer.android.com/jetpack/compose>. Accessed: 13.1.2024.
23. *Kotlin Multiplatform use cases*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform.html#android-and-ios-applications>. Accessed: 1.2.2024.
24. *Kotlin Multiplatform*. 2024. Dostupné také z: <https://kotlinlang.org/docs/cross-platform-frameworks.html#kotlin-multiplatform>. Accessed: 1.2.2024.
25. *Jetpack Compose phases*. 2024. Dostupné také z: <https://developer.android.com/develop/ui/compose/phases>. Accessed: 13.1.2024.
26. 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.
27. *Expected and actual declarations*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-expect-actual.html>. Accessed: 15.3.2024.
28. *Common code*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-discover-project.html#common-code>. Accessed: 2.4.2024.
29. *Kotlin Native*. 2024. Dostupné také z: <https://kotlinlang.org/docs/native-overview.html>. Accessed: 2.4.2024.
30. *Hierarchical project structure*. 2024. Dostupné také z: <https://kotlinlang.org/docs/multiplatform-hierarchy.html>. Accessed: 2.4.2024.
31. *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.
32. *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.
33. 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.
34. 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.
35. *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.
36. GOOGLE. *Guide to app architecture*. 2024. Dostupné také z: [url%20=%20%7Bhttps://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/%7D](https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/). Accessed: 15.3.2024.

37. JETBRAINS. *Navigation and routing*. 2023. Dostupné také z: [url%20=%20%7Bhttps://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html%7D](https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html). Accessed: 15.3.2024.
38. GOOGLE. *Typography*. 2024. Dostupné také z: [url%20=%20%7Bhttps://m3.material.io/styles/typography/type-scale-tokens%7D](https://m3.material.io/styles/typography/type-scale-tokens). Accessed: 15.3.2024.
39. CAFÉ, Adriel. *Overview*. 2024. Dostupné také z: [url%20=%20%7Bhttps://voyager.adriel.cafe/%7D](https://voyager.adriel.cafe/). Accessed: 15.3.2024.
40. CAFÉ, Adriel. *Tab navigation*. 2024. Dostupné také z: [url%20=%20%7Bhttps://voyager.adriel.cafe/navigation/tab-navigation%7D](https://voyager.adriel.cafe/navigation/tab-navigation). Accessed: 15.3.2024.
41. JETBRAINS. *What's new in Compose Multiplatform 1.6.0*. 2024. Dostupné také z: [url%20=%20%7Bhttps://www.jetbrains.com/help/kotlin-multiplatform-dev/whats-new-compose-1-6-0.html#across-platforms%7D](https://www.jetbrains.com/help/kotlin-multiplatform-dev/whats-new-compose-1-6-0.html#across-platforms). Accessed: 15.3.2024.
42. GOOGLE. *Testing your Compose layout*. 2024. Dostupné také z: [url%20=%20%7Bhttps://developer.android.com/develop/ui/compose/testing%7D](https://developer.android.com/develop/ui/compose/testing). Accessed: 15.3.2024.

Obsah příloh

```
readme.txt.....stručný popis obsahu média
├── exe.....adresář se spustitelnou formou implementace
└── src
    ├── impl.....zdrojové kódy implementace
    └── thesis.....zdrojová forma práce ve formátu LATEX
        └── text.....text práce
            └── thesis.pdf.....text práce ve formátu PDF
```