FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Vývoj mobilních aplikací pomocí technologie Kotlin Multiplatform
Student:	Bc. Matouš Dlabal
Vedoucí:	Ing. Dominik Veselý
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Prozkoumejte použití technologie Kotlin Multiplatform a její porovnání s ostatními multiplatformními technologiemi jako Flutter a React Native. Popište výhody i nevýhody těchto technologií. Na základě zjištění naimplementujte co nejvhodněji univerzální knihovnu, která pak slouží jako stavební kámen pro Android a iOS aplikaci. Nad touto knihovnou potom implementujte mobilní aplikaci pro platformu Android. Zadání této vzorové aplikace a iOS implementaci konzultujte s vedoucím práce.

- Prozkoumejte a popište technologii Kotlin Multiplatform
- Porovnejte jednotlivé multiplatformní technologie mezi sebou
- Implementujte univerzální Kotlin Multiplatform knihovnu pro iOS a Android
- Implementujte aplikaci pro platformu Android, která bude postavena nad zmíněnou knihovnou

---

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 14. října 2022 v Praze.

Diplomová práce

VÝVOJ MOBILNÍCH APLIKACÍ POMOCÍ TECHNOLOGIE KOTLIN MULTIPLATFORM

Bc. Matouš Dlabal

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Dominik Veselý
3. května 2023

Obsah

Seznam obrázků

Seznam výpisů kódu

Seznam tabulek

Chtěl bych poděkovat především vedoucímu své práce, Ing. Dominikovi Veselému, za cenné rady, věcné připomínky a zprostředkování konzultací s Ing. Janem Mottlem a Lukášem Hromadníkem, odborníky na systémy Android a iOS, kterým bych chtěl tímto také vyjádřit své díky. V neposlední řadě patří poděkování i mé rodině a přátelům, kteří mi byli během studia a psaní práce oporou.

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 3. května 2023 .....

Abstrakt

Tato práce se zaměřuje na novou technologii Kotlin Multiplatform a její unikátní přístup k vývoji multiplatformních mobilních aplikací v porovnání s frameworky Flutter a React Native. Práce porovnává výhody a nedostatky jednotlivých přístupů a hlouběji popisuje fungování této technologie, která obohacuje nativní vývoj o možnost sdílení části kódu mezi více platformami. Existujícím multiplatformním frameworkům se tak svým novým přístupem nesnaží konkurovat, spíše je doplňuje. Pro test a demonstraci využití této technologie byl vytvořen prototyp — mobilní aplikace pro systém Android. Významnou část aplikace však tvoří sdílený kód, který by bylo možné použít při implementaci iOS aplikace, což bylo úspěšně vyzkoušeno.

Klíčová slova multiplatformní, vývoj, mobilní aplikace, Kotlin Multiplatform, Flutter, React Native, Android, iOS

Abstract

The focus of this thesis is on the new Kotlin Multiplatform technology and its unique approach to multiplatform mobile development in comparison with the Flutter and React Native frameworks. It compares the advantages and shortcomings of each approach and further describes the inner workings of this technology, which extends native developments capabilities by allowing parts of the source code to be shared across multiple platforms. Its new approach does not attempt to compete with existing multiplatform frameworks. Rather, it complements them. To test and demonstrate the use of this technology, a prototype — mobile application for the Android system — has been developed. However, a significant part of the application consists of shared code that could be used when implementing the iOS application. This possibility has been successfully tested.

Keywords multiplatform, cross-platform, development, mobile application, Kotlin Multiplatform, Flutter, React Native, Android, iOS

Seznam zkratek

API	Application Programming Interface
GTFS	General Transit Feed Specification
IR	Intermediate representation (mezikód)
JVM	Java Virtual Machine
KMM	Kotlin Multiplatform Mobile
MVVM	Model–view–viewmodel
PID	Pražská integrovaná doprava
ROPID	Regionální organizátor Pražské integrované dopravy
SDK	Software development kit
UI	Uživatelské rozhraní
Wasm	WebAssembly

Kapitola 1
Úvod

Chytré mobilní telefony jsou v dnešní době využívány širokou veřejností pro nejrůznější aktivity. Doby, kdy telefony sloužily pouze pro telefonování, jsou už dávno pryč. Nyní slouží pro práci i zábavu, pro mladší i starší uživatele. Dají se na nich hrát hry, poslouchat hudba nebo podcasty, sledovat videa, vzdělávat se, tvořit a upravovat fotky a videa, číst zprávy, psát emaily, komunikovat pomocí nejrůznějších chatovacích aplikací — ať už textově, nebo pomocí videa —, spravovat kalendářové události, nastavovat budíky, spravovat finance, sázet a mnoho dalšího. Aby bylo možné mobilní telefony využívat pro tak širokou škálu činností, musí pro ně existovat příslušné aplikace. Ty jsou většinou zaměřené pouze na jednu konkrétní funkcionalitu a proto jich je obrovské množství.

Tyto aplikace je potřeba naprogramovat, což je v základu možné pouze pro konkrétní platformu. Problém nastává, když vývojáři chtějí, aby byla aplikace co nejrozšířenější, aby měla co nejvíce uživatelů. Ať už je to z důvodu maximalizace zisku (nejlepšími příklady jsou sociální sítě, které vydělávají na reklamách a tím pádem na množství uživatelů, nebo mobilní bankovnictví, jehož omezení na jediný operační systém by znamenalo ztrátu části zákazníků) nebo protože je ve veřejném zájmu, aby mohlo mít aplikaci nainstalovanou co nejvíce uživatelů (například aplikace Záchranka, která může pomoci zachránit lidský život), je potřeba vytvořit aplikaci, která bude dostupná na více platformách. V případě moderních mobilních telefonů se jedná především o operační systémy Android a iOS. Protože je vývoj dvou aplikací nákladnější a vyžaduje navíc nějakou koordinaci, není překvapivá snaha sjednotit část, nebo dokonce celý, proces vývoje dohromady.

V této práci se zaměříme na novou technologii Kotlin Multiplatform, která představuje unikátní pohled na řešení problému tvorby multiplatformních aplikací, a to nejen pro mobilní zařízení. Technologii porovnáme s dvěma, v dnešní době nejrozšířenějšími, frameworky pro multiplatformní vývoj, kterými jsou Flutter a React Native. Popíšeme jejich rozdílný přístup k vývoji aplikací pro více platforem a klady a zápory, které s sebou jejich rozdíly přináší. Hlouběji si představíme technologii Kotlin Multiplatform a popíšeme, jak funguje. Seznámíme se s jazykem Kotlin, jak je v něm možné programovat pro různé platformy, a jakým způsobem lze kód mezi platformami sdílet.

Jako demonstraci a test technologie vytvoříme prototyp, který se bude skládat ze společné části — tu bude možné využít jednak pro Android, jednak pro iOS aplikaci — a z Android aplikace, která bude obsahovat uživatelské rozhraní, které využije funkcionalitu společné části. Ověříme, že by bylo obdobně možné využít společnou část v aplikaci pro systém iOS, ale celou aplikaci implementovat nebudeme.

Prototyp bude zpracovávat otevřená data o pražské hromadné dopravě, která jsou veřejně dostupná pomocí webového API. Po získání dat je upraví, aby bylo možné uživateli zobrazit linky, které zastavují u zvolené zastávky a jejich průměrné intervaly odjezdu. Uživatel se tak

například dozví, že linka 18 odjíždí ze zastávky Hradčanská každých přibližně 7,5 minut. Dál prototyp umožní zvolit cílovou zastávku a zobrazit jen linky, které vybrané zastávky spojují. Postup implementace popíšeme, a zhodnotíme, jak se s technologií Kotlin Multiplatform pracovalo. Zdokumentujeme také případné problémy, na které během vývoje narazíme, a vysvětlíme jejich možná řešení.

Kapitola 2

Chytré telefony a jejich aplikace

V této kapitole si popíšeme mobilní aplikace a chytré telefony, na kterých běží. Krátce si představíme historii mobilních telefonů a jejich operačních systémů, mezi kterými v současnosti převládají Android a iOS. Tyto dva dominantní operační systémy si blíže představíme a stručně shrneme nejrůznější multiplatformní technologie, které se snažili sjednotit vývoj aplikací pro tyto platformy. Popíšeme si také přechod uživatelů z webových stránek k používání mobilních aplikací a možné příčiny tohoto trendu.

2.1 Historie chytrých telefonů

Pojďme se nyní krátce podívat na historii chytrých telefonů. Podle Oxfordského slovníku je chytrý telefon (smartphone) takový mobilní telefon, který má některé funkce počítače — umožňuje použití internetu, aplikací a podobně. První chytrý telefon vydala již v roce 1992 firma IBM. *Simon Personal Communicator* sice stále připomínal cihlu, byl už však vybavený dotykovým displejem a umožňoval komunikaci pomocí emailů a faxů. Většina následujících chytrých telefonů měla ještě dalších přibližně 15 let mechanickou klávesnici. Přestože nabízely na svou dobu široké spektrum funkcionalit, z dnešního pohledu, kdy si chytré telefony spojujeme s velkým dotykovým displejem a možností instalace aplikací od širokého spektra vývojářů, je poněkud úsměvné, považovat je za chytré telefony. Skutečný zlom nastal až v roce 2007, kdy firma Apple představila svůj první chytrý telefon — iPhone. [1, 2]

Šlo o telefon bez mechanické klávesnice, rozhodnutí, které se zprvu setkalo s nepochopením a pobouřením, ale v dlouhodobém horizontu změnilo celý trh. Oproti tomu první telefon se systémem Android — vydaný v roce 2008 — měl stále vysouvací mechanickou klávesnici a v porovnání s konkurenčním telefonem iPhone horší dotykovou obrazovku. Přestože Apple představil v mnoha ohledech inovativní zařízení, jednalo se častokrát o kvalitu provedení, ne samotnou myšlenku, která zajistila takový ohromný úspěch. Například možnost instalace aplikací třetích stran existovala i před představením *App Store*. Apple ale celý proces vyhledávání, placení a instalace výrazně zjednodušil a v roce 2008 představil druhý model — iPhone 3G —, který obsahoval do značné míry vše, co od chytrého telefonu čekáme i v dnešní době. Byl to telefon s pokročilým operačním systémem, možností instalace dalších aplikací, 3G konektivitou, fotoaparátem, GPS modulem, vestavěnými senzory a připojením k Wi-Fi nebo Bluetooth. [3, 4, 5]

Uživatelé začali iPhone, a následně další chytré telefony, používat například pro navštěvování webových stránek, poslech hudby nebo hraní her. Právě v oblasti her tak v podstatě vymizely přenosné herní konzole typu Nintendo nebo PlayStation Portable. V roce 2022 tvořily celosvětově mobilní hry přibližně polovinu všech herních tržeb. Tento fakt sám o sobě ukazuje, jak rozšířené se chytré telefony staly. [4, 6]

V současnosti jsou mezi operačními systémy jasnými favority Android a iOS, s celosvětovými podíly na trhu přes 72 % pro Android a 27 % pro iOS. Právě z toho důvodu se na ně zaměřuje většina frameworků pro multiplatformní tvorbu aplikací. Dává to smysl, protože tvorbou aplikací pro tyto dvě platformy jsou schopni pokrýt drtivou většinu trhu. Vynakládat úsilí pro podporu dalších systémů, které netvoří ani celé jedno procento, by bylo kontraproduktivní. Tak tomu je i v případě technologie Kotlin Multiplatform a tedy i této práce. V počátcích vývoje moderních chytrých telefonů, kdy si Android a iOS ještě nedobyly své dominantní postavení, ale existovala celá řada operačních systémů, které si zde stručně shrneme. [7]

Dlouhou dobu dominoval operační systém Symbian, používaný především, ale rozhodně ne výhradně, značkou Nokia. Před představením telefonu iPhone měl v roce 2005 asi 67% podíl. Druhý nejrozšířenější byl se 14 % systém od společnosti Microsoft. Na třetím místě se umístila firma Research In Motion se systémem pro telefony BlackBerry. Nicméně díky velkému úspěchu telefonů firmy Apple se stal operační systém iOS na několik let podobně rozšířeným, jako Symbian (oba měly okolo 35 %), který později zaznamenal v roce 2012 hluboký pokles a byl nahrazen systémem Android. Právě ten se i přes opožděný start docela rychle ujal a od roku 2012, kdy měl na trhu zastoupení až 30 %, je nejrozšířenějším mobilním operačním systémem na světě. Kromě toho, že šlo o kvalitní operační systém svou roli určitě sehrálo i rozhodnutí nabídnout systém zadarmo. Kterýkoliv výrobce mohl začít dodávat telefony se systémem Android. [8, 7]

I přes nemalou snahu mnoha firem se žádnému dalšímu operačnímu systému pro chytré telefony nepodařilo získat výrazné zastoupení na trhu. Firma Microsoft, která měla před revolucí, kterou přinesl Apple v podobě telefonu iPhone, na celosvětovém trhu sice ne dominantní, ale přesto služné zastoupení, se v roce 2008 rozhodla upustit of svého systému Windows Mobile, který nebyl schopný Androidu a iOS konkurovat. Začala vyvíjet nový systém — Windows Phone —, který však mezi uživateli nezískal velkou podporu a nikdy nepřesáhl hranici 2, 55 %. [3]

Uplatnit se snažili například i firmy Samsung nebo LG, ale jejich systémy nakonec na mobilním trhu neuspěly. Místo toho našly využití pro chytré televize. Firma LG má se systémem WebOS a svými skoro 16 % (k roku 2020) podobně velký segment trhu, jako Google se systémem Android TV. Operační systém Tizen, který Samsung používá i pro chytré telefony a další chytrá, přenosná zařízení, dokonce na poli chytrých televizních systémů s necelými 35 % vede. [9, 10]

Je vidět, že mobilní trh byl dříve mnohem více roztříštěný, což vývoj mobilních aplikací značně znesnadňovalo. Vývoj pro většinu používaných platforem je, jak jsme již popsali v úvodu této práce, pro firmy většinou velice důležitý, což znamená, že je dnešní dualita mobilních systémů pro vývojáře výhodná. Podpora většího množství platforem by znamenala výrazně nákladnější nativní vývoj, ale zároveň také složitější situaci s multiplatformními frameworky.

2.2 Android

Android je mobilní operační systém s otevřeným zdrojovým kódem založený na upraveném Linuxovém jádru. Vytvořila ho skupina technologických firem Open Handset Alliance, jejíž součástí byly například společnosti Samsung, HTC nebo T-mobile, ale nejvíce se na jeho vývoji podílí společnost Google. V době svého vzniku měl velké konkurenty v podobě systému pro telefony BlackBerry, systému Symbian a později také iOS, nicméně od roku 2012 je nejrozšířenějším operačním systémem pro chytré telefony. Android je nainstalovaný na široké škále zařízení — z výkonnostního i cenového pohledu — různých výrobců, čímž se výrazně odlišuje od systému iOS, který je dostupný pouze na telefonech od firmy Apple. [11]

První verze Androidu vyšla na konci září 2008. Následující verze byly vydávány s rozestupem maximálně jednoho roku. Většina verzí systému (až do verze 10) má kromě číselného pojmenování i název podle různých sladkostí. Například Android 5.0 Lollipop — první verze s novým vzhledem (Material Design). Aktuální verzí je Android 13. [12]

Aplikace pro Android se historicky vyvíjely v programovacím jazyku Java. V současnosti se sice přechází k používání jazyka Kotlin, ale princip, jakým operační systém aplikace spouští, je stále stejný — sestavený Kotlin kód je také reprezentován v bytekódu. Stejně jako v případě

operačních systémů pro stolní počítače běží na Android telefonech virtuální stroj, který Java bytekód interpretuje. Od Android verze 4.4 byl virtuální stroj Dalvik, který před každým spuštěním aplikace prováděl *Just-In-Time* kompilaci, nahrazen. Nyní se při instalaci aplikace provede *Ahed-of-Time* kompilace, díky čemuž je následné spouštění aplikace rychlejší. [11]

Přestože je Android založený na Linuxu, vývojáři k němu nemají přístup. Jádro tvoří v celkové architektuře Androidu tu nejspodnější vrstvu, nad kterou se nachází nejen nativní knihovny, ale i *Android Runtime*, který obsahuje jednak již zmíněný virtuální stroj, jednak základní Android knihovny, s kterými pracují vývojáři. Pro docílení podpory napříč všemi fyzickými zařízeními, na kterých operační systém běží, se aplikace spouštějí právě ve virtuálním stroji. [13]

Běh jednotlivých aplikací je od sebe v Androidu oddělen. Každá má vlastní sandbox, který jí zabraňuje zasahovat do ostatních aplikací a omezuje, do jaké míry může komunikovat s operačním systémem. Pro jednotlivé typy akcí potřebuje mít přidělená oprávnění, například pro přístup k souborovému systému nebo fotoaparátu. Tato oprávnění uživatelé v dřívějších verzích schvalovali hromadně při instalaci aplikace, nyní si o ně může aplikace žádat jednotlivě a během běhu aplikace. Je tedy možné, že dokud nebudeme chtít používat funkcionalitu, která je závislá na některém z oprávnění, aplikace nás o něj vůbec nebude žádat. V nastavení je navíc vždy možné oprávnění odebrat. [14]

Koncept aplikace bývá z uživatelského hlediska svázán ikonou na hlavní obrazovce a uživatelským rozhraním, které se po spuštění zobrazí. Ne všechny Android aplikace jsou však takto viditelné. Ty, které uživatelské rozhraní mají a uživatel s nimi díky tomu může interagovat, se nazývají aktivity. Dalšími typy jsou servisy, které slouží k dlouhodobějšímu běhu, například pro kontrolování aktualizací. Dále se rozlišují takzvaní Content providers, kteří zprostředkovávají nějaká data a Broadcast receivers, kteří reagují na přijaté události. Všechny tyto typy jsou závislé na systému Android a z pohledu multiplatformního vývoje je problematické jejich kód mezi platformami sdílet. Systém iOS například nenabízí možnost dlouhodobého běhu aplikace na pozadí. [11, 15]

2.3 iOS

Mobilní operační systém iOS je určen pro chytré telefony iPhone (původně se jmenoval iPhone OS), ale dlouho dobu také sloužil například pro tablety iPad, které mají od roku 2019 vlastní, odvozený systém iPadOS. Operační systém iOS byl vyvinut společností Apple a stejně jako systém macOS (pro stolní počítače) je založený na systému UNIX. V době psaní práce je nejnovější verzí iOS 16. Výrazným rozdílem, oproti konkurenčnímu systému Android, je omezené množství chytrých telefonů s kterými je systém dodávaný. Apple systém dodává pouze na svých zařízeních a na rozdíl od Androidu na nich aplikace běží nativně, ne ve virtuálním stroji. [13, 16]

Aplikace lze pro iOS vyvíjet v programovacím jazyku Objective-C, nebo v novějším jazyku Swift. Oba dva jazyky jsou kompilované a objektově orientované. Swift byl vydán v roce 2014 společností Apple, jako nástupce Objective-C. Jde o novější jazyk, navržený pro tvorbu přehlednějšího a bezpečnějšího kódu. [13]

V počátku Apple neměl v plánu nechat uživatele instalovat aplikace od vývojářů třetích stran. První vydaný iPhone neobsahoval App Store, ze kterého by bylo možné aplikace instalovat, a společnost ani nevydala žádné iOS SDK — balíček nástrojů pro vývoj. Původním záměrem bylo využití webového prohlížeče Safari, ve kterém by běžely webové aplikace, nicméně toto řešení se nesetkalo s pozitivním ohlasem, a tak společnost zpřístupnila nástroje pro tvorbu aplikací do druhého modelu iPhone přidala aplikaci App Store, pomocí které mohou uživatelé aplikace vyhledávat a instalovat. Všechny aplikace, které se v App Store objeví, musí být nejdříve schváleny společností Apple, která si strhne 30 % z výnosu z prodeje aplikace. [17, 13]

2.4 Uživatelské preference

S rozšířením rychlejšího a levnějšího přístupu k internetu začali uživatelé čím dál více používat chytré telefony k prohlížení webových stránek. Oproti roku 2011, kdy načtených webových stránek pomocí mobilních telefonů bylo pouze $7,2\,\%$, se v dnešní době k webovým stránkám přistupuje většinou právě z mobilních telefonů. Tato skutečnost změnila pohled na vývoj webových stránek, které nejen že jsou v dnešní době tvořené tak, aby se dobře zobrazovaly na mobilních telefonech — byly responzivní —, ale vyvíjejí se častokrát primárně pro mobilní zařízení.

Pokud webové stránky nabízí uživatelům nějakou rozsáhlejší funkcionalitu, než prohlížení více méně statických stránek, mají častokrát alternativu v podobě mobilní aplikace. Můžeme se s tím setkat například u internetových obchodů, sociálních sítí, zpravodajství nebo třeba mapových aplikací. Podle statistik z počátku roku 2022 uživatelé tráví při používání chytrých telefonů naprostou většinu času ($92,5\,\%$) v mobilních aplikacích a pouze $7,5\,\%$ používáním webového prohlížeče. [18]

Oblíbenost aplikací oproti webovým stránkám není nikterak překvapující. Aplikace představují nativní způsob, jak s telefonem pracovat. Oproti tomu běh webových aplikací zprostředkovává prohlížeč, což znamená i pro nejlépe realizované webové stránky jistá omezení. Nemohou například využívat všechny funkcionality mobilního telefonu, jako jsou nejrůznější senzory, nebo fotoaparát.

Aplikace mají také větší možnosti s ukládáním dat a pamatováním si stavu v jakém se aplikace nachází. Už jen přihlašování k uživatelskému účtu je něco, co v případě používání webové stránky musíme častokrát opakovat. V aplikaci se do účtu přihlásíme jednou a dokud je nainstalovaná, není nutné tento krok opakovat. Schopnost pracovat s lokální pamětí telefonu navíc nabízí potenciál pro použití aplikace bez připojení k internetu. Podcastové aplikace nabízí možnost stáhnout nové episody, aby si je mohl uživatel poslechnout i offline. Obdobnou funkcionalitu nabízí aplikace, které poskytují textový obsahu — například zpravodajské aplikace. Jízdní řády hromadné dopravy se také nemění tak často, aby bylo nutné je vždy načítat znovy a pokud máme v telefonu dostatek místa, můžeme si dokonce stáhnout celé mapové podklady. Tak trochu skrytou výhodou při využití paměti je také rychlejší načítání, protože mobilní aplikace potřebuje získat jen ty podstatné údaje, zatímco webová musí stahovat i uživatelské rozhraní — rozložení stánky, ikonky, fonty, skripty apod.

Plné využití schopností telefonu přináší vývojářům možnost rozšíření funkcionality aplikací. Například použití fotoaparátu, i kdyby se jednalo jen o načtení QR kódu může uživateli zpříjemnit použití aplikace. V případě sociálních sítí, které jsou na sdílení foto a video obsahu založené, je přístup k fotoaparátu, ale také mikrofonu nebo seznamu kontaktů, zásadní.

Vyšší používání mobilních aplikací oproti webovým stránkám však může být částečně způsobeno také vývojáři. Pro firmy je určitě žádoucí, aby měli uživatelé nainstalovanou jejich aplikaci, protože může částečně fungovat, jako odlehčená věrnostní karta. Ano, ve většině případů nenabízí žádné výhody, jako tomu je u oněch karet, ale z vlastní zkušenosti vím, že opakovaný pohled na ikonu aplikace v telefonu, mi neustále připomíná například určitý internetový obchod. Ve chvíli, kdy potřebuji něco nakoupit, spíše využiji již nainstalovanou aplikaci, než abych hledal a stahoval novou, nebo zkusil použít většinou méně kvalitní webovou stránku.

Pokud bych porovnávali kvalitu, mají aplikace nad webem opět náskok. Oproti webu, který musí fungovat i na větších obrazovkách stolních počítačů, jsou aplikace cílené pouze na mobilní telefony. Vývojáři mohou při vývoji použít nativní prvky platformy a vytvořit aplikaci, ve které se budou uživatelé lépe orientovat a bude se jim lépe ovládat. I pokud se pro toto řešení nerozhodnou a pokusí se co nejvíce zachovat vzhled webové stránky, bude aplikace optimalizovaná na ovládání dotykem.

Toto porovnání mobilních a webových aplikací nám ukázalo, že ačkoliv mobilní aplikace jdou nahradit těmi webovými, mají nezanedbatelné výhody. Pokud si vývoj nemůžeme usnadnit webovými aplikacemi, nezbývá nám, než opět řešit problém s více mobilními platformami.

2.5 Vývoj multiplatformních aplikací

Již několikrát jsme zmiňovali multiplatformní aplikace a potřebu jejich vývoje. Pojďme si tedy v rychlosti popsat typy aplikací a některé technologie, které multiplatformní vývoj umožňují.

2.5.1 Rozdělení aplikací

Mobilní aplikace se dají rozdělit podle různých kritérií do několika typů. Podle typu vývoje je lze rozdělit na nativní a multiplatformní.

2.5.1.1 Podle vývoje

Nativní aplikace V tomto případě je slovem nativní myšlen nativní vývoj, tedy používání oficiálních nástrojů, programovacího jazyku, SDK a knihoven dodávaných společností, která daný operační systém vyvíjí. Pro Android je to jazyk Kotlin, respektive Java. Aplikace pro systém iOS se vyvíjejí v jazycích Swift nebo Objective-C. [19]

Multiplatformní aplikace V případě multiplatformních aplikací se při vývoji využívají frameworky a nástroje, které nejsou oficiálně podporované, a je možné (a časté), že se pro implementaci využívají jiné programovací jazyky. Zásadní je, že lze pomocí stejného kódu vytvořit aplikaci pro několik platforem. Jde o velmi široký pojem, který zahrnuje všechny multiplatformní frameworky bez ohledu na způsob, jakým fungují. Z toho důvodu se aplikace dají dělit ještě podle jiného kritéria — jejich realizace. [19]

2.5.1.2 Podle realizace

Hybridní aplikace Teto typ aplikací je založený na webových technologiích a jejich běh v podstatě zajišťuje webový prohlížeč uvnitř aplikace. Příkladem frameworků, které umožňují vývoj hybridních aplikací, jsou *Apache Cordova* nebo *Ionic*. [19]

Nativně běžící aplikace Ve zbylých případech, kdy se kód aplikace, aby mohl na zařízení běžet, nějakým způsobem kompiluje a pro uživatelské rozhraní framework buť využívá nativní grafické prvky, nebo ho vykresluje pomocí vlastního renderovacího enginu, se aplikace považuje za nativně běžící. Jde o frameworky *Xamarin*, *Flutter*, *React Native* a další. [19]

2.5.2 Stručný přehled frameworků

2.5.2.1 Flutter

Tvůrce Google

Rok vydání 2018

Podporované platformy Android, iOS, web, desktop, vestavěné systémy

Programovací jazyk Dart

Look and feel jednotný vzhled, možnost napodobení nativních prvků

UI framework vlastní

Typ aplikace nativně běžící

Popis Flutter je spolu s frameworkem React Native jeden z nepoužívanějších nástrojů pro tvorbu mobilních aplikací. Používá vlastní renderovací engine, což znamená, že aplikace vypadají na všech platformách stejně. [20, 21]

2.5.2.2 React Native

Tvůrce Meta

Rok vydání 2015

Podporované platformy Android, iOS, web, desktop, TV, VR

Programovací jazyk JavaScript

Look and feel nativní

UI framework React

Typ aplikace nativně běžící

Popis Rozšiřuje knihovnu React o použití pro mobilní zařízení a mnoho dalších platforem. Deklarované grafické komponenty se vykreslí pomocí nativního API. Díky své popularitě má velkou komunitu, která sdílí své znalosti a zkušenosti s frameworkem. [22, 23]

2.5.2.3 Xamarin

Tvůrce Microsoft

Rok vydání 2011

Podporované platformy Android, iOS, Windows

Programovací jazyk C#

Look and feel nativní

UI framework vlastní

Typ aplikace nativně běžící

Popis Umožňuje použít jazyk C# pro implementaci multiplatformních aplikací, které využívají nativní grafické prvky. [22, 24]

2.5.2.4 Apache Cordova (dříve PhoneGap)

Tvůrce Nitobi, Apache Software Foundation

Rok vydání 2009

Podporované platformy Android, iOS, Windows

Programovací jazyk HTML, CSS, JavaScript

Look and feel jednotný vzhled

UI framework žádný, nebo možnost použít React, Ratchet a další

Typ aplikace hybridní

Popis Framework pro tvorbu hybridních aplikací pomocí webových technologií. Nabízí pluginy pro přístup k nativním API pro využití senzorů, GPS nebo kamery. [25, 24]

2.5.2.5 Ionic

Tvůrce Ionic

Rok vydání 2013

Podporované platformy Android, iOS, Windows

Programovací jazyk HTML, CSS, JavaScript

Look and feel napodobuje nativní

UI framework React, Anguler, Vue

Typ aplikace hybridní

Popis Umožňuje výběr UI frameworku, který pro vývoj hybridní aplikace použijeme. Pro přístup k nativnímu rozhraní používá Cordova pluginy. [26, 22, 24]

2.5.2.6 NativeScript

Tvůrce Progress, OpenJS Foundation

Rok vydání 2014

Podporované platformy Android, iOS

Programovací jazyk JavaScript, TypeScript

Look and feel nativní

UI framework Angular, Vue

Typ aplikace nativně běžící

Popis Pomocí frameworku NativeScript se dají tvořit aplikace, které využívají nativní grafické prvky, přestože se při vývoji používají webové technologie. [27, 22, 24]

Kapitola 3

Frameworky pro tvorbu multiplatformních aplikací

V této kapitole si představíme vybrané technologie pro tvorbu multiplatformních aplikací — Flutter, React Native a Kotlin Multiplatform — a popíšeme si jejich výhody a nevýhody. Tyto frameworky často umožňují multiplatformní vývoj nejen pro mobilní zařízení, ale i pro web a desktop. My se zaměříme především na vývoj aplikací pro nejrozšířenější mobilní operační systémy — Android a iOS. [7]

3.1 Flutter

Framework Flutter umožňuje multiplatformní vývoj aplikací pro mobilní zařízení, web, desktop i vestavěné systémy. Je vyvíjen jako open source projekt společností Google, která jeho první stabilní verzi vydala v roce 2018. Framework umožňuje psát kód (v programovacím jazyku Dart), který je pro všechny výše zmíněné platformy stejný. Flutter je nejpoužívanějším mobilním multiplatformním frameworkem. Mezi společnosti, které ho používají patří Google, Alibaba, Baidu, eBay, Toyota nebo BMW. [20, 21, 28, 29]

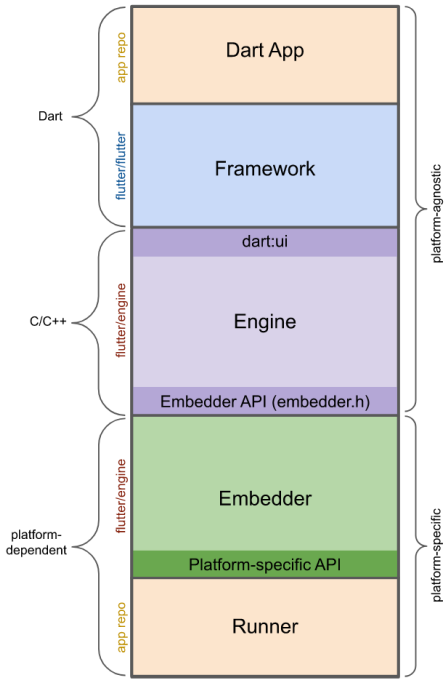
Flutter se od mnoha ostatních frameworků odlišuje tím, že využívá vlastní renderovací engine, což má za následek, že aplikace vypadá na všech platformách stejně. Tento engine je naprogramovaný v jazycích C a C++, kompletně řeší vykreslování aplikace a zaznamenávání a delegaci vstupů od uživatele. Ještě na nižší úrovni obsahuje Flutter vrstvu Embedder, která se stará o komunikaci s operačním systémem. Zajišťuje například práci s vlákny, nebo přístup k vykreslovací oblasti. Embedder je závislý na konkrétní platformně, zatímco Engine, samotný framework a aplikace jsou platformě agnostické (viz obrázek 3.1), díky čemuž je právě možné jednotně vyvíjet aplikace pro Android i iOS. [21, 30]

3.1.1 Widgety

Samotný framework, který je naprogramovaný v jazyku Dart, je ta část Flutteru, s kterou komunikuje vývojář aplikace. Jeho významnou částí jsou *widgety* — vizuální prvky, které reprezentují veškerá tlačítka, ikony, text nebo jejich seskupení, které se zobrazí na obrazovce. Jde o třídy pro tvorbu uživatelského rozhraní, které složené dohromady tvoří stromovitou strukturu. Pro navigaci mezi různými obrazovkami — v Androidu by mohlo jít o změnu Aktivity — zajišťuje framework výběrem stromu widgetů, který se má právě vykreslovat. [30]

Flutter obsahuje dva druhy widgetů, které se odlišují tím, jestli na sebe mají navázaný nějaký stav. Příkladem stateless widgetů, které stav nemají, je textový popis, ikona nebo ListView.

Obrázek 3.1 Anatomie Flutter aplikace [31]



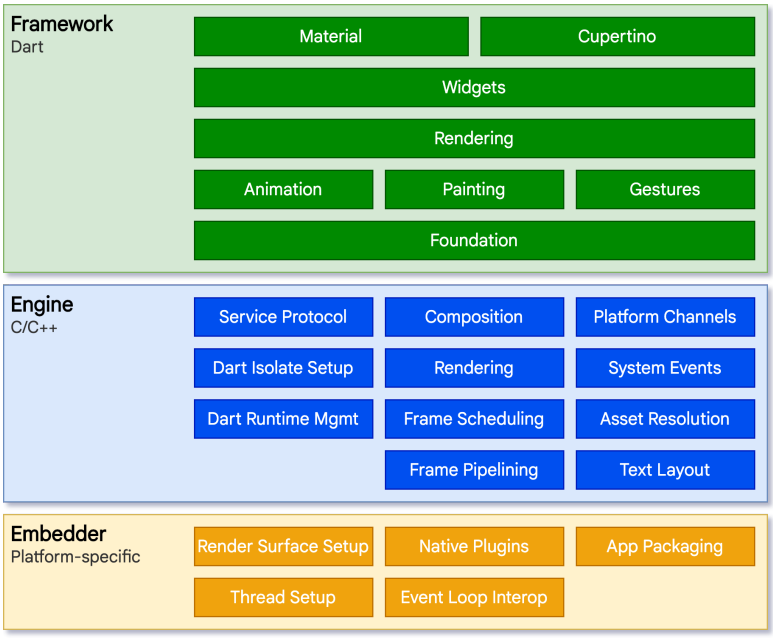
Stateful widgety, například textová nebo zaškrtávací pole, k sobě vážou stav. Všechny widgety jsou immutabilní a pokud mají stav, uchovávají ho v samostatném objektu typu `State`. Díky tomu se můžeme chovat k oběma druhům stejně — v případě vytvoření nové instance se o provázání se správným stavem postará framework. [30]

Toto rozdělení hraje roli při vykreslování obrazovky. Flutter totiž prochází stromovou hierarchii složených widgetů a voláním jejich metody `build()` je přeměňuje na elementy, které konkrétněji reprezentují, jak bude celá struktura vypadat. Pro optimalizování tohoto procesu se framework snaží co nejvíce využít již vytvořené elementy. To je však možné pouze v situaci, kdy skutečně nedošlo k žádné změně nejen u konkrétního widgetu, ale ani u žádného z jeho potomků. V případě, že se stavu widgetu změnil, je na to potřeba framework upozornit zavoláním metody `setState()`. [30]

3.1.2 Vzhled aplikací

Jak jsme již popsali, Flutter se stará o vykreslování veškerých grafických prvků sám. Využívá k tomu vlastní engine a zcela obchází jakékoliv nativní nástroje, které pro vykreslování používají nativní aplikace, ale například i framework React Native. Flutter nabízí obdobné vizuální prvky, jako jsou tlačítka, přepínače, nebo selektor data a času. Tyto prvky mimo jiné obsahuje v Material Design (podle Androidu) a Cupertino (podle iOS) variantě. Je tedy možné použít vzhledově stejné widgety, jako kdybychom vytvářeli aplikaci nativně, ale ve skutečnosti jde o napodobeniny, které vykresluje engine frameworku Flutter. Technicky je tedy možné za běhu zkontrolovat, na jaké platformě aplikace běží a zobrazit podle toho odpovídající uživatelské rozhraní, podle autorů Flutter dokumentace se to však děje zřídka kdy. Častější praktikou je podle nich vytváření jednotného designu, napříč platformami. [21]

Obrázek 3.2 Diagram Flutter architektury [32]



3.2 React Native

React Native je open source framework vydaný v roce 2015, jehož vývoj zaštiťuje společnost Facebook. S využitím programovacího jazyku JavaScript se pomocí něj dají tvořit multiplatformní aplikace pro Android, iOS, web, desktop, TV, VR a další. React Native je založený na knihovně React, která slouží k tvorbě webových uživatelských rozhraní, a rozšiřuje možnosti jejího použití přidáním podpory pro další platformy. Na každé platformě se jednotlivé prvky uživatelského rozhraní reprezentují nativními elementy, takže konkrétní vzhled aplikace závisí na cílové platformě. V roce 2021 byl React Native druhý nejpoužívanější framework pro vývoj multiplatformních aplikací. Společnosti, které React Native používají, jsou například Facebook, Microsoft, Tesla, Pinterest nebo Discord. [33, 23, 28]

Uživatelské rozhraní se pomocí knihovny React tvoří deklarativně, skládáním komponent. React tento princip tvorby UI zpopularizoval a v tuto chvíli se používá i při nativním vývoji pro Android (Jetpack Compose) nebo iOS (SwiftUI). Výše popsaný framework Flutter definuje uživatelské rozhraní také deklarativně. [23]

3.2.1 Komponenty

React a tedy i framework React Native skládají strukturu uživatelského rozhraní z komponent, což jsou funkce, respektive třídy s metodou `render()`. Novější přístup je používání funkcí, ale zatím jsou dostupné obě možnosti. Tyto funkce vrací opět komponenty, pro jejichž zápis se používá JSX. JSX (JavaScript Syntax Extension) je rozšíření syntaxe jazyka JavaScript, které umožňuje vytvářet stromovou strukturu elementů pomocí syntaxe, která se podobá HTML. Jednotlivé komponenty, kterým se mohou také předávat parametry, se touto syntaxí do sebe velice přehledně zanořují. Parametry (props) se používají k nastavení neměnných vlastností komponenty. Pro stav, který se může měnit, se používají proměnné, jejichž změna vyvolá překreslení komponenty. Tyto proměnné k sobě váží setter — funkci, pomocí které se nastaví nová hodnota, a která následně vyvolá znovuvytvoření komponenty (zavolání její funkce). [34]

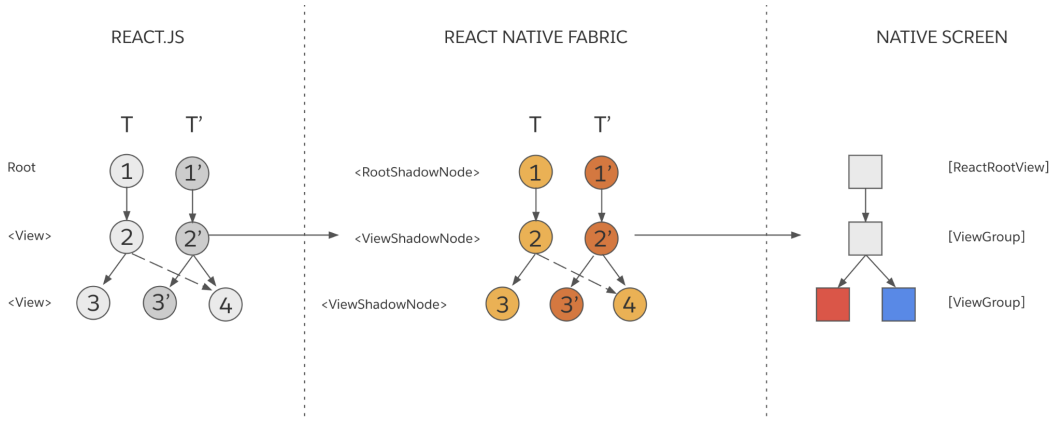
Framework React Native za běhu komunikuje s nativním API platformy a používá ho k zobrazení komponent pomocí nativní grafických prvků. Pokud tedy použijeme komponentu `TextInput`, na Androidu se zobrazí `EditText`, zatímco na iOS `UITextField`. Těmto komponentám se říká *Native Components* a těm nejpoužívanějším, které jsou obsažené přímo ve frameworku, *Core Components*. Další komponenty se dají získat jako rozšíření od komunity nebo je možné si vytvořit vlastní komponenty. [35]

Zajímavé je, že pro vytvoření aplikace s více obrazovkami je namísto řešení dodáveného s frameworkem potřeba použít některou z knihoven. Může se jednat například o knihovnu React Navigation — ta je popsaná přímo v React Native dokumentaci —, ale možností existuje více. Princip je však podobný, jako v případě frameworku Flutter. Nepřepínáme sice přímo komponenty, nicméně definované obrazovky, které k tomu slouží, na sebe mají komponenty navázané. [36]

3.2.2 Architektura

Nyní si podrobněji popíšeme vnitřní fungování frameworku React Native. Běh JavaScriptového kódu v současnosti zajišťuje engine *Hermes*, který je určený a optimalizovaný přímo pro React Native. Dříve se používal engine *JavaScriptCore*, který také zajišťuje spouštění JavaScriptového kódu ve WebKitu — webovém enginu prohlížeče Safari. Další částí frameworku je *Fabric* — renderer, který zajišťuje vykreslování komponent na nativních platformách. Samotný React je určený pouze pro web, kde umí zobrazovat komponenty pomocí DOM uzlů. Fabric je tou částí frameworku React Native, která komunikuje s nativním API na platformách jako je Android nebo iOS. [37, 38]

Obrázek 3.3 Stromové struktury používané při vykreslování [39]

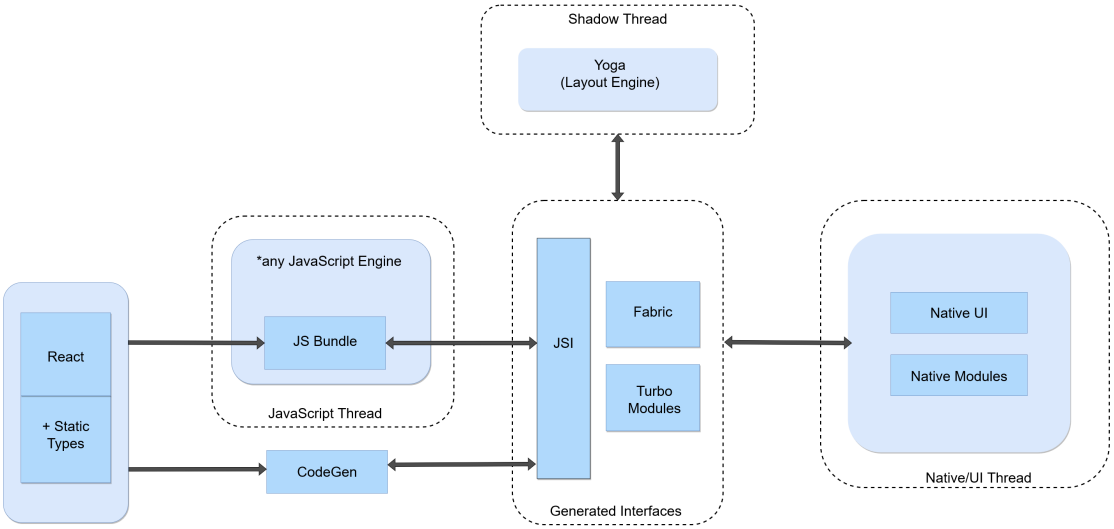


Celý proces vykreslování se dělí do tří částí — render, commit a mount. V první fázi se ještě spouští kódu v JavaScriptu, protože je potřeba z komponent sestavit *React Element Tree*. Tyto JavaSctipt objekty reprezentují, co se má zobrazit. Současně se vytvoří také *React Shadow Tree*, který už je reprezentován objekty v C++. Takto se předají informace rendereru Fabric, který stále pracuje nezávisle na platformě. Pro urychlení opakovaného renderování využívá React Native více vláken, a proto jsou obě stromové reprezentace prvků immutabilní a při změně se vždy tvoří (alespoň z části) nový strom. Z důvodu optimalizace se však mohou využít nezměněné větve, které jsou sdílené více stromy (viz obrázek 3.3). [40]

Ve druhé části vykreslování (commitu) se už pracuje pouze s C++ strukturou *React Shadow Tree*. Provede se výpočet velikosti prvků a jejich rozložení na obrazovce — k tomu se využívá knihovna Yoga — a nově vytvořený strom se označí jako příští k zobrazení. [40]

Třetí a poslední fáze se stará o převedení *React Shadow Tree* na *Host View Tree*, což už je struktura závislá na konkrétní platformně. Pro optimalizaci se nejdříve provádí zploštění stromu,

Obrázek 3.4 Diagram React Native architektury [42]



které má za cíl snížit jeho hloubku a tím pádem i počet uzlů k vykreslení. Prvky, které sami o sobě nic nevykreslují, ale pouze upravují rozložení ostatních prvků — potomků —, se sloučí do jednoho uzlu. Obě doposud používané stromové struktury byly immutabilní, ale platformní *Host View Tree* je možné upravovat. Fabric proto zjistí rozdíly mezi stromem připraveným k zobrazení a tím, který byl vykreslen naposledy, a vygeneruje operace, které starý strom převedou na ten nový. Nový strom se označí za vykreslený — je potřeba ho zachovat pro porovnání v příštím běhu cyklu — a operace se předají platformně závislému modulu, který potřebné změny aplikuje. [40, 41]

3.3 Kotlin Multiplatform

Technologie Kotlin Multiplatform je vyvíjena společností JetBrains, která stojí za jazykem Kotlin, a dalšími open source přispěvateli. Přestože název této kapitoly naznačuje, že se jedná o framework, je to ve skutečnosti SDK (software development kit), tedy sada nástrojů, které umožňují multiplatformní vývoj pomocí jazyka Kotlin. Cílem technologie Kotlin Multiplatform je sloučit výhody psaní kódu pro více platforem a tvorby nativních aplikací, čímž se zásadně odlišuje od výše popsaných frameworků Flutter a React Native. Namísto jednotné aplikace pro všechny cílové systémy a platformě závislé části — engine, respektive renderer —, která zajistí běh kódu na platformě, umožňuje Kotlin Multiplatform vytvářet společný kód, který je možné zkompilovat po jednotlivé platformy. Samotná aplikace je nativní, ale může tento společný, pro všechny platformy stejný, kód volat. Kód je možné využít jak v Android a iOS aplikacích, tak i na webu pro frontend i backend nebo v desktopových aplikacích pro Windows, macOS a GNU/Linux. V této práci se zaměříme především na Kotlin Multiplatform Mobile (KMM), který je určený pro vývoj aplikací pro mobilní zařízení. V říjnu 2022 vyšla beta verze KMM, což JetBrains prezentuje jako téměř stabilní verzi, kterou je bezpečné začít aktivně používat. V současnosti již tuto technologii používají například ve společnostech Netflix, Philips, Baidu nebo VMWare. Do větší hloubky si Kotlin Multiplatform popíšeme v kapitole 4. [43, 44, 45]

3.4 Porovnání

Nyní mezi sebou jednotlivé technologie pro multiplatformní vývoj porovnáme. Popíšeme zde, jak rozdíly mezi nimi ovlivňují nejen vývoj, ale také výsledný dojem na uživatele. Jak se od sebe frameworky liší? Umožňují nějakým způsobem využít existující nativní aplikace? Kdy je vhodné zvolit kterou technologii? A do jaké míry omezují vývojáře, když je potřeba využít platformně specifické funkce? To všechno jsou důležité otázky, které si nyní budeme klást.

3.4.1 Rozsah sdílení kódu

Nejzřetelnějším rozdílem při tvorbě aplikací pomocí výše popsaných technologií je rozsah vývoje, do kterého zasahují. Frameworky Flutter a React Native slouží ke kompletnímu multiplatformnímu vývoji všech částí aplikace — logiky i uživatelského rozhraní. Kotlin Multiplatform Mobile oproti tomu do vývoje UI vůbec nezasahuje. [21, 33, 44]

Jednodušší aplikace se tak dají pomocí frameworků Flutter nebo React Native vytvořit rychleji, protože uživatelské rozhraní se vytvoří jednou pro obě platformy. Problém nastává, pokud je aplikace složitější, nebo potřebuje využívat funkce, které frameworky ještě nepodporují. V takovém případě práci frameworky spíš komplikují.

Kotlin Multiplatform Mobile má z pohledu rychlosti a složitosti vývoje jistou nevýhodu — je potřeba vyvíjet UI pro Android a iOS zvlášť, což zabere více času, respektive je potřeba mít minimálně dva vývojáře, případně týmy vývojářů. Jedná se přeci jen o jiné platformy s jinými nástroji, pro jejichž hlubší znalost je potřeba určitá specializace. Jelikož ale tito vývojáři řeší pouze UI a jeho napojení na společný kód, je možné tuto práce přenechat i méně zkušeným vývojářům. Fakt, že KMM do tvorby UI nijak nezasahuje, je však z jiného úhlu pohledu výhodou. Vývoj UI není na této technologii závislý a probíhá nativně. Vývojář má díky tomu širší možnosti výběru používaných nástrojů a může například využít UI frameworky jako jsou SwiftUI, Jetpack Compose nebo jakýkoliv jiný, v budoucnosti představený, framework. [22]

Dalším faktorem, který může být důležitý především pro uživatele, je velikost aplikace. V tomto ohledu jsou přirozeně nejlepší nativní aplikace. Holá, minimální aplikace může být až 10 krát menší, než ta vytvořená pomocí frameworku Flutter nebo React Native. Avšak tento poměr rozhodně není fixní. Velikost frameworku Flutter ovlivňuje především použití vlastního vykreslovacího enginu a widgetů, které musí být k aplikaci přibalené. React Native zase obsahuje interpreter jazyka JavaScript a renderer, který propojuje multiplatformní a nativní grafické prvky. V tomto ohledu má technologie KMM výhodu minimálně pro Android, pro který se společná část zkompiluje do Java bytekódu, což je pro Kotlin přirozené, takže se jeví jako nativní. Pro systém iOS je kód z jazyka Kotlin zkompilován do nativního kódu, ale aby se k němu dalo přistupovat z jazyků Objective-C, případně Swift, vygenerují se navíc také adaptéry, které mohou výslednou velikost aplikace zvětšit. Toto se dá omezit skrytím části tříd ze sdíleného kódu pomocí klíčového slova `internal`. [46, 47]

3.4.2 Odhad minimálního počtu vývojářů

Pro porovnání minimálního počtu vývojářů potřebných k tvorbě aplikace pro obě platformy budeme zvlášť uvažovat vývojáře frontend a backend částí, a k tomu budeme předpokládat, že se vývojáři specializují jen na jednu platformu. Jednotlivé počty vidíme souhrnně v tabulce 3.1. V případě tvorby zcela nativních aplikací, bez využití výše zmíněných frameworků, potřebujeme alespoň čtyři vývojáře — jednoho na frontend a jednoho na backend pro každou z platforem. Při použití frameworku Flutter nebo React Native nám stačí dva vývojáři — frameworky nám umožňují kompletní vývoj pro obě platformy, rozdělujeme pouze frontend a backend. Kotlin Multiplatform Mobile vyžaduje využití alespoň tří vývojářů — dva pro Android a iOS frontend a jeden pro backend.

Tabulka 3.1 Minimální počet potřebných vývojářů při použití porovnávaných frameworků

	Nativní vývoj	KMM	Flutter	React Native
Počet vývojářů	4	3	2	2

Z pohledu počtu vývojářů nedopadla technologie KMM nejlépe, nicméně je potřeba zmínit, že frontend vývojáři v tomto případě vyvíjí pro jednotlivé platformy nativně, bez nutnosti mít znalost jakéhokoliv dalšího frameworku, či technologie. Pro společný kód navíc KMM využívá programovací jazyk Kotlin, který je v oblasti mobilního vývoje rozšířený. Je používaný pro vývoj Android aplikací a podporovaný společností Google. Pro mobilní vývoj je tedy přirozenější volbou, než JavaScript nebo Dart. React Native je založený na jazyku JavaScript, který sice rozšířený je, ale (až na tento framework) ne v oblasti mobilního vývoje. Mobilních frameworků založených na JavaScriptu je sice víc, ale žádný není natolik rozšířený, jako React Native. Framework Flutter v tomto porovnání dopadá nejhůře, protože jazyk Dart je používán v podstatě jen v rámci tohoto frameworku. Konkrétní programovací jazyk sám o sobě o použitelnosti frameworku nic nevypovídá. Některé jazyky jsou oblíbenější, v jistých ohledech mohou být lepší, každopádně pokud jsou ve vývoji mobilních aplikací již nějaké programovací jazyky zažité — Java, respektive Kotlin, pro Android a Swift, případně Objective-C, pro iOS —, nedá se multiplatformní technologii upřít výhoda, že používá právě jeden z těchto jazyků. [48, 49]

3.4.3 Využití existující aplikace

V situaci, kdy už máme aplikaci pro jednu z mobilních platforem, bychom mohli chtít kód této aplikace využít a nezačínat s vývojem znovu od začátku. Flutter i React Native oba umožňují jistou integraci do existující aplikace. Tato integrace ale nabízí pouze možnost obohatit aplikaci o další obrazovky, ne možnost tvorby aplikace pro druhou platformu. KMM o sobě tvrdí, že je jednoduché začít používat Multiplatform Mobile v existujících projektech. [50, 51, 22]

I s použitím technologie KMM jsme schopni obohatit funkcionalitu existujících Android a iOS aplikací. Kotlin Multiplatform umožňuje vytvořit společný modul, který jsme schopni použít v obou nativních projektech. Toto řešení bude pravděpodobně o něco čistší, protože výstupem KMM je nativní kód, který se do existujících aplikací integruje přirozeněji, než celý framework. Takové použití může být jistě užitečné, ale je potřeba si uvědomit, že existující kód nesdílíme, a proto by se nejspíše muselo jednat o zcela novou nezávislou funkcionalitu.

Pokud chceme využít celý potenciál multiplatformního vývoje, budeme muset začít od znova a existující aplikace příliš nevyužijeme. Z třech zde porovnávaných technologií umožňuje částečné využití nativních aplikací pouze Kotlin Multiplatform Mobile. V jeho případě je nejlepší a nejjednodušší způsob, jak existující projekty využít, ponechání uživatelského rozhraní, do jehož tvorby KMM vůbec nezasahuje. V případě, že máme Android aplikaci, se také můžeme část existující logiky pokusit přepsat na multiplatformní kód. Části specifické pro Android je potřeba nahradit buď obecným multiplatformním řešením, nebo je označit jako specifické pro Android a implementovat variantu pro iOS. Rozsah těchto změn vždy závisí na konkrétní aplikaci. Pro některé může jít o proveditelný krok, pro jiné může být snazší stávající logiku aplikace vůbec nepoužít a začít na zelené louce. [52]

3.4.4 Integrování platformě specifického kódu

Dalším faktorem při výběru, který framework použijeme, je potřeba využívat platformě specifickou funkcionalitu a funkcionalitu nových verzí Android a iOS. Všechny tři porovnávané frameworky nějakých způsobem umožňují psaní kódu pro konkrétní platformu. Co je odlišuje, je za jakých účelem si představují použití této možnosti.

Flutter a React Native ji chápou jako v lepším případě doplněk, spíše však obcházení problému. Jejich hlavním cílem je psát multiplatformní kód. Z toho důvodu je použití mířené spíše na obecné problémy, při kterých není potřeba zabývat se konkrétní platformou. Protože je vývoj pomocí těchto technologií odlišný a oddělený od nativního vývoje, neobejde se propojení společného a platformně závislého kódu bez problémů. [47]

Kotlin Multiplatform toto řeší už tím, jak je koncipován. Protože se od prvopočátku nesnaží vývojáře oddělit od konkrétních platforem a skutečně sdílený kód je vždy doplňován tím platformě specifickým, nedochází k nárůstu složitosti při potřebě využití funkcionalit konkrétního operačního systému. Přístup k těmto funkcím je přirozenější a nevybočuje ze zbytku vývoje.

Funkcionalita mobilních telefonů, která je již zažitá a aplikace ji běžně využívají, bývá z frameworků přístupná výrazně snadněji, než zcela nová funkcionalita, případně vlastnosti specifické jen pro jednu z platforem. Například přístup k fotoaparátu nebo využití geolokace jsou činnosti, které půjdou implementovat snadno. Pro námi diskutované frameworky existuje řada knihoven, které umožňují multiplatformní přístup k nativním API. Tyto npm, respektive Dart, balíčky jsou ale častokrát vyvíjeny komunitou a není možné předpokládat, že pro každou novou funkcionalitu někdo okamžitě vytvoří knihovnu pro námi používaný framework. Na další potencionální problémem poukazuje fakt, že pro jednu funkcionalitu často existuje více knihoven, což může znamenat, že některé knihovny nebyly pro daný projekt použitelné a bylo potřeba vytvořit alternativu. V případě, že existuje širší výběr, zřejmě nenarazíme na problém, nicméně v některých případech nám knihovna nemusí nabídnout požadované řešení. Tímto problémem samozřejmě trpí i knihovny pro Kotlin Multiplatform, u kterého rozsah použití společného kódu značně závisí právě na multiplatformních knihovnách. Významným rozdílem je opět filosofie KMM, která oproti frameworkům Flutter a React Native umožňuje výrazně jednodušší řešení. Kotlin Multiplatform má mnohem blíže k nativnímu vývoji a s částečným rozdělením implementace pro jednotlivé platformy přímo počítá. Knihovny tedy nemusí pokrývat veškeré potřeby projektu, nicméně propojení společného kódu a implementací pro jednotlivé platformy je zde jednodušší.

3.4.5 Odhadovaná cena vývoje

Róbert Nagy ve své knize[1] o technologii Kotlin Multiplatform představil zjednodušený odhad ceny vývoje. Uvádí, že vytvoření realističtějšího srovnání by bylo příliš komplikované, a proto si myslí, že jeho odhad je sice zjednodušený, nicméně pro účely porovnání jednotlivých technologií dostačující. Jde o odhadovanou cenu vývoje pro dvě platformy v závislosti na komplexnosti funkcionality. Komplexnost, namísto počtu, se Nagy rozhodl použít, protože jednotlivé funkcionality se mohou svým rozsahem a složitostí výrazně lišit. [47]

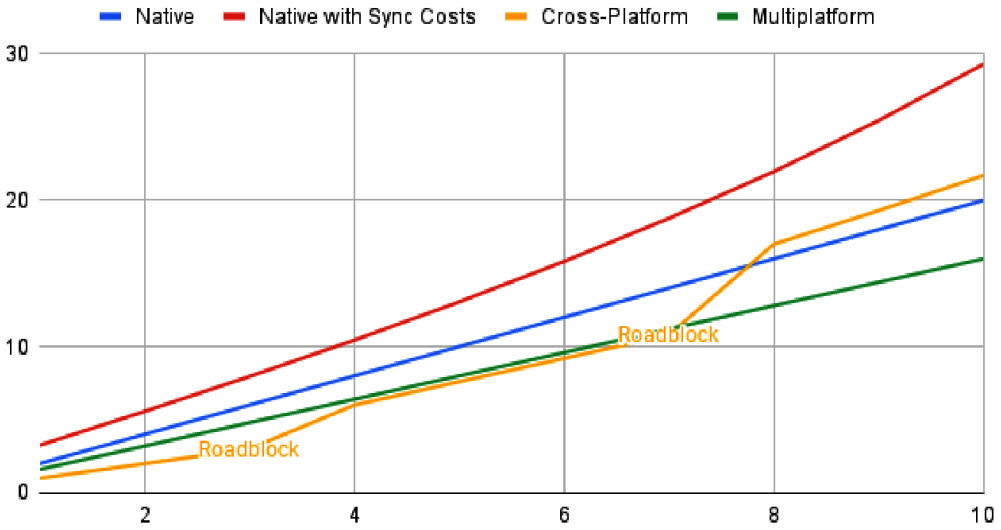
Graf můžeme vidět na obrázku 3.5. Róbert Nagy používá pro frameworky Flutter a React Native označení *cross-platform*, čímž je vymezuje od ideově odlišného multiplatformního vývoje, který pojmenoval podle technologie Kotlin Multiplatform. Červený i modrý graf znázorňuje nativní vývoj, bez použití frameworku. V případě červeného jde o rozšířený a zpřesněný odhad, který navíc bere v úvahu synchronizační cenu mezi odděleným nativním vývojem pro Android a iOS. [47]

Žlutý graf pro cross-platformní frameworky obsahuje problematické části, při kterých se zvyšuje cena vývoje. Může jít o problémy, při kterých je potřeba upustit od klasického použití frameworku a zabývat se integrací platformě závislého kódu, případně jinak obcházet používaný framework. Takové komplikace samozřejmě mohou při tvorbě aplikace nastat opakovaně. Konkrétní počet problematických částí, jejich závažnost a rychlost výskytu při vývoji, se bude v každém projektu lišit. Je však jisté, že s rostoucí komplexností aplikace bude přibývat i počet komplikací. [47]

Pro jednoduché aplikace bude cena vývoje nejnižší při použití frameworků Flutter nebo React Native. S rostoucí složitostí se ale cena cross-platformních frameworků srovná s cenou technologie

[1] *Simplifying Application Development with Kotlin Multiplatform Mobile* [47]

Obrázek 3.5 Porovnání ceny vývoje v závislosti na rostoucí komplexitě funkcionality. [47]



Kotlin Multiplatform. Pro větší projekty, které závisí na použití platformě specifických funkcionalit, nebo pro projekty, které dál porostou a do budoucna se jim bude rozšiřovat funkcionalita, bude pravděpodobně cenově výhodnější použití technologie Kotlin Multiplatform. [47]

Další dimenzí, kterou však graf 3.5 nezachycuje, je kvalita výsledné aplikace. Dá se ale očekávat, že nativním řešením bude možné dosáhnout kvalitnějšího výsledku, než pomocí cross-platformních frameworků, které staví mezi vývojáře a platformu vlastní vrstvu. V případě frameworku Flutter je to jeho renderovací engine, v případě frameworku React Native je to bridge, který zajišťuje komunikaci mezi JavaScriptovým a nativním kódem. [47]

Róbert Nagy porovnání zakončil svým pohledem na výběr vhodné technologie. Pokud aplikace hluboce závisí na nativních prvcích a je vyžadována její vysoká kvalita, doporučuje vyhnout se cross-platformním frameworkům a programovat nativně. Pokud je potřeba vyvážit cenu vývoje a kvalitu aplikace, doporučuje Kotlin Multiplatform. Zmiňuje také, že jelikož KMM dovoluje sdílet jen část kódu, je možné kombinovat nativní a multiplatformní vývoj. Cross-platformní frameworky jsou sice co do nákladů nejefektivnější, je to ale vykoupeno nižší kvalitou aplikace. Doporučil by je spíš pro projekty s krátkou životností. [47]

3.4.6 Uživatelské rozhraní

Aplikace vyvíjené pomocí frameworku Flutter, který používá vlastní renderovací engine, vypadají stejně na všech platformách. Problém je, že vzhled aplikací pro Android a iOS se většinou liší. Každý z operačních systémů má vlastní doporučení, jak by měly aplikace vypadat. Pro uživatele, kteří jsou zvyklí na vzhled nativních aplikací, může být jinak vypadající UI neintuitivní, či matoucí. Pro některý typ aplikací, především aplikací s unikátním uživatelským rozhraním, to může být výhoda, ale pokud se aplikace skládá ze zažitých UI prvků, je pravděpodobně žádoucí zachovat vzhled — *look and feel* —, na který jsou uživatelé dané platformy zvyklí. Dále je potřeba mít na paměti, že Apple aplikace před přidáním do AppStore kontroluje. Pro úspěšné přijetí musí iOS aplikace dodržovat předepsané zásady (*Apple Interface Guidelines*). [22, 53]

React Native i Kotlin Multiplatform využívají, na rozdíl od frameworku Flutter, nativní vizuální prvky. Jak jsme si již výše popsali, uživatelské rozhraní se ve frameworku React Native vytváří pro všechny platformy jednotně, nicméně základní komponenty se při renderování reprezentují nativními grafickými prvky. Jde tedy o kompromis, kdy drobnější prvky působí nativně — a nemusí jít pouze o vzhled políček pro textový vstup, ale například o podobu upozornění nebo

výběru data a času, což uživatelův zážitek ovlivní výrazně víc —, ale celkové rozložení aplikace bude pro iOS i Android stejné. Přestože mají oba zde zmiňované mobilní operační systémy vlastní pohled například na navigaci mezi obrazovkami, nemusí být jednotné rozložení problém. Material Design upřednostňuje menu, které vyjede ze strany, zatímco Apple doporučuje použití lišty na dolním okraji obrazovky. Jde ale pouze o doporučení a některé z těchto prvků, zvlášť spodní navigační lišta se objevuje i v Android aplikacích. Je však otázkou, zda je to z důvodu snahy o zlepšení ovládání aplikace — spodní lišta může být v případě malého počtu položek výrazně přehlednější a rychleji dostupná než menu schované za tlačítkem —, nebo je to pouze vedlejší efekt používání multiplatformních frameworků pro vývoj aplikací. [23, 54]

V případě technologie KMM probíhá vývoj uživatelského rozhraní zcela nativně, odděleně pro každou z platforem. Vývojáři sice mají větší kontrolu nad výslednou podobou pro jednotlivé platformy, ale musí UI vytvářet dvakrát a současně mezi sebou koordinovat vzhled aplikace. Cílem je totiž vytvořit platformní varianty, ne zcela odlišné aplikace. Je potřeba sjednotit design prvků, barevné schéma aplikace, rozložení funkcionalit mezi obrazovky apod. Mnoho z těchto rozhodnutí se provede dohromady, ale stejně dochází ke zdvojení. V tomto ohledu balancuje přístup frameworku React Native mezi výhodami nativního vývoje (s využitím nativních prvků) a výhodami společné tvorby UI pro více platforem.

3.4.7 Shrnutí

Frameworky Flutter, React Native a Kotlin Multiplatform přistupují k tvorbě multiplatformních aplikací odlišným způsobem. Flutter poskytuje možnost jednotného vývoje aplikací včetně uživatelského rozhraní a pomocí vlastního grafického enginu zcela obchází nativní SDK. Pro použití jakékoliv nativní funkcionality jsou vývojáři závislí na dodatečných Flutter balíčcích, které zajistí přístup k platformnímu rozhraní. Hodí se především pro jednodušší aplikace s totožným UI napříč platformami a umožňuje rychlou tvorbu s menším počtem vývojářů.

React Native se frameworku Flutter v mnohém podobá. Protože implementace uživatelského rozhraní probíhá jednotně, je opět možné aplikace vyvíjet v menších týmech. Stejně tak ale zůstává problém se závislostí na externích knihovnách pro zajištění rozsáhlejší funkcionality. Oba dva frameworky oddělují uživatele od nativního vývoje, což usnadňuje hodně práce, ale zároveň omezuje možnosti vývojářů. Čím se React Native liší, je využívání nativních grafických prvků, což umožňuje tvorbu aplikací, které do jisté míry zachovávají vzhled a zážitek nativního uživatelského rozhraní. S propojením aplikace a nativního API, využitím jazyka JavaScript a runtime architekturou však souvisí nižší výkonnost v porovnání s frameworkem Flutter. [46]

Technologie Kotlin Multiplatform se svým přístupem k multiplatformnímu vývoji výrazně liší. Je mnohem blíže nativnímu vývoji, který „pouze“ obohacuje o možnost sdílení části kódu, namísto vytvoření vrstvy oddělující nativní kód a vývojáře. Je stále nutné psát nativní kód, ale tím pádem také odpadá většina problémů spojená s obcházením frameworku pro přístup k nativnímu API. Nevýhodou je, že technologie KMM nenabízí možnost tvorby uživatelského rozhraní pomocí společného zdrojového kódu. To zvyšuje náklady na vývoj, ale zároveň také umožňuje větší všestrannost a přizpůsobivost. Nabízí tak lepší kontrolu nad rozsahem využití. Svou blízkostí k nativnímu vývoji — vzhledem k využití jazyka Kotlin především pro systém Android — navíc umožňuje jednodušší adaptaci nativních vývojářů na tuto technologii a možnost postupného přechodu aplikováním technologie pouze v menším rozsahu.

I z odhadu minimálního počtu vývojářů (popsaného výše v této kapitole) vyplývá, že se technologie KMM nachází někde mezi nativním vývojem a ostatními cross-platformními frameworky. Díky možnosti sdílet části kódu ubývá počet potřebných vývojářů, avšak stále jich je nutných více, než při použití frameworků. KMM je vhodný pro vývoj složitějších aplikací nebo aplikací s dlouhou životností.

Pokud se technologie Kotlin Multiplatform v budoucnu uchytí, dá se předpokládat, že půjde právě o rozšíření nativního vývoje. V případě, že cross-platformní frameworky nenabídnou požadované možnosti a vývojáři se rozhodnou implementovat aplikaci nativně, budou mít stále

možnost sdílet část kódu a snížit tak čas, respektive náklady na vývoj. Jelikož je však jednotná tvorba UI velice lákavá možnost, technologie KMM — ani v případě potenciálního ohromného úspěchu a míry využití — jistě nevytlačí zbývající frameworky, které nabízí řešení jiných problémů, pro jiné typy aplikací.

Kapitola 4
Kotlin Multiplatform

V této kapitole se hlouběji zaměříme na jazyk Kotlin a technologii Kotlin Multiplatform. Popíšeme si její vnitřní fungování a rozebereme jednotlivé části, které umožňují její úzké propojení s nativním vývojem. Kotlin Multiplatform je souhrnné označení, používané společností JetBrains, pro nástroje, které umožňují využití programovacího jazyku Kotlin pro implementaci aplikací určených pro různé platformy s možností sdílení části kódu mezi více platformami. Pro pochopení nejdříve krátce popíšeme jazyk Kotlin a jeho verze, specifické pro různé platformy. [43]

4.1 Jazyk Kotlin

Kotlin je staticky typovaný programovací jazyk vyvíjený firmou JetBrains. Snaží se napravit časté problémy, které se vyskytují v jiných jazycích, proto je navržen tak, aby byl snadno čitelný a aby svým návrhem předcházel chybám. Vznikl s cílem být plně interoperabilní s jazykem Java, ale oproti jazykům jako Scala, Ceylon nebo Clojoure není akademickým počinem. Přestože i Java se vyvíjí a doplňuje jazyk o populární a užitečné konstrukty, Kotlin jich stále nabízí víc, než Java 8 nebo i 11. Snaží se být pragmatický, stručný a bezpečný. Jazyk Kotlin je nejen objektově orientovaný, ale také funkcionální. Obsahuje tedy třídy a rozhraní, ale i funkce vyšších řádů, lambda výrazy nebo líné vyhodnocení. [55]

Vývoj Kotlinu započal roku 2010. Poté, co v roce 2016 vyšla verze 1.0, netrvalo společnosti Google ani rok, aby přijala Kotlin jako oficiálně podporovaný jazyk pro vývoj Android aplikací. V počátku sloužil Kotlin především jako alternativa k Javě a běžel na Java Virtual Machine (JVM). Experimentální podpora multiplatformního vývoje byla zahájena v roce 2017 s možností sdílení kódu pro JVM a JavaScript. Společnost JetBrains také oznámila, že pracuje na projektu Kotlin/Native — překladači Kotlinu do nativních binárních kódů. V tuto chvíli je možné v Kotlinu programovat mobilní, webové i desktopové aplikace. [56, 57]

Kotlin má rozsáhlou syntaxi, o které vznikají celé knihy. Proto si zde tento jazyk představíme jen velmi stručně. Pro deklarování proměnných se v Kotlinu používají klíčová slova `val` a `var`. První zmíněné slouží pro vytvoření immutabilní proměnné, kterou je vhodné používat co nejčastěji. Tuto proměnnou není možné změnit, ale to samozřejmě neznamená, že nelze změnit samotný objekt. Při deklaraci, je možné specifikovat datový typ, ale Kotlin překladač umí ve většině případů typ sám odvodit.

```
1 val name: String = "Tom"
```

Oproti Javě lze v Kotlinu pro přiřazení hodnoty do proměnné použít výraz `if-else`, který vrací hodnotu, a Kotlin proto neobsahuje ternární operátor. Cykly jsou v případě `for` varianty typu `foreach` a pro iterování čísel používají rozsah. [55]

```
for (i in 1..9) {
  println(i)
}
for (i in 1 until 10) {
  println(i)
}
```

Při práci s vnořenými cykly se může hodit možnost označit ten, ke kterému se má vztahovat výraz `break` nebo `continue`.

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

Stejně tak lze ovlivnit i místo návratu při volání `return` z lambda funkcí. [55]

Funkce se deklarují pomocí klíčového slova `fun`. Parametrům je možné přiřadit výchozí hodnoty a při volání je určit jménem. [55]

```
fun incA(value: Int, increment: Int = 1): Int {
  return value + increment
}
fun incB(value: Int, increment: Int = 1) = value + increment

incA(5)                 // 6
incA(5, 2)              // 7
incA(5, increment = 2) // 7

val incLambda =      { value: Int, increment: Int -> value + increment }
val incAnonymous = fun(value: Int, increment: Int): Int = value + increment

incLambda(5, 1)        // 6
incAnonymous(5, 1)     // 6
```

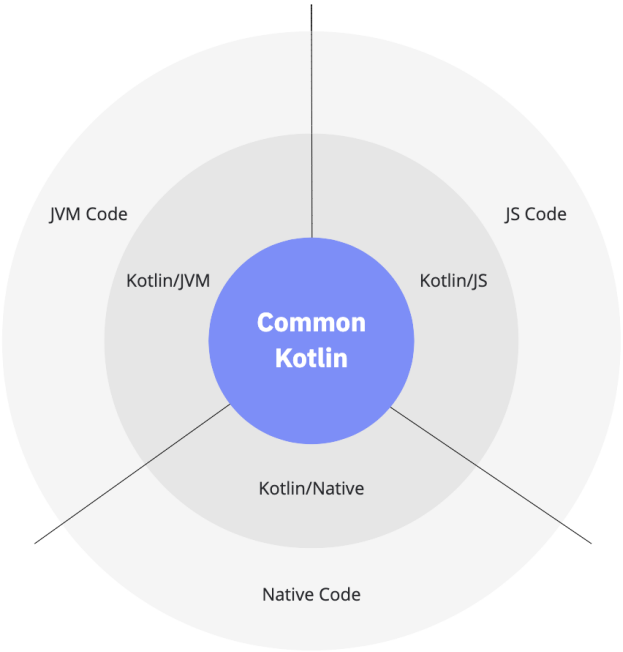
Nejspíše nejznámější vlastností Kotlinu je null safety. Kotlin má dva základní typy proměnných — obyčejné a nullable. Aby mohly proměnné nabývat hodnotu `null`, použijeme verzi datového typu s otazníkem na konci. Pro bezpečné volání takových proměnných používáme operátor `?.` a pro určení náhradní hodnoty (namísto `null`u) použijeme Elvis operátor (`?:`). Pokud v podmínce zkontrolujeme, že hodnota není `null`, překladač v těle podmínky dál nepokládá proměnnou za nullable. [55]

```
var name: String? = null

val len1 = name?.length ?: 0
val len2 = if (name != null) name.length else 0

println("Hello, ${name ?: "stranger"}!")
```

4.2 Kotlin pro různé platformy

Přestože je v obecném povědomí Kotlin spjatý především s JVM — a tím pádem s jazykem Java —, cílí i na několik dalších platforem. Tím však namyslíme přímo multiplatformní vývoj, nýbrž možnost kompilace třeba do nativního kódu, například pro systémy Windows, nebo macOS.

Obrázek 4.1 Diagram základního rozdělení Kotlin Multiplatform [60]



Jednotlivé platformní varianty, které umožňují vyvíjet pomocí Kotlinu pro JVM, web nebo různé operační systémy si zde nyní představíme. [43]

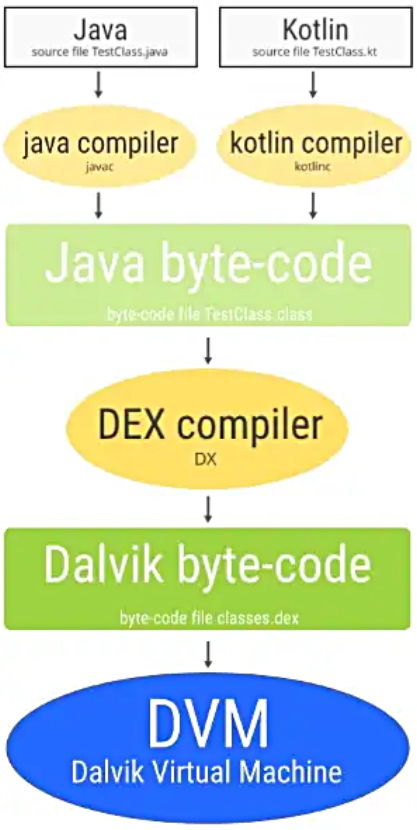
Platformě specifické verze Kotlinu jsou Kotlin/JVM, Kotlin/JS, Kotlin/Native a nově také Kotlin/Wasm (WebAssembly). Jedná se o kombinaci specifických implementací standardní Kotlin knihovny, různých rozšíření jazyka, doplňujících knihoven, kompilátorů a dalších nástrojů. Pro každou z těchto čtyř variant má Kotlin samostatný kompilátor, respektive backend kompilátoru. Tématice Kotlin kompilátoru se více věnuje kapitola 4.3. [43, 58, 59]

Samostatné použití těchto platformních variant je samozřejmě možné, nicméně jejich hlavním přínosem je využití pro multiplatformní tvorbu. Jelikož se v jádru vždy jedná o jazyk Kotlin, který je pro různé platformy pouze obohacen o specifická API, je možné část kódu sdílet. Společnou částí všech variant je Common Kotlin, který lze použít pro kteroukoliv z podporovaných platforem. Kromě samotného jazyka obsahuje také například core knihovny, ale nenajdeme v něm části specifické například pro Kotlin/JS. Toto rozdělení demonstruje obrázek 4.1. Common Kotlin obsahuje značnou část jazyka včetně deklarací, podmínek, typového systému nebo kolekcí, ale pro přístup k JavaScript konzoli, JSON objektu nebo DOM API jsou zapotřebí rozšíření varianty Kotlin/JS. Stejně tak funkce vázané na operační systém jsou dostupné pouze pro Kotlin/JVM a Kotlin/Native varianty. [43, 61]

4.2.1 Kotlin/JVM

Za základní variantu Kotlinu by se dal považovat Kotlin/JVM. Jde o jeho nejrozšířenější podobu, která se používá především pro vývoj serverových aplikací a aplikací pro Android. Průzkum firmy JetBrains z roku 2020 ukázal, že 65 % respondentů využívá Kotlin k vývoji Android aplikací, 48 % k vývoji Kotlin/JVM serverových aplikací a pro jiné typy aplikací využívá Kotlin/JVM 15 % dotazovaných. Další anketa, taktéž provedená firmou JetBrains, z roku 2021 odhalila, že 49 % dotazovaných programují v Kotlinu mobilní Android aplikace a 67 % serverové aplikace. U druhé ankety je potřeba uvést, že u serverových aplikací není explicitně uvedeno, zda jde

Obrázek 4.2 Diagram procesu kompilace Kotlinu pro platformu Android. [64]



o Kotlin/JVM, nebo Kotlin/JS, nicméně použití Kotlin/JS v kombinaci s technologií Node.js, ač možné, není příliš pravděpodobné. Dotazník z roku 2020 ukázal, že Kotlin/JS používalo jen 6 % dotazovaných. [62, 63]

Z názvu Kotlin/JVM není podpora Androidu přímo zřetelná a mohlo by se zdát, že tato varianta cílí na Java Virtual Machine, což není zcela pravda. Kód se kompiluje do Java bytekódu, díky čemuž může běžet na JVM, ale i na Androidu. Hojně rozšířené serverové aplikace běží právě v JVM, zatímco situace s Androidem je poněkud složitější. Kompilace aplikací pro operační systém Android probíhá v zásadě ve dvou krocích. První z nich je kompilace Kotlin, respektive Java, kódu do bytekódu, ale protože Android nepoužívá JVM, je potřeba bytekód ještě dál transformovat. Android používal do verze 4.4 virtuální stroj Dalvik (DVM). Od verze 5.0 používá Android výhradně Android Runtime. V obou případech je bytekód dál transformován do podoby, v které ho jde na Androidu spustit. Tento proces můžeme vidět na obrázku 4.2. Díky společnému bytekódu lze jazyky i kombinovat, například doplnit existující Java kód kódem v jazyku Kotlin. Přestože lze jazyky kombinovat a sdílejí stejnou mezivrstvu (bytekód), Kotlin nabízí řadu užitečných vlastností, které v Javě — alespoň v té verzi, kterou podporuje Android — chybí. [55]

Mezi společnosti, které používají Kotlin k vývoji mobilních aplikací, patří mimo jiné Autodesk nebo Netflix. Mezi zástupce společností, které využívají Kotlin/JVM pro serverové aplikace patří Netflix, Adobe nebo Atlassian se svým nástrojem Jira. [65, 66, 67]

4.2.2 Kotlin/JS

Tato varianta Kotlinu umožňuje transpilaci z jazyku Kotlin do jazyku JavaScript. Umožňuje tedy vývojářům využívat všechny oblíbené vlastnosti Kotlinu, jako je typování a null safety, a samozřejmě standardní knihovnu. Nabízí také přístup k některým webovým rozhraním, například pro práci s DOM. Výstupem Kotlin/JS je JavaScriptový kód, který současná verze vytváří dle specifikace ES5. [68]

Ačkoliv podle již výše zmiňovaných průzkumů Kotlin/JS používalo v roce 2020 jen 6 % respondentů, stabilní podpora pro JavaScript existuje už od verze Kotlin 1.1 (vydané 2016-02-15). Možností, jak využít tuto variantu, je několik. Může sloužit pro vývoj webových frontend aplikací, k čemuž existuje několik frameworků specificky pro Kotlin/JS, nebo dokonce i wrapper pro framework React. S využitím Node.js je také možné vytvářet serverové aplikace. Další možností je vytvořit knihovnu, kterou je dále možné využít v jakémkoliv JavaScript nebo TypeScript kódu. Tím se již přibližujeme multiplatformnímu využití Kotlinu. Pokud vytvoříme takovou knihovnu pomocí čistě Common Kotlin závislostí, nic nám nebrání ji využít nejen pro JavaScript frontend, ale i pro JVM backend server, mobilní nebo desktopovou aplikaci. [62, 63, 69, 68]

Avšak Kotlin/JS nabízí s jazykem JavaScript ještě bližší provázání. Umožňuje volat JavaScript kód buď přímo — vyhodnocením inline řetězce —, nebo pomocí externích deklarací. Externí deklarace nemá tělo a kompilátor ji nepřekládá, protože se očekává existence odpovídající implementace buď v jiném JavaScript souboru, nebo z npm balíčku. Slouží pouze pro informování Kotlinu, že taková funkce, případně třída, existuje. Tyto deklarace je bohužel zatím potřeba psát ručně. Existuje sice nástroj Dukat, který se snaží konvertovat TypeScript definice (z `.d.dts` souborů), jde však o experimentální projekt, který v tuto chvíli není prakticky použitelný. Pro některé npm balíčky je sice možné najít již vytvořené externí deklarace, ale vzhledem k celkovému množství balíčků a nízkému využití Kotlin/JS to není příliš pravděpodobné. [70, 71]

Je nutné mít na paměti, že JavaScript není typovaný jazyk. Kotlin proto v případě, že nedefinujeme detailní externí deklarace, využívá typ `dynamic`. Tímto způsobem se v podstatě vypne typová kontrola, protože takovou hodnotu lze přiřadit do kterékoliv jiné proměnné a naopak do `dynamic` proměnné lze přiřadit libovolný datový typ. Dále je na těchto proměnných možné volat bez jakékoliv kontroly libovolné funkce, aniž by musely existovat. Takové volání se konvertuje doslovně. [70, 72]

Pro volání tříd a funkcí vygenerovaného JavaScript kódu je od konce roku 2022 (Kotlin verze 1.8.0) nutné označit deklarace anotací `@JsExport`. V této verzi byl totiž původní kompilátor nahrazen novým IR kompilátorem, který toto explicitní označení vyžaduje. Změna kompilátoru přinesla pro Kotlin/JS řadu změn, včetně několika vylepšení. Umožňuje například lepší optimalizaci, snížení velikosti vygenerovaného kódu nebo rychlejší, inkrementální kompilaci. Vygenerovaný kód je zabalen do JavaScript modulů, které odpovídají struktuře Kotlin balíčků. Přesný systém modulů — například UMD, nebo CommonJS — je možné nastavit. Kromě toho lze také zvolit přesné jméno vygenerované funkce. Jelikož může kompilátor v některých případech — především při přetěžování — jména dekorovat, je občas nezbytné použití anotace `@JsName("")`. Většina datových typů jazyka Kotlin je namapovaná na jejich přibližné JavaScript protějšky, nicméně některé využívají vlastní Kotlin implementaci. [73, 74, 75]

Výše jsme již zmínili několik možných přístupů pro použití Kotlin/JS. Tím pravděpodobně nejužitečnějším je využití v rámci Kotlin Multiplatform. Přestože může být možnost použití Kotlinu čistě pro tvorbu JavaScript kódu lákavá, komplikované použití npm balíčků a složitější hledání chyb nejspíše zabrání jejímu širšímu použití. Oproti tomu sdílení části, nebo dokonce celé, business logiky napříč platformami je v dlouhodobém horizontu pravděpodobně jediné uplatnění Kotlin/JS. Pro sdílení kódu mezi webem a mobilními aplikacemi používá Kotlin Multiplatform například aplikace TV5Unis, vyvinutá firmou Mirego. Pro stejný přístup se rozhodla i firma Memrise, která nabízí aplikaci pro výuku jazyků, nebo Down Dog, která navíc dokonce sdílí část kódu i mezi frontend aplikacemi a webovým backendem. [76, 77, 78]

4.2.3 Kotlin/Wasm

WebAssembly je přenosný nízkoúrovňový jazyk s binárním formátem určený pro běh na virtuálním stroji. Je určený především pro webové stránky jako doplnění jazyku JavaScript, z kterého je možné k WebAssembly funkcím přistupovat. Jeho cílem je být co nejefektivnější, ale zároveň zajistit kompatibilitu s ostatními webovými technologiemi. Ačkoliv je vysokoúrovňový, dynamicky typovaný JavaScript na webu široce rozšířený, jeho flexibilita má za následek nižší výkonnost. WebAssembly se snaží nabídnout možnost, jak problémy týkající se výkonnosti — například při renderování 3D grafiky, editování obrázků, úpravě videí v reálném čase, šifrování nebo jiných výpočetně náročných operacích — vyřešit. [79]

V současnosti již WebAssembly 1.0 podporují téměř všechny moderní prohlížeče, ale také například Node.js. K jeho vývoji slouží již existující programovací jazyky — především C, C++ a Rust —, z kterých je potom WebAssembly kompilované. Stejně tak lze pro vývoj použít i Kotlin. V aktuálně nejnovější verzi 1.8.20 (vydané 2023-04-03) byla přidána zatím experimentální varianta Kotlin/Wasm. Při vývoji cíleném na WebAssembly je možné využít standardní Kotlin knihovnu a knihovnu pro testování kódu. Podobně jako v případě varianty Kotlin/JS je možné pomocí klíčového slova `external` volat JavaScript kód. A naopak z JavaScriptu volat Kotlin deklarace označené anotací `@JsExport`. [79, 58, 80]

V minulosti již v Kotlinu existovala podpora pro WebAssembly pomocí varianty Kotlin/Native, která obsahovala cíl `wasm32` a zajišťovala kompilaci do LLVM, z kterého se teprve kompiloval WebAssembly kód. S představením Kotlin/Wasm však byla tato podpora ukončena a vývojáři díky samostatnému překladači (backendu) slibují rychlejší kompilaci a lepší interoperabilitu s JavaScriptem. [58]

4.2.4 Kotlin/Native

Varianta Kotlin/Native umožňuje kompilaci do nativního binárního kódu pro různé architektury (Windows, GNU/Linux, macOS, iOS, tvOS, watchOS a Android NDK). Tento kód lze spustit bez virtuálního stroje a hlavním účelem je především umožnit spouštět kód na zařízeních, která nedovolují běh JVM. Je to právě Kotlin/Native, díky kterému je možné cílit na mobilní zařízení s operačním systémem iOS. Tvorba multiplatformních aplikací pro systémy Android a iOS je nejspíše hlavní využití této varianty Kotlinu. Používají ji tak například společnosti Netflix nebo Autodesk, která sdílí část kódu aplikace PlanGrid i s verzí pro Windows. Zkušenosti z vývoje PlanGrid potvrzují, že využití této technologie pro iOS je výrazně jednodušší, než pro Windows. [81, 65, 82]

Kromě kompilace spustitelných souborů umožňuje Kotlin/Native také několik možností, jak spolupracovat s nativním kódem. Kód vytvořený v jazyku Kotlin je možné použít v nativní aplikaci — a to v rámci projektů implementovaných v C, C++, Objective-C nebo Swift — a naopak existující knihovny a další nativní kód je možné používat v rámci Kotlinu. Kotlin/Native používá automatickou správu paměti pomocí vlastního garbage collectoru, který je schopný kooperovat se Swift a Objective-C správou paměti (Automatic Reference Counting). V případě používaní C knihoven je však v rámci Kotlinu možné alokovat paměť také manuálně. [81, 83]

4.2.4.1 Volání Kotlinu z nativního kódu

Pro C a C++ projekty vytvoří Kotlin/Native kompilátor kromě samotného kódu knihoven (statických nebo dynamických) i hlavičkové soubory, které definují rozhraní v jazyku C. Číselné datové typy a třídy používané jazykem Kotlin pro reprezentaci ukazatelů jsou pomocí konstrukce `typedef` namapované na jejich příslušné varianty z jazyka C. Objekty a třídy jsou reprezentovány C strukturou, která obsahuje pouze ukazatel na samotná data. Pro deklaraci funkcí a samotného obsahu tříd se vygenerují zanořené C struktury, která odpovídá struktuře balíčků, v kterých se

Výpis kódu 4.1 Ukázka použití Kotlin kódu z jazyka C — část Kotlin. (převzato z Kotlin dokumentace [84])

```
package example

class Clazz {
  fun memberFunction(p: Int): ULong = 42UL
}

fun strings(str: String) : String? {
  return "That is '$str' from C"
}
fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }
```

Výpis kódu 4.2 Ukázka použití Kotlin kódu z jazyka C — část C. (převzato z Kotlin dokumentace [84])

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
  //obtain reference for calling Kotlin/Native functions
  libnative_ExportedSymbols* lib = libnative_symbols();

  lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
  lib->kotlin.root.example.forFloats(1.0f, 2.0);

  //use C and Kotlin/Native strings
  const char* str = "Hello from Native!";
  const char* response = lib->kotlin.root.example.strings(str);
  printf("in: %s\nout:%s\n", str, response);
  lib->DisposeString(response);

  //create Kotlin object instance
  libnative_kref_example_Clazz newInstance = lib->kotlin.root.example.Clazz.
      Clazz();
  long x = lib->kotlin.root.example.Clazz.memberFunction(newInstance, 42);
  lib->DisposeStablePointer(newInstance.pinned);

  printf("DemoClazz returned %ld\n", x);

  return 0;
}
```

Výpis kódu 4.3 Alokace nativní paměti pomocí Kotlinu (převzato z Kotlin dokumentace [85])

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<use buffer>
nativeHeap.free(buffer)

val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

třídy nacházejí. Jelikož koncept tříd není součástí jazyka C, metody vygenerovaného rozhraní obsahují navíc jeden parametr pro předání konkrétní instance. Volání tak může vypadat například jako na ukázce 4.2, kde voláme kód z výpisu 4.1. [84]

Stejně jako v případě jazyků C a C++ je možné volat kód z jazyků Objective-C a Swift. Změnou konfiguračního gradle souboru můžeme změnit výstup kompilátoru na Apple framework. Kotlin/Native nám v tom případě také vygeneruje hlavičkové soubory pro Objective-C, které definují rozhraní pro komunikaci s Kotlin kódem. Ačkoliv jde o jiné jazyky, přístup je velice podobný, jako v případě jazyka C/C++. Číselné typy Kotlinu jsou namapovány na potomky `NSNumber`, ve Swiftu například `KotlinInt` nebo `KotlinBoolean`. Kotlin kolekce jsou také namapovány na jejich odpovídající typy ve Swiftu, respektive Objective-C. Typ `List` je navázán na `Array`, respektive `NSArray`. Ze tříd a objektů se stane `@interface`, které rozšiřuje `KotlinBase` a má v případě Objective-C název vždy s předponou podle názvu frameworku. Poslední co stojí za zmínku z tématu mapování, jsou globální deklarace. Ty jsou uschované do třídy `LibKt`, respektive `<Package>LibKt`. Jelikož jsou Kotlin i Swift oba moderní jazyky, není ve výsledku v kódu zas tak velký rozdíl, rozhodně ne tak velký, jako v případě volání kódu z jazyka C. [83]

4.2.4.2 Volání nativního kódu z Kotlinu

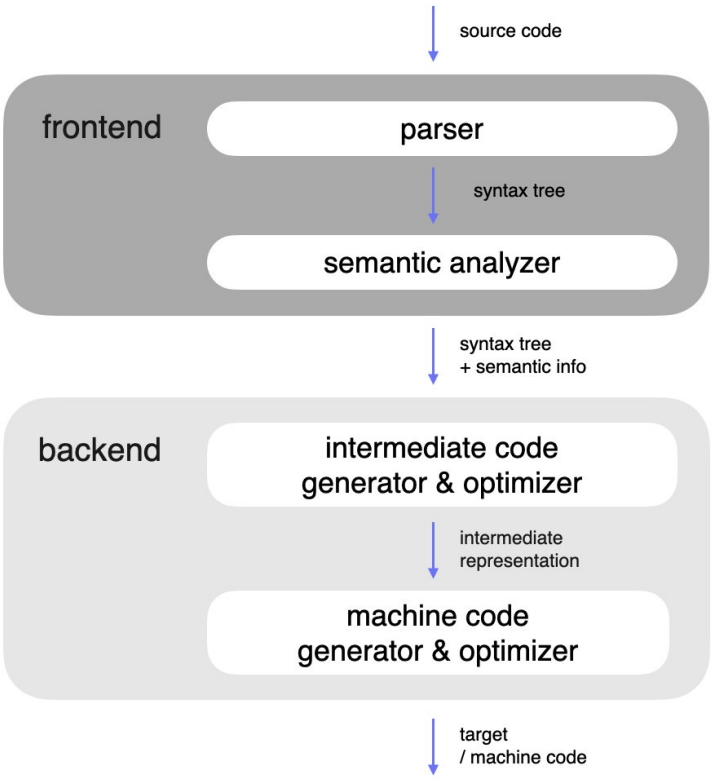
Pro zajištění interoperability mezi Kotlinem a jazykem C, slouží nástroj `cinterop`, který z hlavičkových souborů generuje Kotlin wrappery, které je možné v našem kódu použít. Pro volání systémových a některých dalších knihoven obsahuje Kotlin/Native *platformní knihovny*, připravené k použití bez nutnosti generování wrapperů. Jde například o knihovny POSIX (pro GNU/Linux), Win32 (pro Windows), Metal a Foundation (pro iOS), OpenGL nebo gzip. [81, 85]

Pro určení knihovny, pro kterou chceme nástrojem `cinterop` vygenerovat mapování na jazyk Kotlin, slouží soubor `.def`, ve kterém definujeme knihovnu a její hlavičkové soubory. Výsledkem generování jsou zkompilované wrappery v souboru `.klib` a zdrojové soubory, díky kterým může IDE například našeptávat návrhy. Statické knihovny jsou do souboru `.klib` přibaleny, zatímco kód dynamických knihoven jeho součástí není a je tedy potřeba zajistit, aby cílové zařízení knihovnu obsahovalo. [85]

Provázání číselných typů, ať už celočíselných nebo s plovoucí desetinou čárkou, je zcela přirozené. Ukazatele a pole jsou reprezentovány pomocí třídy `CPointer<T>?` — například `uint8_t*` je v Kotlin kódu reprezentován jako `CPointer<int_8tVar>`. Výjimkou jsou ukazatele `const char*`, které jsou reprezentovány pomocí Kotlin řetězců (`String`). Součástí Kotlin/Native je navíc pro snadný převod také rozšíření této třídy o atribut s C řetězcem a naopak na ukazateli existuje metoda pro převod na `String`. Pro alokování nativní paměti nabízí Kotlin několik možností. Jde alokovat a uvolňovat ručně, nebo je možné paměť alokovat uvnitř bloku, který určí rozsah platnosti (viz výpis 4.3). Po vystoupení z bloku se všechna alokovaná paměť uvolní. [85]

Kotlin umožňuje volání pouze Objective-C kódu. Pokud chceme pracovat s kódem napsaným v jazyku Swift, musí být — stejně jako kdybychom ho chtěli použít v rámci Objective-C — exportovaný pomocí atributu `@objc`. Jelikož jsou všechny číselné typy Kotlinu namapované na

Obrázek 4.3 Základní rozdělení Kotlin překladače [87]

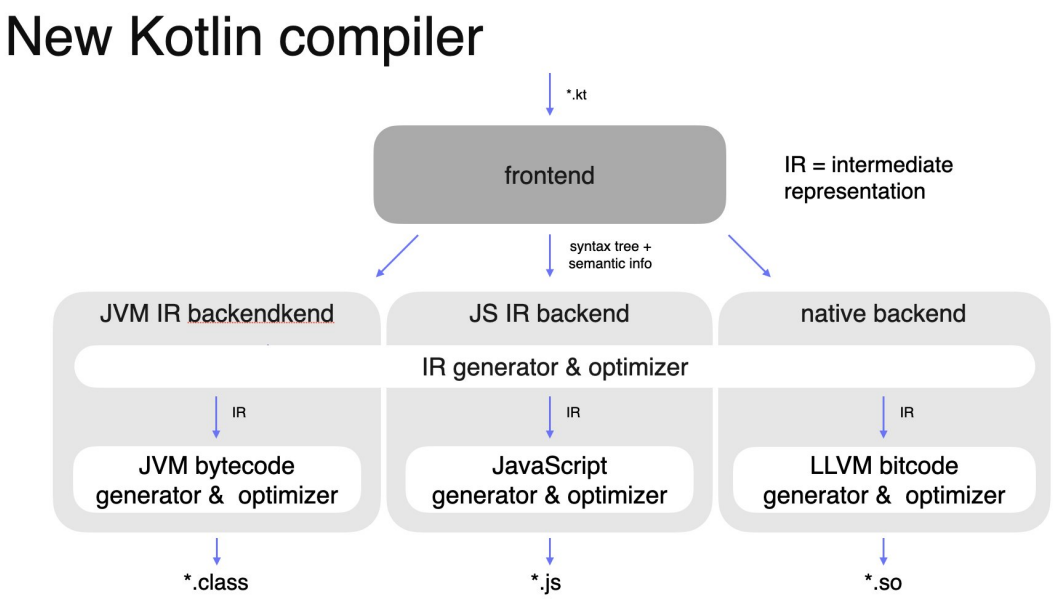


typ `NSNumber`, není možné v opačném směru rozeznat konkrétní typ a programátoři musí konverzi provést sami. Pokud chceme například volané metodě předat `Int`, musíme ho nejprve standardní Kotlin notací převést (`42 as NSNumber`). Dalším omezením je, v tomto směru provázání jazyků, dědění tříd. Kotlin umožňuje dědit Swift třídy, respektive Objective-C protokoly, pouze pomocí final tříd, které už dál nelze dědit. Oproti tomu v jazyku Swift je možné vytvořit celou hierarchii tříd založených na Kotlin prarodiči. [86]

4.3 Kotlin překladač

Kompilátor (také překladač) je počítačový program, jehož úkolem je přeložit zdrojový kód napsaný v jednom programovacím jazyku do kódu v jiném jazyku. Kompilátor se skládá z několika částí. Základní, vysokoúrovňové, rozdělení je na frontend a backend, mezi kterými je případně ještě optimalizátor. Každá z částí se sama dělí na menší díly, kterými se zde zabývat nebudeme. Cílem frontendu je rozpoznání a porozumění vstupnímu jazyku. Tato část kontroluje, zda je vstupní kód validní. Backend má za úkol vytvořit výstup překladače, například strojový kód, nebo bytekód. Aby bylo možné navázat backend na frontend, je potřeba, aby si mezi sebou v nějaké vnitřní reprezentaci předaly kód, se kterým pracují. K tomu často slouží mezikód, což může být kód podobný assembleru, nebo kód reprezentovaný stromem. Jde o interní reprezentaci kódu, s kterou pracuje optimalizátor, který mezikód analyzuje a upravuje. Z mezikódu vychází také generátor výstupního kódu. Toto rozdělení překladače a používání mezikódu umožňuje vytvořit

Obrázek 4.4 Nový Kotlin překladač [89]



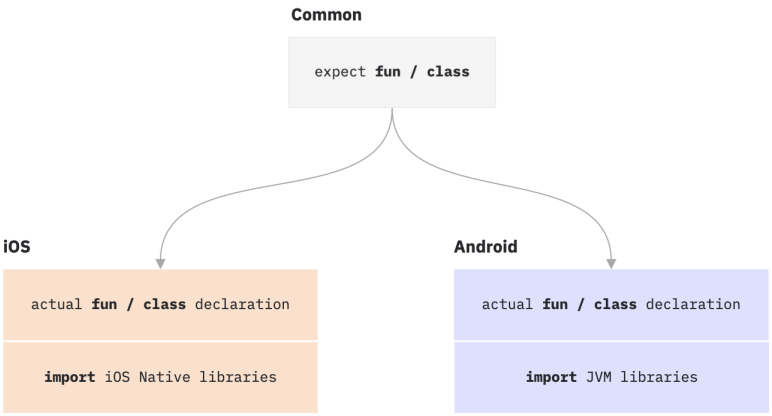
překladač pro jinou cílovou platformu změnou jen jeho části, konkrétně backendu. Přesně na tomto principu pracuje překladač Kotlinu. Alternativně je možné cílit na jednu platformu více zdrojovými kódy. V takovém případě má každý z nich samostatný frontend, ze kterého vystupují data ve stejném mezikódu. [88]

Vývojáři Kotlin kompilátoru ho dělí jen na dvě části a optimalizaci řadí do backend části překladu. Na diagramu 4.3 vidíme, že jsou data mezi frontendem a backendem reprezentována pomocí syntaktického stromu a tabulky se sémantickými informacemi. Přestože se technicky jedná o typ mezikódu (interní reprezentaci), Kotlin vývojáři tak nazývají až mezikód generovaný v backend fázi kompilace. Tento mezikód má také stromovou strukturu. [88, 59]

První verze Kotlin překladače spolu sdílely pouze frontend. Backend měly zcela samostatný a při generování výstupního kódu vycházely přímo ze syntaktického stromu (a sémantických informací). Mezikód (IR) začal používat nejdříve backend pro Kotlin/Native a vývojáři Kotlinu od té doby pracují na nových backendech i pro zbylé dvě platformy. Toto úsilí označují jako IR kompilátor pro Kotlin/JVM, respektive JS, a dohromady s novým frontendem hovoří o K2 kompilátoru. Důvodem k implementaci nových IR backendů je možnost sdílení jejich části — konkrétně optimalizace a rozšíření (pluginů) pro překladač — mezi všemi platformními variantami. Tuto skutečnost zachycuje obrázek 4.4. [59]

Nové IR kompilátory (pro Kotlin/JVM a Kotlin/JS) se poprvé objevují ve verzi 1.4.0 (vydané 2020-08-17). Ve verzi 1.4.30 dosáhl IR kompilátor pro Kotlin/JVM beta verze a od Kotlinu 1.5.0 (2021-05-05) jde o výchozí backend. Původní Kotlin/JVM backend byl pak ve verzi 1.8.0 zcela odstraněn. IR backend pro Kotlin/JS dosáhl beta statusu ve verzi 1.5.30 (2021-08-23) a od verze 1.8.0 (2022-12-28) je stabilní a používán jako výchozí. Stav nového kompilátoru v tuto chvíli zachycuje diagram 4.4 a vývoj probíhá především na novém frontendu, který je nyní v alfa verzi a zatím podporuje jen JVM a nově také částečně JavaScript projekty. Měl by mimo jiné výrazně urychlit čas kompilace a JetBrains ho považuje za natolik zásadní změnu, že s jeho stabilním vydáním zvýší verzi Kotlinu na 2.0. [90, 91, 75, 92, 59]

Obrázek 4.5 Výrazy `expect` a `actual` [93]



4.4 Sdílení kódu mezi platformami

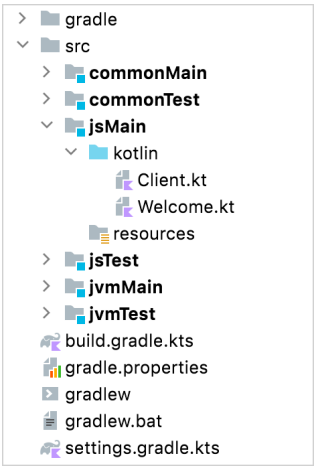
Nyní, když jsme si popsali, jak dokáže Kotlin cílit na jednotlivé platformy, se přesuneme k tématu sdílení kódu pro více než jednu z těchto platforem. Obecně to ukazuje obrázek 4.1, kdy kód napsaný v takzvaném Common Kotlinu, tedy společném nebo obecném Kotlinu, je možné přeložit pro všechny tři cílové platformy. Jde o obecné výrazy, které nejsou specifické jen pro jednu z nich. Pokud je potřeba sdílet funkcionalitu, která obsahuje platformě závislý kód, je potřeba napsat tyto části programu odděleně a jednotně se na ně ve společném kódu odkazovat. Toho lze docílit buď přímo pomocí výrazů `expect` a `actual`, nebo nepřímo využitím multiplatformních knihoven. [43]

Klíčové slovo `expect` slouží k definování očekávaného rozhraní a neobsahuje konkrétní implementaci. K tomu slouží klíčové slovo `actual`, které označuje implementace pro konkrétní platformy. Jak ukazuje diagram 4.5, každý výraz `expect` má na sebe navázáno několik implementací. Kód je v multiplatformním projektu rozdělen do složek podle jednotlivých platforem (viz obrázek 4.6), takže například výraz `actual` ve složce `androidApp/` obsahuje implementaci pro Android a výraz ve složce `iosApp/` obsahuje implementaci pro iOS. Takovýmto způsobem se dá implementovat většina deklarací, které Kotlin nabízí (především funkce, třídy nebo rozhraní). Tyto očekávané deklarace se pak dají volat ve společném kódu a při kompilaci se za ně pro každou platformu dosadí konkrétní implementace. [94]

Zabalením konkrétní funkcionality s využitím výrazů `expect` a `actual` dostaneme multiplatformní knihovnu. Použití knihoven na ně umožňuje delegovat část funkcionality využitím jejich rozhraní. Multiplatformní knihovny je možné použít ve společných částech kódu a řeší platformně specifické problémy za nás. Takové knihovny se mohou soustředit pouze na mobilní platformy, nebo implementovat funkcionalitu zároveň i pro JVM, JavaScript, Windows, macOS a GNU/Linux, případně jakoukoliv jejich kombinaci. Technologie Kotlin Multiplatform nevylučuje použití „obyčejných“ knihoven. Jediné v čem nám zabrání, je jejich volání ve společném kódu. Je potřeba je abstrahovat do rozhraní — nebo jejich funkcionalitu jinak oddělit — a to následně implementovat pro různé platformy různými knihovnami. Mezi oficiální Kotlin knihovny patří Ktor (pro tvorbu webových klientských i serverových aplikací), Coroutines (pro asynchronní programování), Serialization nebo DateTime. [43, 96, 97]

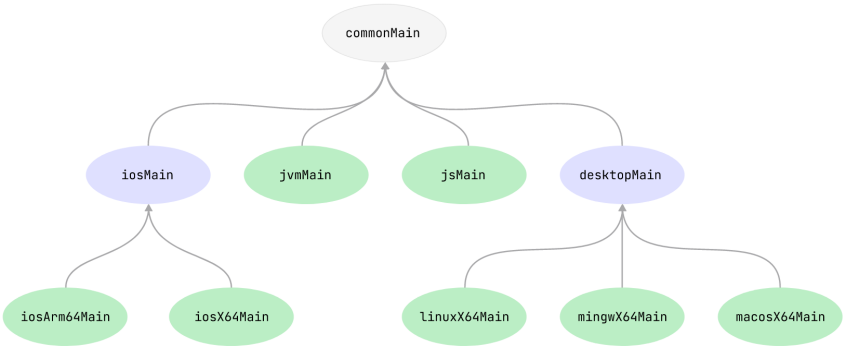
Jak jsme si již ukázali, Kotlin Multiplatform rozděluje kód na společný a platformě specifický. Toto rozdělení ale může mít i více vrstev, kdy budeme například sdílet část kódu jen pro nativní desktop. Můžeme tak oddělit části společné pro všechny platformy (například web a desktop) a části pro podmnožinou platforem (například Windows, macOS a GNU/Linux). Příklad takové

Obrázek 4.6 Rozdělení zdrojového kódu v multiplatformním projektu [95]



struktury najdeme na obrázku 4.7. Tato hierarchie ale samozřejmě není závazná a je možné ji nastavit podle potřeb konkrétního projektu. Jediné, co Kotlin v toto chvíli neumožňuje, je sdílet kód mezi více JVM výstupy, respektive mezi JVM a Androidem, nebo mezi více JavaScriptovými výstupy. [96]

Obrázek 4.7 Sdílení kódu mezi platformami [98]

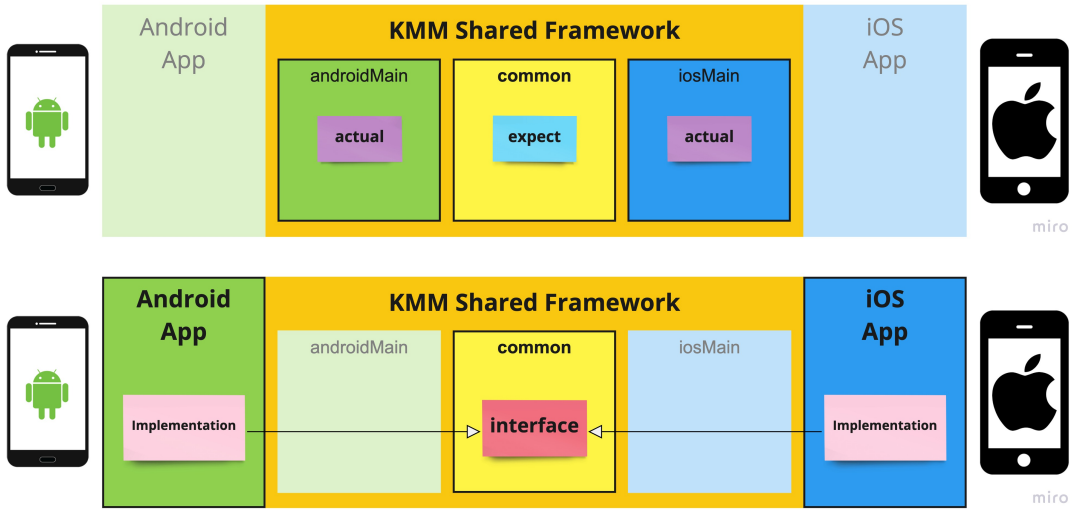


4.4.1 Sdílení kódu s platformou iOS

Jelikož je Kotlin mnohem blíže vývoji pro Android, než pro iOS, je dobré mít na paměti, že spolupráce s iOS knihovnami a především s jazykem Swift může být problematická. Kotlin/Native interoperabilita je zaměřená primárně na jazyk Objective-C. Jazyk Swift zatím není podporován přímo z Kotlinu, ale skrze právě Objective-C. To může vést k nespokojenosti iOS vývojářů, kteří používají Swift. Jednou věcí je přechod ke Kotlinu, druhou je využívání jiného API. V případě psaní platformního kódu se sice používají iOS rozhraní, jsou ale založená na Objective-C variantách. [99]

Jednou z možností, jak umožnit iOS vývojářům používat pro implementaci platformně závislých částí kódu Swift, je vyhnout se konstruktu `expect/actual`. V některých jednoduchých případech nemusí být rozdíl moc patrný a nepůjde tedy o velký problém. Jakmile se však přesuneme od víceméně jednořádkových metod ke složitějšímu použití, můžeme narazit na problémy.

Obrázek 4.8 Rozdílné přístupy pro implementaci platformně specifického kódu [101, 102]



Frustraci z využívání Objective-C rozhraní pocítí především Swift vývojáři a konkrétní příklad popsal na svém blogu Derek Lee. [99, 100]

Jako řešení popisuje nahrazení `expect/actual` za „obyčejné“ rozhraní (definované v common modulu), které implementujeme až v kódu iOS aplikace — ve Swiftu. Jediným úskalím je, že Kotlin za nás platformní `actual` implementaci sám dosazuje, takže v případě rozhraní budeme muset vkládání vhodných závislostí zajistit sami. Nejlepším řešením by ale nejspíše bylo použití některé z multiplatformních knihoven, jejichž množství se snad s nárůstem popularity technologie Kotlin Multiplatform postupně zvýší. Právě tvorba multiplatformních knihoven je jediným příkladem použití konstruktu `expect/actual`, který Derek Lee doporučuje. [99, 100]

Pokud implementujeme společný kód, který budeme ve Swiftu dál používat, měli bychom si dávat pozor na několik úskalí, které s sebou použití technologie Kotlin/Native přináší. Výše jsme si popsali možnost použití rozhraní pro implementaci platformně specifických částí. Dokud budeme rozhraní používat pouze z Kotlinu, vše bude fungovat v pořádku a nejspíš nenarazíme na žádný problém. Pokud bychom však chtěli pracovat s Kotlin rozhraními ve Swiftu, nebude možné vygenerovaný `protocol` porovnávat. Pro porovnání je možné ho převést na `NSObject`, ale lepším řešením je v Kotlinu namísto rozhraní definovat abstraktní třídu. Všechny třídy, i ty abstraktní, při převodu do Swiftu, respektive Objective-C, mapují metodu `equals()` (a další) z `kotlin.Any` na metodu `isEquals()`. Zároveň také ve Swiftu implementují protokoly `Equatable` a `Hashable`. [103]

Dalším problémem jsou generické typy, jejichž možnosti použití jsou v Kotlin/Native zatím omezené. Pro jejich dobrý převod do Swiftu je navíc občas potřeba aplikovat triky typu nahrazení `ClazzName<T>` za `ClazzName<T: Any>`, abychom vynutili nenulovost generického typu. V neposlední řadě je dobrou praktikou vyhnout se kolizi jmen, nehledě na jejich oddělení v rámci Kotlin balíčků. Kotlin/Native je převést dokáže, ale nebude snadné je mezi sebou rozeznat. Na některá z těchto doporučení možná budeme moci díky dalšímu vývoji technologie časem zapomenout, nicméně je vidět, že se interoperabilita Kotlinu a Swiftu neobejde bez problémů. Měli bychom se proto při používání Kotlin Multiplatform snažit co nejvíce kódu sdílet a z jednotlivých platforem s nám pracovat co nejjednodušším způsobem. [103]

Kapitola 5
Prototyp

V této kapitole si popíšeme vývoj prototypu, který budeme realizovat pomocí technologie Kotlin Multiplatform Mobile. Cílem této aplikace bude praktické prozkoumání použití technologie KMM, nikoliv vývoj plnohodnotné aplikace. Z tohoto důvodu vychází její označení prototyp, a také některé ústupky ve vývoji, které bychom si nemohli dovolit, pokud by měla být aplikace skutečně používána. Je také namístě zde připomenout pozorování vycházející z porovnávání některých multiplatformních frameworků v kapitole 3. Pro takto jednoduchou aplikaci cílenou na obě mobilní platformy by totiž mohlo být výhodnější, pokud pomineme zkušenosti vývojářů s danou technologií, použití frameworku Flutter, nebo React Native. Protože je ale cílem této práce především technologie Kotlin Multiplatform, využijeme právě ji.

Ve zkratce se bude jednat o mobilní aplikaci, která pomocí veřejného API s daty o pražské integrované dopravě zobrazí pro vybranou zastávku linky, které u ní zastavují. Pro daný den v týdnu a čas také zjistí, že spoj jezdí průměrně každých x minut. Pro dvě zadané zastávky aplikace vypíše linky, které je spojují.

5.1 Analýza

Tato sekce se zabývá analýzou prototypové aplikace. Jsou zde nejprve popsány funkční a nefunkční požadavky, dále případy užití a nakonec zdroj a formát dat, s kterými bude aplikace pracovat. Jak požadavky, tak i zdroj dat vychází z diskuse s vedoucím práce o vhodném rozsahu prototypu. Zásadní myšlenou bylo, aby aplikace data z nějakého webového API nejen přebírala, ale i dále zpracovávala. Tím se pak při vývoji vyzkouší použití technologie KMM jednak pro získání dat a jejich deserializaci, jednak jejich další úprava a použití obecných programovacích konstruktů pro obě platformy.

5.1.1 Požadavky

V této části jsou popsány požadavky na vytvářenou aplikaci. Funkční popisují požadavky na logiku a funkce aplikace, zatímco nefunkční určují požadavky například na použité technologie.

5.1.1.1 Funkční požadavky

F1 Aplikace pro zadanou zastávku zobrazí linky, které u ní zastavují.

F2 Aplikace najde linky, které spojují dvě zadané zastávky.

F3 Aplikace pro zadanou zastávku, den a čas vypočte průměrnou dobu mezi jednotlivými spoji.

5.1.1.2 Nefunkční požadavky

NF1 Aplikace bude vyvinutá pomocí technologie Kotlin Multiplatform Mobile.

NF2 Logika aplikace bude společná pro obě mobilní platformy (Android a iOS).

NF3 Bude vytvořená aplikace pro Android se základním uživatelským rozhraním.

NF4 Bude otestována možnost využití společné části pro systém iOS.

NF5 Aplikace bude zobrazovat aktuální data.

NF6 Aplikace bude získávat data z veřejného API datové platformy Golemio [104].

5.1.2 Případy užití

Tato sekce obsahuje čtyři případy užití — jejich popis a scénáře. Abychom si jednoduše ověřili, že jsme nezapomněli na žádnou funkcionalitu, namapujeme případy užití na jednotlivé funkční požadavky. Na tabulce 5.1 vidíme, že každý z funkčních požadavků je realizován alespoň jedním případem užití.

Tabulka 5.1 Mapování případů užití na funkční požadavky.

	UC1	UC2	UC3	UC4
F1	✓	✓		
F2	✓		✓	✓
F3	✓	✓		✓

Aktéři
Tato jednoduchá aplikace má jen jednoho aktéra — jakéhokoliv uživatele aplikace. Nerozlišuje se mizi nimi žádnou formou autentizace ani autorizace.

5.1.2.1 UC1: Vyhledat odjezdovou zastávku

Už při vyhledávání zastávky se projevuje fakt, že navrhujeme prototyp. Vyhledání na základě přesného — až na diakritiku a velikost písmen — jména zastávky by jsme si ve skutečné aplikaci nemohli dovolit. Lepší forma vyhledávání, například podle části jména zastávky, by se však projevila především při vývoji uživatelského rozhraní, které není významným cílem tohoto prototypu.

Popis
Uživatel očekává možnost zadat název zastávky, kterou chce vyhledat — zastávka na které se chystá nastoupit do dopravního prostředku. Po vyhledání zastávky uživatel očekává, že uvidí seznam linek, které ze zastávky vyjíždějí, a pro každou z nich navíc průměrnou dobu mezi odjezdy jejich spojů — průměrný interval odjezdů.

Hlavní scénář

1. Aplikace zobrazí vyhledávací pole pro zadání názvu zastávky a tlačítko pro vyhledání zastávky a jejích linek.
2. Uživatel vyplní název zastávky a klikne na tlačítko pro vyhledání.
3. Aplikace podle zadaného názvu vyhledá zastávku.

4. Aplikace zastávku nalezne.
5. Aplikace začne vyhledávat linky — případ užití **UC2**, případně **UC4**.

Alternativní scénář — zastávka nenalezana
Navazuje na třetí bod hlavního scénáře.

4. Aplikace zastávku nenalezne a oznámí to uživateli.

Alternativní scénář — prázdný název zastávky
Navazuje na první bod hlavního scénáře.

2. Uživatel název zastávky nevyplní a pouze stiskne tlačítko pro vyhledávání.
3. Aplikace uživatele upozorní, že pro vyhledání zastávky musí zadat její název.

5.1.2.2 UC2: Vyhledat linky, které staví na vybrané zastávce

Tento případ užití navazuje na vyhledání odjezdové zastávky (**UC1**). Proběhne, pokud byla úspěšně nalezena odjezdová zastávka, ale cílová zastávka nebyla zadána/nalezena (**UC3**).

Popis
Uživatel očekává, že uvidí seznam linek, které staví na vybrané zastávce, a pro každou z nich navíc průměrnou dobu mezi odjezdy jejich spojů — průměrný interval odjezdů.

Hlavní scénář
Navazuje na čtvrtý bod hlavního scénáře **UC1**.

4. Aplikace zastávku nalezne.
5. Aplikace začne vyhledávat linky, které ze zastávky v den a čase vyhledávání jezdí.
6. Aplikace v průběhu vyhledávání uživatele informuje, že vyhledávání probíhá.
7. Aplikace zobrazí seznam linek a jejich průměrné intervaly odjezdu.

Alternativní scénář — prázdný seznam linek
Navazuje na šestý bod hlavního scénáře.

7. Aplikace nenalezne žádné linky, které by ze zastávky v následující hodině odjížděly.
8. Aplikace uživatele informuje, že žádné linky nenalezla.

Alternativní scénář — chyba při stahování dat
Navazuje na šestý bod hlavního scénáře.

7. Během získávání dat z webového API nastane chyba.
8. Aplikace uživatele o chybě informuje.

5.1.2.3 UC3: Vyhledat cílovou zastávku

Popis
Uživatel očekává možnost zadat název zastávky, kterou chce vyhledat — zastávka do které je možné se dostat přímým spojem z odjezdové zastávky. Po vyhledání zastávky uživatel očekává, že uvidí seznam linek, které spojují odjezdovou a tuto cílovou zastávku, a pro každou linku navíc průměrný interval odjezdů.

Hlavní scénář

1. Aplikace zobrazí vyhledávací pole pro zadání názvu zastávky a tlačítko pro vyhledání zastávky a jejích linek.
2. Uživatel vyplní název zastávky a klikne na tlačítko pro vyhledání.
3. Aplikace podle zadaného názvu vyhledá zastávku.
4. Aplikace zastávku nalezne.
5. Aplikace začne vyhledávat linky — případ užití **UC4**.

Alternativní scénář — zastávka nenalezana
Navazuje na třetí bod hlavního scénáře.

4. Aplikace zastávku nenalezne a oznámí to uživateli.

5.1.2.4 UC4: Vyhledat linky, které spojují dvě zastávky

Tento případ užití navazuje na vyhledání odjezdové a cílové zastávky (**UC1** a **UC3**). Proběhne, pokud byla úspěšně nalezena odjezdová i cílová zastávka.

Popis
Uživatel očekává, že uvidí seznam linek, které přímo spojují odjezdovou a cílovou zastávku, a pro každou z linek navíc průměrný interval odjezdů.

Hlavní scénář
Navazuje na čtvrtý bod hlavního scénáře **UC1**/**UC3**.

4. Aplikace zastávku nalezne.
5. Aplikace začne vyhledávat linky, které z odjezdové zastávky v den a čase vyhledávání jezdí a navíc spojují odjezdovou a cílovou zastávku.
6. Aplikace v průběhu vyhledávání uživatele informuje, že vyhledávání probíhá.
7. Aplikace zobrazí seznam linek a jejich průměrné intervaly odjezdu.

Alternativní scénář — prázdný seznam linek
Navazuje na šestý bod hlavního scénáře.

7. Aplikace nenalezne žádné linky, které by ze zastávky v následující hodině odjížděly a spojovaly odjezdovou a cílovou zastávku.
8. Aplikace uživatele informuje, že žádné linky nenalezla.

Alternativní scénář — chyba při stahování dat
Navazuje na šestý bod hlavního scénáře.

7. Během získávání dat z webového API nastane chyba.
8. Aplikace uživatele o chybě informuje.

5.1.3 Zdroj dat

Data o zastávkách, linkách a jízdních řádech metra, tramvají, městských autobusů a dalších vozidel Pražské integrované dopravy (PID) vystavuje ROPID (Regionální organizátor Pražské integrované dopravy). Veřejně přístupná jsou od roku 2013, z počátku však jen na vyžádání. V současnosti je možné je stáhnout na stránkách PIDu pod creative commons licencí CC-BY. Data jsou zveřejňována dle specifikace GTFS jako zip archiv obsahující textové CSV soubory. Jdou denně aktualizovaná, nicméně by měla být platná pro následujících přibližně 12 dní. V rámci této práce budeme používat API datové platformy Golemio, která data od ROPIDu přebírá a vystavuje je ve formě RESTful API. [105, 104, 106]

5.1.3.1 Formát GTFS

General Transit Feed Specification je formát, který vytvořili v roce 2005 společnosti TriMet a Google za účelem zveřejňování informací o hromadné dopravě a možnosti rozšíření aplikace Mapy Google o tyto informace. Specifikaci začali používat i další provozovatelé hromadné dopravy, díky čemuž je možné vytvářet univerzální aplikace pro jízdní řády, plánování tras, vizualizace a podobně. [107]

Specifikace je rozdělena na dvě části — GTFS Schedule a GTFS Realtime. Nás zajímá především první část, která obsahuje zastávky, trasy, a jízdní řády. Druhá část, GTFS Realtime, se zaměřuje na data o poloze vozů a odhadovaná zpoždění. [108]

Specifikace GTFS Schedule určuje následující povinné soubory:

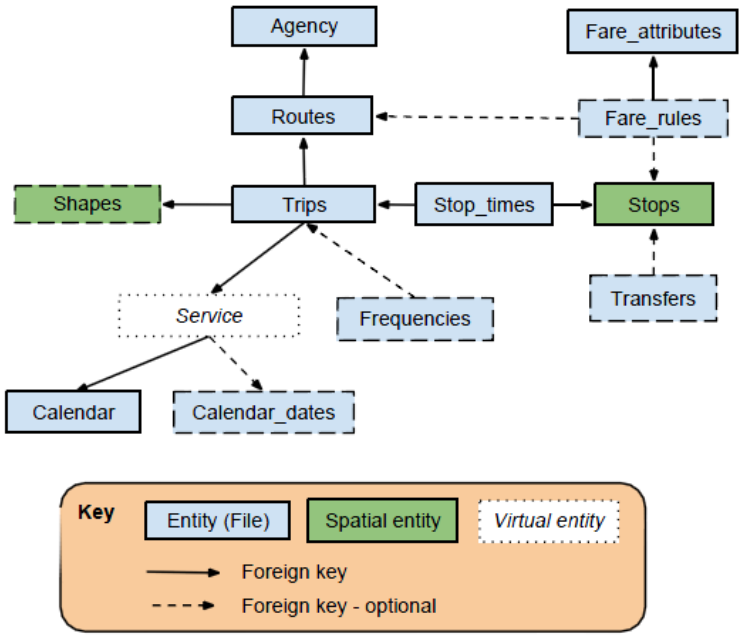
- `agency.txt` (dopravci),
- `stops.txt` (zastávky),
- `routes.txt` (linky),
- `trips.txt` (spoje linek),
- `stop_times.txt` (časy příjezdů a odjezdů spojů k zastávkám),
- `callendar.txt` nebo `calendar_dates.txt` (určují dny, kdy spoje jezdí).

Vztahy mezi nimi můžeme vidět na diagramu 5.1. V případě dat od ROPIDu je dopravce jen jeden společný. Informace o tom, kteří smluvní dopravci provozují které linky, zaznamenává ROPID v samostatném souboru `route_sub_agencies.txt`. Příklady linek (`routes`) jsou například tramvaj 18 (Nádraží Podbaba — Vozovna Pankrác) nebo autobus 107 (Dejvická — Suchdolské náměstí — Suchdol). Linka má krátký a dlouhý název, provozuje ji určitý dopravce a může se jednat o noční, regionální nebo náhradní linku. Konkrétní spoj (`trip`) linky udává směr (například tramvaj 18 ve směru Nádraží Podbaba), které dny jezdí (například víkendový spoj), a zda má bezbariérový přístup či povoluje převážet jízdní kola. Zastávky (`stops`) mají především jméno, zeměpisné souřadnice (dle standardu WGS84), pásmo nebo bezbariérovost. Samotné jízdní řády určují data ze `stop_times.txt`, které obsahují ID stanice a spoje, časy příjezdu a odjezdu a také pořadí zastávky na trase spoje. Specifikace obsahuje i další, nepovinné struktury. Například pomocí `shapes.txt` je možné zjistit trasu spoje s větší přesností, než jen výčet zastávek. Tvar trasy je zadaný v podstatě pomocí uspořádané n-tice zeměpisných souřadnic. [109, 105]

5.1.3.2 Golemio API

Golemio vystavuje na adrese `https://api.golemio.cz/v2/gtfs/` RESTful API s částí dat od ROPIDu. Data jsou z CSV formátu transformovaná do formátu JSON, nicméně hodnoty stále odpovídají specifikaci GTFS. Ukázková data tramvajové linky č. 18 v obou formátech můžeme vidět ve výpisech 5.1 a 5.2. Pro tuto práci jsou významné především endpointy

Obrázek 5.1 Diagram entit specifikace GTFS Schedule [110]



- `/gtfs/stops` — umožňující filtrování podle jména nebo ID zastávky —,
- `/gtfs/trips` — dává možnost filtrovat podle ID zastávky nebo data —,
- `/gtfs/routes/{id}`,
- `/gtfs/stoptimes/{id}` — umožňující získání jízdního řádu pro konkrétní zastávku s možností filtrování podle data a časového intervalu. [104]

Abychom mohli API využívat, musíme v hlavičce HTTP dozadu poslat API klíč. Je možné si ho vygenerovat po založení *Golemio API Keys Management* účtu. Každý takový klíč má nastavený limit 10 000 dotazů za 10 sekund. Příkladem dotazu, jehož výsledkem je JSON 5.1 je

```
curl -H 'X-Access-Token: xXxXxXx' https://api.golemio.cz/v2/gtfs/routes/L18.
```

Další vlastnost API, kterou je potřeba zmínit, je stránkování výsledků. Pro dotazy vracející seznam objektů lze omezit množství výsledků a určit posun od první položky. Maximální množství výsledků je 10 000.

5.2 Návrh

Tato sekce se zabývá návrhem aplikace, konkrétně Android aplikace a společné části sdílené oběma platformami. Z nefunkčních požadavků (konkrétně **NF4**) víme, že nebudeme vyvíjet aplikaci pro operační systém iOS, pouze otestujeme, že je možné společnou část v iOS využít. V sekci Android aplikace si popíšeme návrh uživatelského rozhraní a architekturu celé aplikace. Pro společný kód si připomeneme, jakým způsobem zajistit, aby se dal sdílet mezi platformami a navrhneme rozdělení kódu do tříd.

Výpis kódu 5.1 Tramvajová linka č. 18 ve formátu JSON

```
{
  "agency_id": "99",
  "is_night": false,
  "is_regional": false,
  "is_substitute_transport": false,
  "route_color": "7A0603",
  "route_desc": null,
  "route_id": "L18",
  "route_long_name": "Nádraží Podbaba - Vozovna Pankrác",
  "route_short_name": "18",
  "route_text_color": "FFFFFF",
  "route_type": "0",
  "route_url": "https://pid.cz/linka/18"
}
```

Výpis kódu 5.2 Tramvajová linka č. 18 ve formátu CSV (zkráceno)

```
route_id,agency_id,route_short_name,route_long_name,route_type,route_url,
L18,99,18,"Nádraží Podbaba - Vozovna Pankrác",0,"https://pid.cz/linka/18",
```

5.2.1 Android aplikace

Začneme návrhem Android aplikace, kterou budeme programovat v jazyku Kotlin (dle **NF1**). Ačkoliv jde o prototyp a nemáme tedy důvod podporovat co nejširší spektrum zařízení, zvolíme minimální verzi SDK API 23, což odpovídá Androidu 6.0. Aplikace tak bude podporovat více než 97 % zařízení.

5.2.1.1 Uživatelské rozhraní

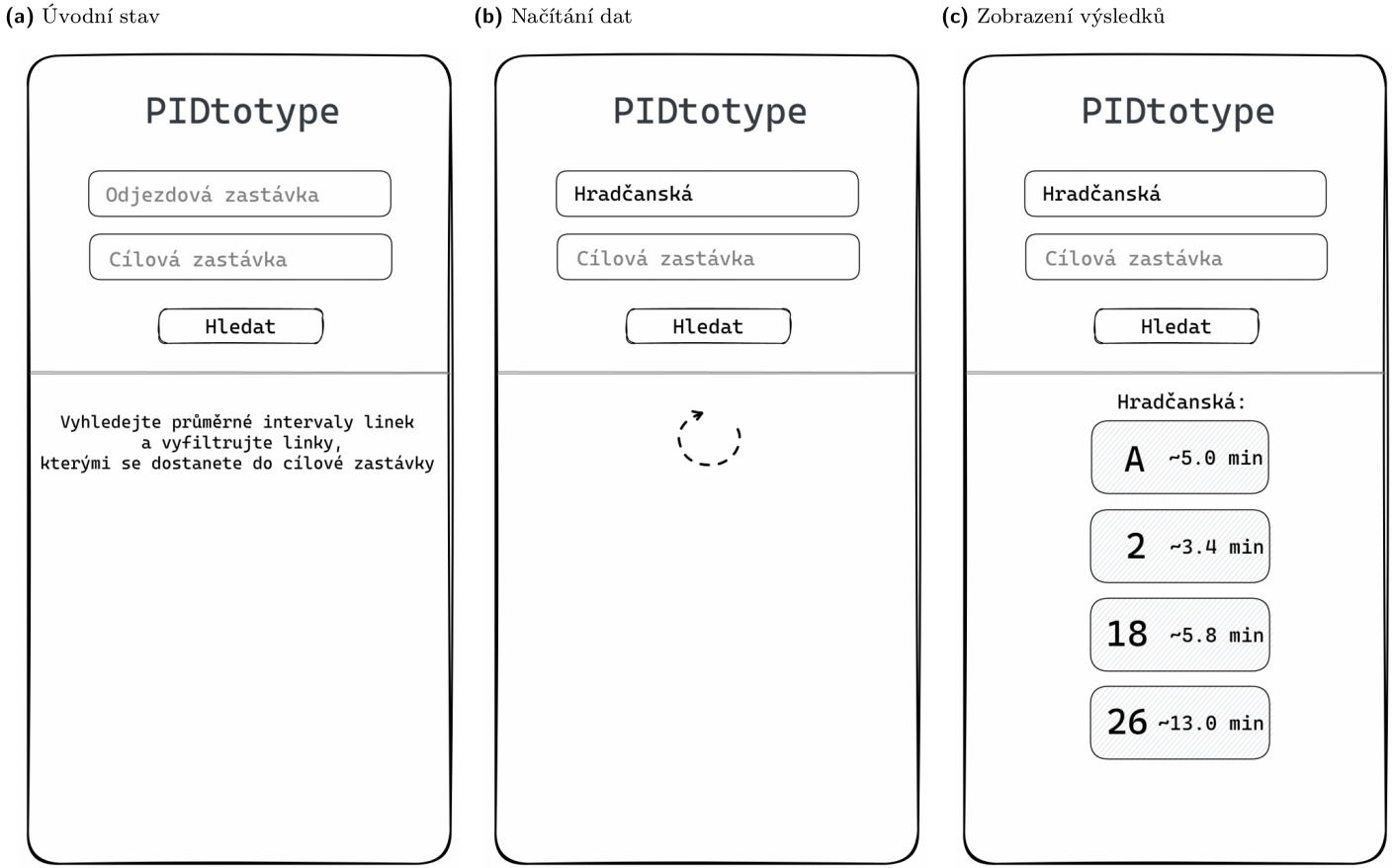
Nyní se zaměříme na návrh uživatelského rozhraní. Ačkoliv by byl návrh UI normálně významnou částí procesu návrhu jakékoliv aplikace, cílem našeho prototypu není vytvořit uživatelsky přívětivou a hezky vypadající aplikaci. Cílem je demonstrovat využití technologie Kotlin Multiplatform Mobile při vývoji mobilní aplikace.

Wireframes
Uživatelské rozhraní prototypu si vizualizujeme pomocí drátěných modelů — wireframes. Ty slouží k návrhu rozložení prvků na obrazovce tak, aby umožňovali co nejlépe splnit případy užití a dosáhnout požadovaného výsledku/cíle. Wireframe nemá za cíl zachytit přesný finální vzhled včetně obrázků, barev, fontů a dalších designových prvků, pouze rozložení funkčních a informativních prvků na obrazovce tak, aby jsme měli představu o tom, jak budou uživatelé s aplikací interagovat, nikoliv jak bude vypadat.

Případy užití a jejich scénáře ze sekce 5.1.2 můžeme shrnout do dvou fází. První fází při zjišťování intervalů linek je zadání názvu odjezdové a případně i cílové zastávky. Druhou fází je vyhledání zastávek a jim příslušných linek a jejich následné zobrazení uživateli. Aplikace je tedy dostatečně jednoduchá na to, aby bylo možné veškerou funkcionalitu promítnout do jedné obrazovky.

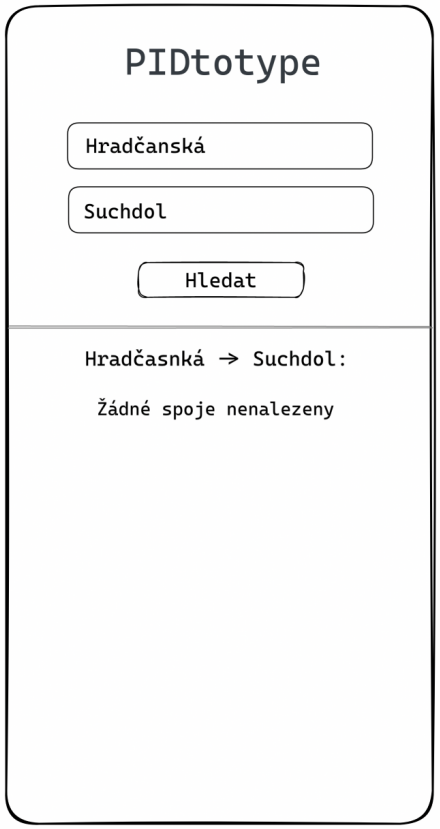
Obrazovku rozdělíme na dvě poloviny. Horní bude sloužit pro výše zmíněnou první fázi a bude obsahovat textová pole pro zadání názvů zastávek a tlačítko pro vyhledání výsledků. Dolní část bude určená pro druhou fázi — indikaci načítání a zobrazení výsledků. Takové rozdělení vidíme na obrázcích 5.2 a 5.3. Stěžejní je wireframe 5.2c, na kterém vidíme seznam výsledků — linky a jejich průměrné intervaly v minutách.

Obrázek 5.2 Wireframes pro základní průchod aplikací

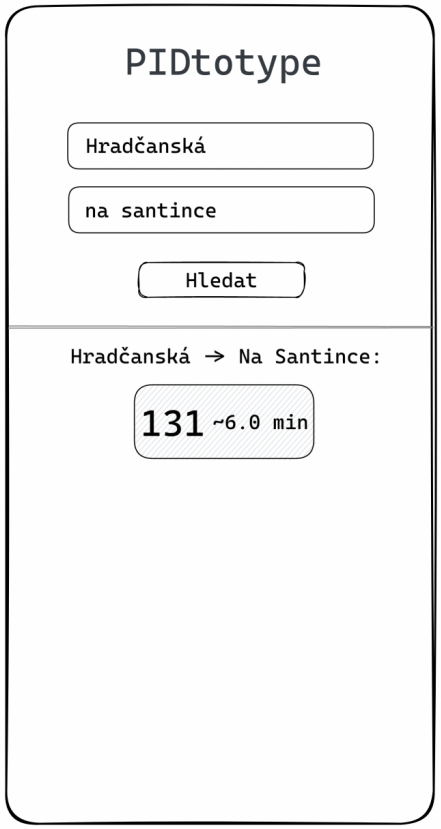


Obrázek 5.3 Dodatkové wireframes

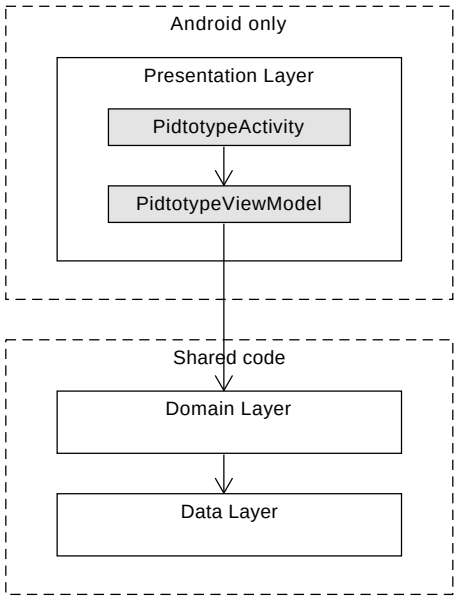
(a) Zobrazení prázdných výsledků



(b) Zobrazení linek spojující dvě zastávky



Obrázek 5.4 Diagram architektury prototypu



5.2.1.2 Architektura

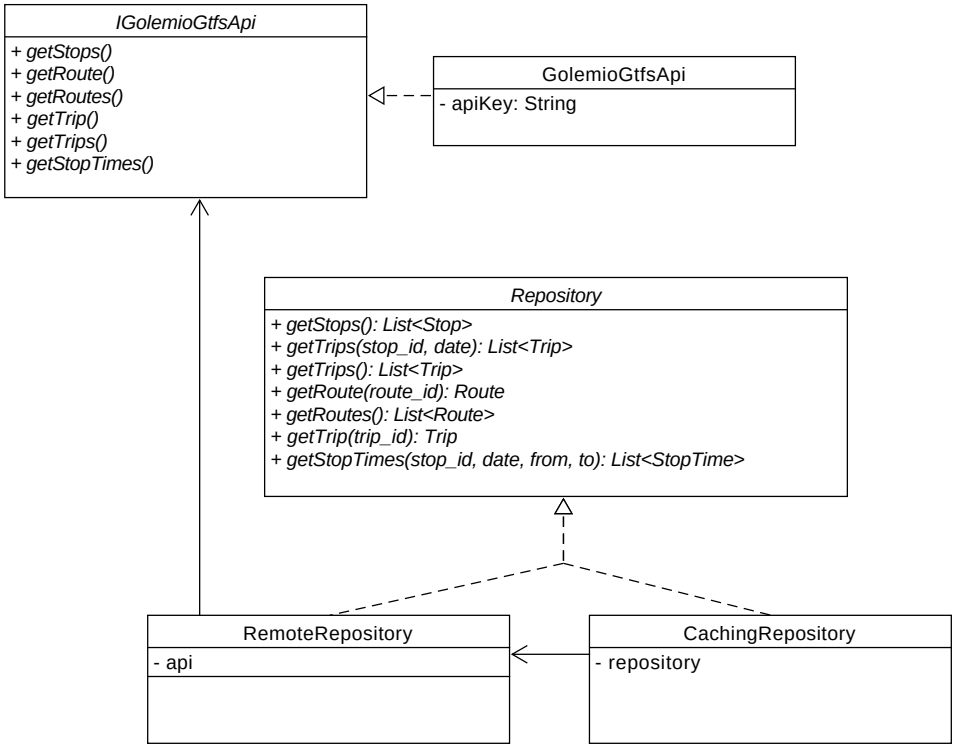
V této sekci popíšeme architekturu, kterou bude naše aplikace používat. Společnost Google vydala v Android developer dokumentaci doporučení, jak by měla architektura vypadat, aby splňovala obecně uznávané principy jako je *Separation of concerns* nebo *Unidirectional Data Flow*. Googlem doporučovaná architektura rozděluje aplikaci na vrstvy, minimálně prezentační (UI) a datovou vrstvu. Závislosti by měly být jen v jednom směru s tím, že na prezentační vrstvě není závislá žádná jiná vrstva. V zásadě se jedná o clean architekturu v kombinaci s MVVM (Model-View-ViewModel). Pro uchování dat, které aktivita potřebuje, se doporučuje použití třídy `ViewModel`. Pro datovou vrstvu se doporučuje použití repozitářů, jejichž účelem je spravování zdrojů dat, jejich abstrahování, centralizování jejich změn, řešení konfliktů, kešování a další. [111]

Architektura našeho prototypu bude podle těchto doporučení rozdělena do vrstev. Jednotlivé vrstvy a jejich rozdělení mezi společnou a čistě Android část vyobrazuje diagram 5.4. Aktivita si bude držet objekt třídy `PidtotypeViewModel` a zobrazovat jeho data uživateli. View model bude komunikovat se sdílenou částí aplikace a zprostředkovávat hledání zastávek, linek, které jimi projíždí, a intervaly odjezdů. Ponese také informaci o stavu, ve kterém se aplikace nachází. Může jít například o úvodní stav (viz 5.2a), kdy ještě není nic vyhledáno, stav načítání, stav, kdy aplikace zobrazuje seznam linek (viz 5.3b), nebo chybový stav.

5.2.2 Společná část

Zde si popíšeme návrh společné části kódu, kterou by měla sdílet aplikace pro Android i iOS. Abychom se vyvarovali nutnosti psaní platformě specifického kódu, budeme se snažit do nejvyšší míry použít takzvaný Common Kotlin — variantu popsanou v kapitole 4.4, která obsahuje konstrukce jazyka, které je možné sdílet mezi platformami. S tím také souvisí výběr knihoven, které budeme potřebovat minimálně pro komunikaci s webovým API. Měli bychom zvolit knihovny multiplatformní, které jsou pro použití ve sdíleném kódu přímo určené. V případě nutnosti lze samozřejmě použít platformně závislé varianty, znamenalo by to ale tvorbu společného rozhraní, které by pomocí konstrukce expect/actual (viz opět kapitola 4.4) delegovalo volání variantě pro

Obrázek 5.5 Class diagram repozitářů



jednu či druhou platformu. Nejen, že by jsme museli psát kód dvakrát, museli bychom pracovat s dvěma různými knihovnami, které dělají v podstatě to samé, ale mají jiné rozhraní.

Společná část bude obsahovat doménovou a datovou vrstvu, které budou zajišťovat získání dat z API, serializaci do datových entit, kešování a práci s entitami — jejich mapování, slučování, filtrování a agregaci.

5.2.2.1 Potřebná data

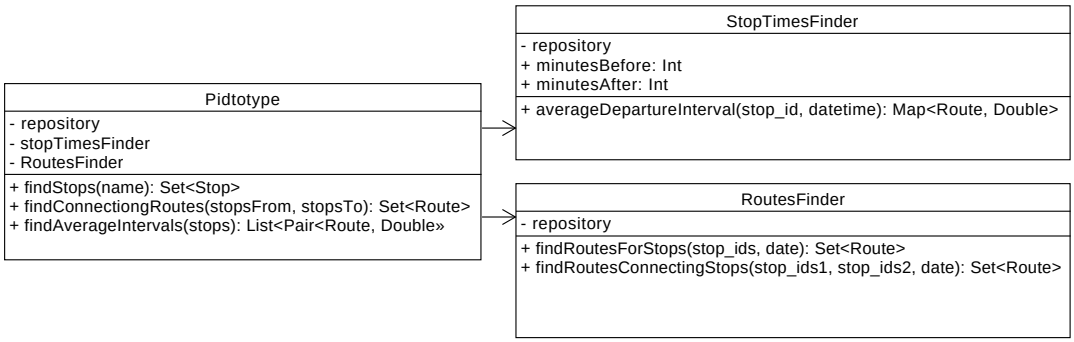
Pojďme si rychle shrnout, jaká data, která vystavuje API, budeme potřebovat pro splnění funkčních požadavků (5.1.1.1) z analýzy. Pro zpracování funkčních požadavků **F1** a **F2** budeme jistě potřebovat získat data o zastávkách (`stops`) a linkách (`routes`). Pro výpočet průměrných intervalů odjezdů spojů ze zastávky (**F3**) budeme potřebovat „jízdní řády zastávek“, tedy `stop_times`. Tyto časy se vážou na spoje (`trips`), které teprve mají informaci o lince, a proto budeme potřebovat pracovat i s nimi.

5.2.2.2 Class diagram

Pomocí class diagramu si strukturujeme kód do tříd a rozdělíme mezi ně jednotlivé odpovědnosti. Jednou velkou skupinou tříd budou datové entity, které budou pouze držet data získaná z webového API. Budou se proto strukturálně velice podobat entitám formátu GTFS, který jsme si popsali v kapitole 5.1.3.1.

Na obrázku 5.5 vidíme class diagram pro rozhraní `Repository` a `IGolemioGtfsApi`. Třída `RemoteRepository` bude implementovat rozhraní `Repository` a data získávat ze vzdáleného zdroje. `RemoteRepository` bude pro získání data závislá na rozhraní `IGolemioGtfsApi`, které

Obrázek 5.6 Class diagram finderů



bude zajišťovat samotnou webovou komunikaci s API, od které bude repozitář zcela odstíněn. Golemio API výsledky stránkuje a má horní limit 10 000 výsledků. Proto bude mít třída za úkol získat kompletní data, což bude v některých případech znamenat spojit výsledky několika dotazů.

Rozhraní `Repository` bude vystavovat metody pro získání potřebných dat k sestavení výstupu pro uživatele. Seznamy všech zastávek, spojů a linek získáme pomocí metod `getStops()`, `getTrips()` a `getRoutes()`. Na konkrétní spoj nebo linku se můžeme dotázat pomocí metody `getTrip(trip_id)`, respektive `getRoute(route_id)`. Pro získání „jízdního řádu“ zastávky bude rozhraní vystavovat metodu `getStopTimes(stop_id, date, from, to)`, která vrátí data pro vybraný datum a časový interval. Poslední metodou, kterou bude rozhraní vystavovat, bude `getTrips(stop_id, date)`, která nám vrátí spoje zastavující ve vybraný den u zvolené zastávky.

Aby se dotazy na API zbytečně neopakovaly, budeme chtít stáhnutá data kešovat. K tomu nám postačí in-memory kešování pomocí základních datových struktur Kotlinu. Bude k tomu sloužit třída `CachingRepository`, která bude také implementovat repozitář, ale bude mít na starosti kešování výsledků z jiného repozitáře — konkrétně `RemoteRepository` —, na který si bude udržovat referenci. Pro kešování tedy využijeme návrhový vzor proxy.

Samotná příprava výsledků uživatelova hledání bude probíhat ve třídách `StopTimesFinder` a `RoutesFinder`. Na diagramu 5.6 vidíme třídu `Pidtotype`, která bude zastřešovat celou funkcionalitu. Právě s ní bude komunikovat prezentační vrstva Android, respektive iOS, aplikace. Vystaví metody pro vyhledání zastávek podle jména, vyhledání průměrných intervalů linek pro skupinu zastávek a vyhledání množiny linek, které spojují dvě skupiny zastávek (skupina může obsahovat zastávky pro různě dopravní prostředky a oddělané zastávky pro jednotlivé směry jízdy). Třída `Pidtotype` také zajistí předání aktuálního data a času třídám, které samotnou tvorbu výsledků implementují.

`StopTimesFinder` bude zpracovávat časy příjezdů spojů k zastávkám — stop times. Bude je filtrovat tak, aby výsledný průměrný interval odjezdu odpovídal jen časovému úseku okolo zadaného data a času. K tomu poslouží parametry `minutesBefore` a `minutesAfter`, které určí časový rozsah — například 15 minut před a 60 minut po žádaném okamžiku. Třída `RoutesFinder` bude vyhledávat linky, které zastavují u dané zastávky/zastávek. Pomůže nám zjistit, které linky v určený den přímo spojují dvě skupiny zastávek.

5.3 Implementace

V této kapitole si popíšeme implementační detaily našeho prototypu. Představíme si nástroje a knihovny, které budeme používat, blíže si popíšeme společnou část kódu — například deserializaci, webovou komunikaci nebo práci s keší — a implementaci prezentační vrstvy Android

aplikace. Nakonec krátce zmíníme, jak se nám podařilo použít společný kódu v aplikaci pro iOS, na jaké problémy jsme při tom narazili a navrhneme jejich možná řešení.

5.3.1 Nástroje

Prototyp budeme programovat ve vývojovém prostředí Android Studio, konkrétně v aktuální verzi Electric Eel. To již v základní instalaci obsahuje plugin *Kotlin*, pro podporu tohoto programovacího jazyka. Pro podporu multiplatformního vývoje si musíme nainstalovat plugin *Kotlin Multiplatform Mobile*. Pokud bychom potřebovali psát kód pro platformu iOS, museli bychom pracovat na počítači od firmy Apple, který má operačním systémem macOS, s nainstalovaným vývojovým prostředím Xcode. [112]

Pro vyzkoušení aplikace se dá v Android Studiu použít buď virtuální zařízení běžící v emulátoru, nebo připojený fyzický smartphone. Android Emulator simuluje Android zařízení a umožňuje větší kontrolu než fyzický telefon, rychlejší spouštění a možnost širšího výběru zařízení. Nevýhodou jsou vyšší požadavky na systém, ve kterém emulátor běží. Pro spouštění naší aplikace během vývoje si vytvoříme Android virtuální zařízení emulující smartphone Pixel 2 s operačním systémem Android 11. Po otestování aplikace na reálném zařízení využijeme telefon značky Nokia s Android verzí 13. Na telefonu nejdříve odkryjeme nastavení pro vývojáře, kde musíme povolit bezdrátové ladění (alternativně také ladění přes USB). V Android Studiu pak zvolíme možnost spárování zařízení pomocí Wi-Fi.

5.3.2 Společná část

Když už jsme si popsali vývojové prostředí a možnost spuštění aplikace během vývoje, přesuneme se na samotnou implementaci. Začneme společným kódem, který tvoří hlavní část našeho prototypu. Přiblížíme si multiplatformní knihovny, které budeme používat, a detailně rozebereme sestavování výsledků, komunikaci s API, deserializaci získaných dat a jejich kešování.

5.3.2.1 Knihovny

Zde si popíšeme knihovny, které jsme použili ve společné části kódu. Všechny tyto knihovny jsou multiplatformní, podporující minimálně požadovaný Android a iOS. Pro použití knihoven v projektu je musíme přidat jako závislosti do souboru `build.gradle.kts` ve složce `shared/`. Přidání závislosti ukazuje výpis 5.3.

Ktor
Pro komunikaci s webovým API použijeme multiplatformní framework Ktor, který se dá použít k tvorbě webových aplikací, mikroservis a HTTP klientů. Jde o oficiální JetBrains framework vyvinutý speciálně pro potřeby multiplatformních aplikací. Pro asynchronicitu při webové komunikaci využívá Kotlin coroutines. [113]

Pro samotné zpracování webových dotazů používá Ktor takzvaný *engine*. Jednotlivé enginy se liší v platformách, které podporují, a mohou mít různé funkce a vlastnosti. My si zvolíme HTTP engine `CIO`, který podporuje JVM, Android i Kotlin/Native, tedy iOS. Bližší popis implementace volání API si popíšeme v kapitole 5.3.2.3.

Coroutines
Knihovna `kotlinx.coroutines` je oficiální multiplatformní řešení pro implementaci asynchronních funkcí v Kotlinu. Stejně jako Ktor jsou i coroutines vyvíjené firmou JetBrains. Knihovna umožňuje vytvořit *suspending* funkce, které jsou určené pro asynchronní běh. V naší aplikaci je budeme, spolu s frameworkem Ktor, využívat pro volání HTTP dotazů. [114]

Výpis kódu 5.3 Extrakt ze souboru `shared/build.gradle.kts`

```
plugins {
  kotlin("multiplatform")
  id("com.android.library")
  kotlin("plugin.serialization") version "1.8.10"
}
kotlin {
  // ...
  val ktorVersion = "2.2.3"
  val coroutinesVersion = "1.6.4"
  val dateTimeVersion = "0.4.0"

  sourceSets {
    val commonMain by getting { dependencies {
      implementation("io.ktor:ktor-client-core:$ktorVersion")
      implementation("io.ktor:ktor-client-cio:$ktorVersion")
      implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
      implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:
          $coroutinesVersion")
      implementation("org.jetbrains.kotlinx:kotlinx-datetime:
          $dateTimeVersion")
    }}
    // ...
  }
  // ...
}
// ...
```

Serialization
Pro deserializaci získaných JSON dat použijeme knihovnu `kotlinx.serialization`. Opět jde o oficiální JetBrains knihovnu, která zajišťuje serializaci a deserializaci dat z různých formátů — JSON, CBOR, XML, YAML, BSON a dalších. Pro použití dat z webového API je jejich deserializace do Kotlin datových tříd zcela zásadní. Deserializace si blíže popíšeme v sekci 5.3.2.4. [115]

Datetime
Pro práci s datem a časem budeme používat multiplatformní knihovnu `kotlinx.datetime` od JetBrains. Umožňuje získat aktuální datum a čas, odečíst od času x minut, nebo zjistit rozdíl dvou časových údajů ve vybrané jednotce, třeba v minutách. Třídy, které knihovna používá jsou navíc serializovatelné pomocí výše popsané `kotlinx.serialization`.

5.3.2.2 Sestavování výsledků

Zde se zaměříme na implementační detaily tříd `RoutesFinder` a `StopTimesFinder`. Slouží k sestavení výsledků ze základních dat, které jim poskytne repozitář.

Hlavní myšlenka za hledáním linek, které spojují dvě zastávky, je průnik množin linek, které u každé ze zastávek zastavují. Pro nalezení linek, které v určitý den zastavují u zastávky, musíme jít přes takové spoje. To je informace, kterou nám API, respektive repozitář, nabízí. Spoj má informaci o ID linky, což je vše co potřebujeme pro získání kompletního objektu linky. Jelikož pro jednu linku jezdí více spojů, je potřeba odstranit duplicity a máme potřebný mezivýsledek. Protože se „zastávka“ zvolená uživatelem podle jejího názvu bude ve skutečnosti skládat z více konkrétních zastávek — například pro různé směry —, spojíme nalezené linky dílčích zastávek dohromady. Není už nic jednoduššího, než získat průnik množit linek pro každou skupinu zastávek a získáme linky, které přímo spojují tyto skupiny zastávek.

Výpis kódu 5.4 Extrakt ze třídy `GolemioGtfsApi`

```
internal class GolemioGtfsApi(private val apiKey: String) : IGolemioGtfsApi
    {
  private val httpClient = HttpClient(CIO)
  private val requestBuilderTemplate = HttpRequestBuilder().apply {
    url("https://api.golemio.cz/v2/gtfs")
    header("X-Access-Token", apiKey)
  }
  private val format = Json { ignoreUnknownKeys = true }

  override suspend fun getStops(limit: Int, offset: Int): List<Stop> =
    format.decodeFromString(
      FeatureCollectionSerializer,
      httpClient.get {
        takeFrom(requestBuilderTemplate)
        url {
          appendPathSegments("stops")
          parameters.append("limit", limit.toString())
          parameters.append("offset", offset.toString())
        }
      }.bodyAsText()
    )
  // ...
}
```

Zajímavější je implementace třídy `StopTimesFinder`. Ta pro zastávku, datum a čas vyhledá odpovídající objekty typu `StopTime`. Chceme získat průměrný interval odjezdu spojů ze zastávky. Základ je vůbec získat objekty `StopTime` seskupené podle linek. Seskupení zajistí metoda `groupBy`. Jediné, co jí musíme předat, je postup, jak ze třídy `StopTime` vydolovat informaci o lince (`Route`). Musíme prostě podle ID najít spoj, z něho vzít ID linky a najít podle toho celou linku. Pro sestavení dob intervalů se krásně hodí metoda `zipWithNext`, která ze seznamu časů vytvoří dvojice sousedících časů. Z těch spočteme jejich rozdíl v minutách a tyto intervaly zprůměrujeme.

Jelikož třída zajišťuje i práci s časovým úsekem okolo zvoleného okamžiku — průměrný interval za celý den by nebyl příliš praktický —, je potřeba myslet na okrajové situace, kdy je zadaný čas na začátku/konci dne. Po spočítání okamžiků, které ohraničují zájmový úsek, musíme zkontrolovat, zda se nachází ve stejném dnu. Pokud tomu tak není, rozdělíme hledání `StopTime` dat na dobu do půlnoci a dobu po ní. Výpočet průměrného intervalu budeme provádět až na spojených datech.

5.3.2.3 Komunikace s API

Pro komunikaci využijeme framework Ktor. Výpis kódu 5.4 zachycuje část třídy `GolemioGtfsApi`, která Ktor používá. Vytvoříme si objekt `HttpClient`, který inicializujeme enginem `CIO`. Při všech dotazech budeme používat stejné API, takže část URL se bude stále opakovat. Vždy budeme také muset posílat stejný API klíč. Z toho důvodu si připravíme předlohu dotazu, který bude tyto informace obsahovat. Konkrétní dotazy pak předlohu už jen doplní. Pro získání seznamu zastávek zavoláme nad objektem `HttpClient` metodu `get` a HTTP dotaz specifikujeme — doplníme konec URL adresy a přidáme parametry `limit` a `offset`. Textovou odpověď předáme k JSON deserializaci. Protože API vrací JSON strukturu, kde je seznam zastávek zabalený do pomocného objektu, specifikujeme vlastní serializátor, který ho umí zpracovat a vrátit seznam Kotlin objektů `Stop`.

Výpis kódu 5.5 Datová třída `Coordinates`

```
@Serializable
data class Coordinates(val latitude: Double, val longitude: Double)
```

Výpis kódu 5.6 Datová třída `Stop`

```
@Serializable
data class Stop(
    @SerialName("stop_id")
    val id: String,
    @SerialName("stop_name")
    val name: String?,
    @Serializable(CoordinatesSerializer::class)
    val position: Coordinates,
    @SerialName("location_type")
    @Serializable(LocationTypeSerializer::class)
    val locationType: LocationType,
    @SerialName("platform_code")
    val platformCode: String?,
)
```

5.3.2.4 Deserializace

Nyní se podíváme na deserializaci dat, získaných z webového API, které vrací výsledky ve formátu JSON. Ten budeme pomocí knihovny `kotlinx.serialization` převádět do datových tříd Kotlinu. Pro jednotlivé entity vytvoříme datové třídy obsahující stejné atributy, jako JSON objekty. Postup si ukážeme na příkladu zastávky. Jak vidíme na výpisu 5.6, přidáme datové třídě `Stop` anotaci `@Serializable` — třída je serializovatelná. Třídě přidáme potřebné atributy — například unikátní identifikátor (`id`), název zastávky (`name`) nebo její souřadnice (`position`) — s příslušnými datovými typy. Pokud zvolíme jméno atributu odlišné, od JSON předlohy, musíme to knihovně oznámit anotací `@SerialName("json_attribute_name")`.

Použité datové typy musí být sami serializovatelné. Základní datové typy — String, Double, Boolean, List Intů a další — umí knihovna serializovat sama. Pokud jsme použili vlastní datový typ — například třídu `Coordinates` — musíme ho také anotovat jako serializovatelný. Problém však je, že naše třída obsahující souřadnice (výpis 5.5) neodpovídá struktuře JSON objektu s těmito daty. Pro tyto situace je možné určit vlastní třídu, která provede (de)serializaci — kód 5.6, anotace na řádku 7. Nejjednodušší způsob, jak vytvořit vlastní serializátor je implementovat `JsonTransformingSerializer`. Stačí napsat metodu, která přijme objekt `JsonElement`, transformuje ho a vrátí objekt stejného typu. Typ `JsonElement` je mezistupeň při (de)serializaci. Vstupní textový JSON je preveden do stromové struktury elementů, které si pamatují svůj název a hodnotu. Při implementaci transformujícího serializátoru určíme, který serializátor se má po provedení transformace použít. Vstupní JSON objekt se tedy nejdříve transformuje a poté předá serializátoru, který se vygeneroval použitím anotace `@Serializable` pro třídu `Coordinates`.

Při vývoji prototypu se postupně objevilo několik problémů, které souvisejí s deserializací. Jeden z nich se týká jména zastávky, které nemusí být vždy uvedeno. „Skutečné“ zastávky sice musí mít jméno vždy, komě nich jsou ale v seznamu také stanice, vstupy do nich, nástupní místa a obecné body, které jméno mít nemusí. Samotné řešení je jednoduché — pro úspěšnou deserializaci všech těchto typů zastávek stačí změnit typ proměnné `name` na nullable `String?`. Odhalilo to však potřebu filtrovat zastávky podle typu lokace, což bylo při analýze opomenuto. Filtrování zajistí rozhraní `Repository`, které slouží pro získávání potřebných dat.

Dalším problémem je nestandardní formát času, který určuje GTFS. Pro případy, kdy spoj vyjede v jeden den, ale jízdu skončí až následující den (po půlnoci), se časy udávají s hodinou

Výpis kódu 5.7 Deserializace GTFS času

```
override fun deserialize(decoder: Decoder): LocalTime {
  decoder.decodeString().split(':').map { it.toInt() }.let {
    return LocalTime(it.component1() % 24, it.component2(), it.component3())
  }
}
```

vyšší, než 24 — například `27:32:00`, místo `03:32:00`. Spoj je totiž stále součástí původního „servisního dne“. Golemio API nám správně vrací seznam `stop_times` s časy v určitý den, ale časy spojů z předešlého servisního dne zůstávají v původním formátu. Pro převod si vytvoříme serializátor `GtfsTimeSerializer`, který bude implementovat `KSerializer<LocalTime>`. Ze vstupního stringu vytáhneme jednotlivé časové složky, hodinu zmodulujeme a vrátíme objekt typu `LocalTime` (výpis kódu 5.7). Časovým parametrům třídy `StopTime` určíme tento serializátor obdobně, jako jsme to udělali se souřadnicemi zastávky (výpis 5.6, řádky 7 a 8).

Zasáhnout do deserializace obdobnými způsoby bylo potřeba ještě několikrát. Nikdy ale nešlo o nic zvlášť složitého, což ukazuje kvalitu a schopnosti knihovny, která tak umožňuje snadné multiplatformní řešení (de)serializace.

5.3.2.5 Cache

Nyní si popíšeme, jak pracuje prototyp s kešováním. Třída `CachingRepository` je proxy k repozitáři a zajišťuje ukládání získaných dat do kolekcí (in-memory cache). Třída neřeší invalidaci keše, protože data jsou platná po delší dobu, než bude aplikace zapnutá. Pro optimalizaci aplikace se při dotazování na konkrétní objekty `Route` nebo `Trip` podle jejich ID nakešují všechny objekty daného typu. Důvodem je, že pro skupinu zastávek, která se neskládá jen ze dvou — jedné pro každý směr —, potřebujeme získat tolik linek a spojů, že je srovnatelné načíst rovnou všechny. Dotaz na webové API je poměrně náročný a vyplatí se při něm získat co nejvíce výsledků naráz.

Pro výpočet průměrných intervalů odjezdů potřebujeme data `StopTime`. Je nutné získat data pro časové rozmezí okolo současného okamžiku. Pokud bychom si data do keše ukládali na základě parametrů dotazu — ID zastávky, datum, čas od, čas do —, načítali bychom při opakovaných dotazech data znovu. Data uložená v keši by se vůbec nevyužila. Možným řešením, by bylo data ukládat jen na základě ID zastávky a datumu, bez informace o časech. Při dotazu bychom se podívali do keše, přepoužili alespoň část požadovaného úseku a zbytek donačetli. Z důvodu, který si vysvětlíme v dalším odstavci zvolíme jednodušší variantu — pro požadovaný den načteme všechny objekty, bez časového omezení, a ty pak budeme sami filtrovat.

Krom toho, že je tato varianta výrazně jednodušší na implementaci je možné tato data použít pro získání spojů (`Trip`) pro určitou zastávku. Tuto informaci potřebujeme pro vyhledání linek, které spojují dvě zastávky. Této operaci ale bude vždy předcházet hledání průměrných intervalů linek odjezdové zastávky, které pak jen vyfiltrujeme. Pokud budeme mít `StopTime` objekty už uložené, můžeme z nich přes identifikátory spojů snadno získat celé spoje. Všechny spoje už máme v tu chvíli stejně uložené v keši. První návrh pro optimalizaci získávání spojů spočíval samozřejmě v kešování všech spojů s informací o zastávkách. To by však bylo jednak velice krkolomné, jednak nemožné, protože tuto $N : M$ vazbu zajišťují právě data `StopTime`.

5.3.3 Android prezentační vrstva

V této části se zaměříme na vývoj Android aplikace. Konkrétně se podíváme na implementaci prezentační vrstvy, jelikož o zbytek se stará výše popsaná společná část kódu. Popíšeme si použité knihovny, Android aktivitu a view model.

Výpis kódu 5.8 Ošetření výjimek

```
routesWithIntervals =
  runCatching { pidtotype.findAverageIntervals(stopCluster) }
    .onSuccess {
      _state.value = if (it.isEmpty()) State.EMPTY else State.RESULT
    }
    .onFailure { _state.value = State.ERROR }
    .getOrDefault(emptyList())
```

5.3.3.1 Knihovny

Nejprve si popíšeme použité knihovny, technologie a závislosti Android aplikace.

Jetpack Compose
Jetpack Compose je moderní toolkit pro tvorbu nativního Android UI. Jeho první stabilní verzi vydal Google v červenci 2021 a na konci března 2023 vyšla verze 1.4. Jetpack Compose používá k tvorbě UI deklarativní přístup a nabízí jednoduchou, ale mocnou, náhradu za Views, s kterými však jde v případě potřeby kombinovat. Uživatelské rozhraní mají pomocí Jetpack Compose vytvořené například aplikace Twitter, Airbnb nebo Booking.com. [116, 117, 118]

Závislosti
Závislosti pro Android aplikaci přidáme do souboru `build.gradle.kts` ve složce `androidApp/`. Mimo jiné potřebujeme mít nastavené závislosti na Jetpack Compose, Kotlin coroutines — potřebujeme volat suspending funkce — a pro tvorbu *lifecycle-aware* komponent také Android Lifecycle. Závislost na sdílené části nastavíme přidáním `implementation(project(":shared"))`.

5.3.3.2 ViewModel

Jako mezivrstvu, mezi uživatelským rozhraním popsaným v aktivitě a kódem, který řeší samotnou logiku, použijeme třídu `PidtotypeViewModel`. Bude rozšiřovat Android třídu `ViewModel` a obsahovat informace o aktuálním stavu aplikace. Dále bude obsahovat referenci na třídu `Pidtotype`, skrze kterou bude komunikovat se zbytkem logiky a načítat potřebná data.

Při vytvoření view modelu se spustí stahování seznamu zastávek, které aplikace potřebuje uložené v keši, aby mohla uživatele rychle informovat o tom, zda je hledaný název zastávky platný. Postupně se použijí stavy `LOADING` a `INITIAL`, případně `INITIAL_CACHE_ERROR`. Třída veřejně vystaví metodu `findStops`, která bude přijímat textové názvy hledaných zastávek — obsah textových polí aktivity. Jelikož od uživatele potřebujeme alespoň odjezdovou zastávku, může se stát, že se aplikace dostane do stavu `FROM_STOP_NOT_FOUND`. Ať už uživatel zadal platnou pouze odjezdovou, nebo i cílovou zastávku, přesune se aplikace do stavu `LOADING`.

Pro získání výsledků budeme volat metody ze třídy `Pidtotype`. Protože jsou to suspend funkce, budeme je volat z coroutine bloku `viewModelScope.launch { /*...*/ }`. Tento scope je lifecycle-aware — automaticky zajistí, že běžící coroutines se v případě zničení view modelu zruší. Jelikož výsledek závisí na datech z webového API, musíme ošetřit případné výjimky. Jak vidíme ve výpisu 5.8, stav view modelu nastavíme podle toho, jestli volaná metoda vyhodí výjimku, nebo —pokud ji nevyhodí — podle obsahu výsledku.

Aktivita bude od view modulu také požadovat stav zastávek — zda se podle zadaného názvu podařilo nějakou zastávku najít. Enum `StopState` bude nabývat stavů `FOUND`, `NOT_FOUND` a `EMPTY`. Všechny tyto stavy bude view model vystavovat jako `StateFlow`, což aktivitě umožní měnit UI pouze při změně stavu.

```
private val _state = MutableStateFlow(State.INITIAL)
val state get() = _state.asStateFlow()
```

Výpis kódu 5.9 State hoisting

```
@Composable
fun StopNameInput(/*...*/, stopName: String, onValueChange: (String) -> Unit
    ) {
    OutlinedTextField(
        value = stopName,
        onValueChange = onValueChange,
        /*...*/
    )
}
```

5.3.3.3 Aktivita

Aktivita bude sloužit k samotnému zobrazení UI, k čemuž bude využívat Jetpack Compose. K získání a uchování dat bude mít view model, který má delší životní cyklus. Aby aktivita vždy získala ten stejný view model, musíme ho inicializovat následujícím způsobem:

```
private val viewModel: PidtotypeViewModel by viewModels()
```

Aby jsme zajistili rekompozici UI při změně stavu view modelu, vyzvedneme `StateFlow` data následujícím způsobem:

```
val state by viewModel.state.collectAsStateWithLifecycle()
```

Tímto způsobem — oproti metodě `collectAsState()` — zajistíme, že stav Jetpack Compose, na základě kterého se spouští rekompozice, bude lifecycle-aware. Znamená to, že pokud bude běh aplikace nějakým způsobem přerušen, přeruší se i změna stavu a tím případná rekompozice. Tato metoda byla přidána v Lifecycle verzi 2.6.0 v březnu 2023.

Samotné UI vytvoříme podle wireframe návrhů seskládáním základních komponent, které nám Jetpack Compose nabízí. Tyto stavební bloky jsou composable funkce, které se do sebe skládají. Všechny prvky na obrazovce budou seřazeny pod sebe do sloupce. V horní části, se kterou bude uživatel interagovat budeme mít nadpis, textová pole a tlačítko. Dolní část bude sloužit k zobrazení výsledků, případně chybových hlášek. Konkrétní obsah dolní části bude záviset na stavu, ve kterém se nachází view model.

Pro definování komponent, které se opakují, si vytvoříme vlastní composable funkce. Definujeme si tak například objekt, který zobrazuje linku a její průměrný interval. Pro získání uživatelova vstupu si definujeme composable funkci `StopNameInput(labelText, isError)`, která na základě vstupů vykreslí textové pole. Pro získání hodnoty, kterou uživatel do pole zadal by normálně sloužila návratová hodnota funkce. To však v případě composable funkcí není vhodné — měly by buď produkovat UI, nebo vracet návratovou hodnotu. Namísto toho aplikujeme přístup označovaný jako `state hoisting`. Podstatnou část kódu můžeme vidět na výpisu 5.9. Stav vytáhneme z funkce ven a hodnotu (zde pro zobrazení) budeme funkci předávat jako parametr. Pro změnu hodnoty budeme předávat ještě funkci `onValueChange`.

5.3.4 Volání společného kódu z iOS

Nyní si popíšeme, jestli se nám podařilo společnou část implementovat takovým způsobem, aby se dala použít pro iOS aplikaci a na jaké problémy jsme při tom narazili. Ze seznamu nefunkčních požadavků (viz kapitola 5.1.1.2) zjistíme, že jsme si definovali za cíl pouze vyzkoušet použití sdíleného kódu pro platformu iOS, ne tvorbu ucelené aplikace. Pro otestování se spokojíme s tím, že zavoláme metodu ze společné části. Konkrétně se pomocí metody `CachingRepository::getStops()` dotážeme webového API na seznam zastávek.

Při vytváření projektu nám Android Studio automaticky vygenerovalo složku `iosApp/`, ve které se nachází kód v jazyku Swift k prázdné iOS aplikaci, která však už importuje sdílenou část kódu. Kód pro iOS je potřeba otevřít ve vývojovém prostředí *Xcode* od společnosti Apple. Naše zkušební volání API umístíme do souboru `ContentView.swift`. Odstraníme předgenerované volání třídy `Greeting`, kterou Kotlin Multiplatform používá jako obdobu `Hello, World!`, a namísto toho vytvoříme instanci třídy `CachingRepository`. Použijeme k tomu faktory metodu, které předáme klíč pro Golemio API. Doposud jsme nenarazili na žádný problém, XCode nám našeptává název třídy a my nemáme — z pohledu iOS vývoje ve Swiftu — důvod se domnívat, že jde o kód napsaný v Kotlinu. Doplníme volání metody `getStops()`, které předáme callback, respektive *completionHandler* (podle Swift terminologie), v podobě anonymní funkce, respektive *closure*.

To z pohledu Swiftu stále nepředstavuje nic pozoruhodného. Tato skutečnost je ale zajímavá z pohledu Kotlinu, kde byly metody implementovány jako suspending funkce knihovny Coroutines. Tato knihovna je sice multiplatformní, ale použití suspending funkcí se mezi cílovými platformami liší. Pro Android, který běží na JVM existuje implementace této knihovny a suspending funkce se tak dají použít stejně, jako v čistém Kotlinu. Swift je zcela jiný jazyk, určený pro odlišnou platformu, takže se vzniklé funkce volají jinak. Existuje také podpora volání suspending funkcí ze Swift kódu jako asynchronních, pomocí klíčového slova `await`. Je to umožněno od Swiftu 5.5, ale tato funkcionalita je zatím experimentální a není tedy jisté, jestli se tato možnost v budoucnu udrží. [86]

Po sestavení a spuštění projektu na emulátoru telefonu se systémem iOS jsme narazili na první problém — aplikace vyhodila výjimku a spadla. Na první pohled nebylo z chybové hlášky jasné, kde přesně nastala chyba, aplikace vypsala:

```
Function doesn't have or inherit @Throws annotation and thus exception isn't
propagated from Kotlin to Objective-C/Swift as NSError.
```

To nás upozorňuje na problém vzniklý používáním Kotlin kódu z jazyku Swift. Kotlin totiž nemá kontrolované výjimky, zatímco Swift je má všechny kontrolované — překladač ověřuje, že je nějakým způsobem ošetřujeme. Aby se tato informace Swiftu předala, je potřeba v Kotlinu funkce, které mohou vyhazovat výjimky, anotovat pomocí `@Throws`. [86]

Naštěstí byla tato výjimka doplněna o bližší popis:

```
TLS sessions are not supported on Native platform.
```

Ukázalo se, že problém nevzniká přímo v našem společném kódu, ale v knihovně Ktor, konkrétně jejím HTTP enginu CIO. Tento engine sice podporuje jak Android, tak iOS platformu, ale zřejmě pro obě platformy nepodporuje veškerou funkcionalitu. Řešením, pokud se tomu tak dá říct, je volání API pomocí HTTPS nahradit nešifrovaným HTTP. Takto upravený kód už po spuštění nespadne a úspěšně se dotazuje API na požadované zastávky.

Pokud by se jednalo o aplikaci, která by se měla fakticky nasadit do produkčního provozu, nebylo by příliš žádoucí spokojit se s nešifrovaným HTTP voláním. Pro mnoho dalších API, které pracují s citlivějšími daty, by byla šifrovaná komunikace kritická. Dalo by se toho docílit ustoupením od komfortu multiplatformního enginu. Pokud bychom pro iOS platformu vybrali jiný, schopnější engine, mohli bychom se vrátit k šifrované komunikaci. Museli bychom poté například vytvořit továrnu, která by vracela instanci typu `HttpClientEngine` a jejíž implementace by závisela na konkrétní platformě.

5.4 Testování

Protože jde o důležitou součást vývoje, podíváme se nyní na testování. Přestože bychom naši aplikaci nutně testovat nemuseli — je to přeci jen prototyp, který se nedostane do produkčního nasazení a u kterého tedy není požadavkem mít výsledný kód zcela bezchybný —, testování

Výpis kódu 5.10 Test deserializace JSON dat

```
@Test
fun testCoordinatesDeserialization() {
  val json = """
    {
        "type": "Point",
        "coordinates": [14.4633, 50.07827]
    }
  """.trimIndent()
  assertEquals(
    "Coordinates deserialization",
    Coordinates(50.07827, 14.4633),
    format.decodeFromString(CoordinatesSerializer, json)
  )
}
```

stejně provedeme. Jelikož je cílem projektu vyzkoušet vývoj aplikace pomocí technologie Kotlin Multiplatform, je potřeba se zaměřit na schopnost testování sdíleného kódu. Kromě toho si ale také popíšeme testy uživatelského rozhraní Android aplikace.

5.4.1 Společný kód

Testování je téma, které jsme zatím v této práci nezmiňovali. V případě technologie KMM je hlavní vyzkoušet testování společného kódu. Právě vyčlenění sdílené multiplatformní části odlišuje KMM od nativního vývoje. V podstatě se ale testování vůbec neliší. Pro tvorbu unit testů společného kódu použijeme knihovnu `kotlin.test`, kterou přidáme jako závislost do souboru `build.gradle.kts` ve složce `shared/`:

```
val commonTest by getting {
  dependencies {
    implementation(kotlin("test"))
  }
}
```

Samozřejmě lze použít i jiné testovací knihovny, například Kotest, který podporuje JVM, JavaScript i nativní platformy a obsahuje vlastní rozsáhlou assertions knihovnu. [119]

Podobně jako můžeme rozdělit zdrojové kódy na společné a platformně závislé, je možné rozdělit i testy, které se pak nachází v oddělených složkách — `androidTest/`, `jsTest/` apod. Pro platformy lze určit samostatné testovací závislosti, například pro specifické knihovny na tvorbu mock objektů, ale opět jde použít i ty multiplatformní. Naše aplikace neobsahuje platformně závislý kód, takže nebude potřeba testy tímto způsobem rozdělovat. Přidáme je do složky `shared/src/commonTest/`. Ta slouží pro společné testy, které se automaticky spouští pro všechny platformy. [120]

Kotlin knihovna, kterou pro testování používáme, určuje testy pomocí anotace `@Test`. Výpis kódu 5.10 obsahuje test deserializace JSON dat do třídy `Coordinates`. Výsledek testu ověříme pomocí běžných kontrolních metod jako `asserEquals()`, `assertContains()`, `assertNotNull()` a dalších.

V kódu, z důvodu komunikace s webovým API, často využíváme suspending funkce. Při testování je budeme volat uvnitř bloku `runBlocking { ... }`, který je spustí synchronně a zablokuje hlavní vlákno. Jelikož při testování stejně nahradíme zdroje zdržení (volání API) alternativou určenou právě pro testování (například lokálním repozitářem), testy proběhnou rychle a synchronní volání nebude problém.

Výpis kódu 5.11 Ukázka testu uživatelského rozhraní

```
@ExperimentalTestApi
class MyComposeTest {

  @get:Rule
  val composeTestRule = createAndroidComposeRule<PidtotypeActivity>()

  @Test
  fun stopNotFoundTest() {
    composeTestRule.apply {
      // počkáme na načtení
      waitUntilExactlyOneExists(
          hasText(activity.getString(R.string.resultTextInitial)),
          30_000
      )

      // provedeme akci
      onNodeWithText(activity.getString(R.string.stopLabelFrom))
        .performTextInput("neexistuje")
      onNodeWithText(activity.getString(R.string.searchButton))
        .performClick()

      // zkontrolujeme výsledek
      onNodeWithText(activity.getString(R.string.resultTextStopNotFound))
        .assertIsDisplayed()
    }
  }
}
```

5.4.2 Android aplikace

Vývoj Android aplikace probíhá nativně, s obyčejnou závislostí na sdílenou část, jako by se jednalo o kteroukoliv jinou knihovnu. Z toho důvodu se testování nijak neliší a můžeme využívat všechny knihovny a nástroje, které bychom využili při nativním vývoji.

Jelikož se logika aplikace nachází ve společné části, kterou již testujeme, zaměříme se na testování uživatelského rozhraní. Testování lze provádět manuálně, nebo ho automatizovat. V obou případech využijeme pro tvorbu testovacích scénářů ty, které jsme definovali pro případy užití v kapitole 5.1.2. Pro ověření, že se aplikace chová podle očekávání — jde spustit a voláním společného kódu získává očekávané výsledky —, ji sestavíme a vyzkoušíme na reálném Android telefonu. Ačkoliv takto uživatelské rozhraní úspěšně otestujeme, projít jednotlivé scénáře je časově náročné a opakované provádění těchto testů by nebylo efektivní. Z toho důvodu existují pro Android frameworky pro testování UI.

Pro testování uživatelských rozhraních tvořených z view slouží framework Espresso. Naše aplikace používá pro tvorbu UI JetPack Compose, který má vlastní rozhraní pro tvorbu testů. Jde o instrumented testy, které probíhají během běhu aplikace na zařízení — ať už reálném, nebo emulovaném. Tyto testy jsou typicky ve složce `androidTest/`, zatímco složka `test/` je určená pro unit testy. [121]

Testovací knihovna poskytuje rozhraní, které nám dovolí vyhledat prvky na obrazovce, provést nad nimi akce — například vložení textu nebo stisknutí — nebo zkontrolovat stav grafického prvku — existuje, obsahuje určitou hodnotu, je vybraný a podobně. Jelikož změna stavu po kliknutí může chvilku trvat, framework Compose se snaží co nejlépe zajistit, že před pokračováním v testu počká, až aplikace předešlou operaci dokončí a bude v klidovém stavu — zajišťuje synchronizaci. Problém může nastat především v případě spouštění asynchronního kódu. Tomu lze

předejít nahrazením reálné implementace testovací variantou, která bude běžet synchronně, případně výslovným čekáním na splnění podmínky — například dokud nezmizí element indikující načítání. Framework Compose umožňuje počkat, dokud se nesplní podmínka, protože doba trvání operace může záviset na mnoha faktorech a čekat pevně definovaný čas by mohlo někdy trvat zbytečně dlouho a prodlužovat tak test. Jindy by se zase pokračovalo ještě před dokončením operace a test by skončil neúspěchem. [121, 122]

Na výpisu 5.11 vidíme ukázku testu uživatelského rozhraní. Testujeme zobrazení chybové hlášky v případě, že zastávku nebylo možné podle zadaného jména nalézt. Nejprve si definujeme testovací pravidlo (`ComposeTestRule`), ke kterému přiřadíme naši aktivitu. Pokud nepotřebujeme přístup k aktivitě, můžeme použít metodu `createComposeRule()` a následně pravidlu nastavit kontext — `Composable` funkci, kterou chceme testovat. Po vytvoření aktivity, přesněji řečeno view modelu, se načítá cache, takže musíme počkat, až tato akce skončí a aplikace zobrazí úvodní text, jehož znění pro kontrolu získáme ze string resources. Pokud bychom chtěli testovat pouze UI, bylo by vhodné spíše mockovat repozitář, z kterého aplikace získává data, čímž bychom se vyhnuli nutnosti synchronizace testu. Pro vyzkoušení tohoto nového rozhraní a provedení kompletního end to end testu ve stejné podobě, jako bychom ho prováděli manuálně, ponecháme aplikaci skutečný repozitář, který data stahuje. Pro zadání jména hledané zastávky nejprve nalezneme textové pole, do kterého jméno vyplníme. Následně požádáme framework o kliknutí na tlačítko, které má text *Hledat* a nakonec zkontrolujeme, že se skutečně objevila hláška *Zastávka nenalezena.*

Kapitola 6
Závěr

V této práci jsme popsali mobilní aplikace, výhody jejich využití a důvody, proč je potřeba je vyvíjet pro více platforem. Zaměřili jsme se na technologie Kotlin Multiplatform, Flutter a React Native. Stručně jsme vysvětlili jejich fungování a následně je mezi sebou porovnali.

Ukázalo se, že Kotlin Multiplatform je — na rozdíl od dvou v současnosti nejpopulárnějších multiplatformních frameworků, kterými jsou Flutter a React Native — svým přístupem velice blízký nativnímu vývoji, především pro operační systém Android. Součástí technologie KMM není žádné platformě závislé běhové prostředí, prostředník, nebo engine pro vykreslování grafiky, který by byl součástí každé aplikace a zajišťoval běh multiplatformního kódu. Kotlin místo toho obsahuje překladač pro jednotlivé platformy, mezi kterými jsou i systémy Android a iOS, a nástroje pro jejich interoperabilitu s jazykem Kotlin. Lze tedy implementovat aplikaci, která bude část kódu sdílet mezi více platformami, pro které vyvineme individuální uživatelská rozhraní pomocí nativních nástrojů. Nevýhodou je nutnost nativního vývoje UI pro Android i iOS, ale výraznou výhodou je jednoduchý přístup k nativním rozhraním pro jednotlivá zařízení. Hlavní výhodou frameworků Flutter a React Native je jednotný vývoj UI a odstínění vývojářů od platformy, takže jim stačí pouze znalost frameworku. To je však výhoda pouze v případě, kdy framework vystačí pro implementaci požadované funkcionality. Kotlin Multiplatform tak vyplňuje volný prostor mezi multiplatformními frameworky, kde je veškerý kód společný, a nativním vývojem, kdy není možné sdílet žádný kód.

Pomocí technologie Kotlin Multiplatform, kterou jsme hlouběji popsali ve 4. kapitole, jsme vytvořili prototyp — aplikaci pro výpis průměrných intervalů odjezdů linek hromadné dopravy. Společná část obsahuje kód pro komunikaci s webovým API, serializaci do datových tříd, kešování a zpracování těchto dat pro získání požadovaného výstupu. Tento sdílený kód jsme použili při implementaci Android aplikace a ověřili jsme, že by ho bylo možné použít i pro tvorbu aplikace pro iOS.

V práci jsme tedy technologii Kotlin Multiplatform popsali, porovnali jsme ji s populárními alternativami a zhodnotili, kdy je vhodné použít kterou technologii. Implementací prototypu jsme vyzkoušeli, že ačkoliv je technologie nová a zatím v beta verzi, je dobře použitelná, funkční a intuitivní. Vzhledem k tomu, že Kotlin Multiplatform nabízí pro sdílený kód širší množství cílových platforem, bylo by možné práci rozšířit o praktický test některé další platformy, například tvorbou webové nebo desktopové aplikace. Dalším možným rozšířením by bylo obohacení společného kódu o platformě závislou funkcionalitu, jelikož vybrané zadání prototypu cílilo především na komplexnější zpracování dat získaných z webového API. Přestože umožňovat jednotnou tvorbu uživatelského rozhraní není cílem technologie Kotlin Multiplatform, společnost JetBrains vytváří framework *Compose Multiplatform* (založený na Jetpack Compose z Androidu), který se snaží tuto možnost poskytnout. V další práci by tedy mohlo být zajímavé prozkoumat také tuto technologii, která od dubna 2023 podporuje (v alfa verzi) i systém iOS.

Bibliografie

1. Smartphone. In: *Oxford Advanced American Dictionary* [online]. Oxford University Press, 2023 [cit. 2023-03-29]. Dostupné z: https://www.oxfordlearnersdictionaries.com/definition/american_english/smartphone.
2. TOCCI, Meghan. Smartphone History and Evolution. In: *SimpleTexting Blog* [online]. SimpleTexting, 2023 [cit. 2023-03-29]. Dostupné z: https://simpletexting.com/where-have-we-come-since-the-first-smartphone/.
3. ARTHUR, Charles. The history of smartphones: timeline. In: *The Guardian* [online]. The Guardian, 2012 [cit. 2023-03-29]. Dostupné z: https://www.theguardian.com/technology/2012/jan/24/smartphones-timeline.
4. ELGAN, Mike. How iPhone Changed the World. In: *Cult of Mac* [online]. Cult of Mac, 2011 [cit. 2023-03-29]. Dostupné z: https://www.cultofmac.com/103229/how-iphone-changed-the-world/.
5. GRABHAM, Dan; JONES, Robert. History of the iPhone 2007-2017: the journey to iPhone X. In: *T3* [online]. T3, 2018 [cit. 2023-03-29]. Dostupné z: https://www.t3.com/features/a-brief-history-of-the-iphone.
6. CLEMENT, J. Mobile gaming market worldwide — Statistics & Facts. In: *Statista* [online]. Statista, 2022 [cit. 2023-03-29]. Dostupné z: https://www.statista.com/topics/7950/mobile-gaming-market-worldwide/#topicOverview.
7. STATCOUNTER. Mobile Operating System Market Share Worldwide. In: *StatCounter GlobalStats* [online]. StatCounter, 2022 [cit. 2022-12-07]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide#monthly-200901-202211.
8. 64 million smart phones shipped worldwide in 2006. In: *Canalys* [online]. Canalys, 2007 [cit. 2023-03-30]. Dostupné z: https://www.canalys.com/newsroom/64-million-smart-phones-shipped-worldwide-2006.
9. PERRIN, Guillaume. 2022, a decisive year in the ever-changing Smart TV OS market. In: *Dataxis* [online]. Dataxis, 2022 [cit. 2023-03-30]. Dostupné z: https://dataxis.com/researches-highlights/417144/2022-a-decisive-year-in-the-ever-changing-smart-tv-os-market/.
10. RICKER, Thomas. MeeGo is dead: Resurrected as Tizen, the newest Linux-based open source OS. In: *The Verge* [online]. The Verge, 2011 [cit. 2023-03-30]. Dostupné z: https://www.theverge.com/2011/9/28/2456253/meego-is-dead-resurrected-as-tizen-another-new-linux-based-open.
11. ABLESON, Frank. Introduction to Android development. In: *IBM Developer* [online]. IBM, 2018 [cit. 2023-03-31]. Dostupné z: https://developer.ibm.com/articles/os-android-devel/.

12. CALLAHAM, John. The history of Android: The evolution of the biggest mobile OS in the world. In: *Android Authority* [online]. Authority Media, 2022 [cit. 2023-04-04]. Dostupné z: https://www.androidauthority.com/history-android-os-name-789433/.

13. NOVAC, Ovidiu Constantin; NOVAC, Mihaela; GORDAN, Cornelia; BERCZES, Tamas; BUJDOSÓ, Gyöngyi. Comparative study of Google Android, Apple iOS and Microsoft Windows Phone mobile operating systems. In: *2017 14th International Conference on Engineering of Modern Electric Systems (EMES)*. 2017, s. 154–159. Dostupné z DOI: 10.1109/EMES.2017.7980403.

14. GOOGLE. Application Sandbox. In: *Android OS Documentation* [online]. Google, 2022 [cit. 2023-04-04]. Dostupné z: https://source.android.com/docs/security/app-sandbox.

15. Preparing your UI to run in the background. In: *Apple Developer Documentation* [online]. Apple Developer Documentation, 2023 [cit. 2023-04-04]. Dostupné z: https://developer.apple.com/documentation/uikit/app_and_environment/scenes/preparing_your_ui_to_run_in_the_background.

16. WUERTHELE, Mike. Apple unveils iPadOS, adding features specifically to iPad. In: *AppleInsider* [online]. AppleInsider, 2019 [cit. 2023-04-04]. Dostupné z: https://appleinsider.com/articles/19/06/03/apple-supplements-ios-13-with-new-tablet-specific-ipad-os-branch.

17. Jobs' original vision for the iPhone: No third-party native apps. In: *9to5Mac* [online]. 9to5Mac, 2011 [cit. 2023-04-05]. Dostupné z: https://9to5mac.com/2011/10/21/jobs-original-vision-for-the-iphone-no-third-party-native-apps/.

18. KEMP, Simon. Digital 2022: Time Spent Using Connected Tech Continues to Rise. In: *DataReportal* [online]. DataReportal, 2022 [cit. 2023-03-31]. Dostupné z: https://datareportal.com/reports/digital-2022-time-spent-with-connected-tech.

19. MAGGINI, Giovanni. Deciding between native and cross-platform mobile frontend programming frameworks. In: *IBM Developer* [online]. IBM, 2019 [cit. 2023-04-26]. Dostupné z: https://developer.ibm.com/articles/deciding-between-native-and-cross-platform-mobile-frontend-programming-frameworks/.

20. *Flutter: Multi-Platform* [online]. Google, [b.r.] [cit. 2022-12-07]. Dostupné z: https://flutter.dev/multi-platform.

21. ZAKHOUR, Shams; CHALIN, Patrice; GODERBAUER, Michael et al. FAQ. In: *Flutter documentation* [online]. Google, 2022 [cit. 2022-12-09]. Dostupné z: https://docs.flutter.dev/resources/faq.

22. POLYAKOV, Andrey; PETROVA, Ekaterina. The Six Best Cross-Platform App Development Frameworks. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2022-12-10]. Dostupné z: https://kotlinlang.org/docs/cross-platform-frameworks.html.

23. WHITE, Eli. React Native Team Principles. In: *React Native Blog* [online]. Meta Platforms, Inc., 2020 [cit. 2022-12-09]. Dostupné z: https://reactnative.dev/blog/2020/07/17/react-native-principles.

24. HEREMBOURG, Kevin. 8 Most Popular Mobile App Development Frameworks in 2023. In: *Purchasely Blog* [online]. Purchasely, 2022 [cit. 2023-04-26]. Dostupné z: https://www.purchasely.com/blog/mobile-app-frameworks.

25. SPEEDY, Jd. Adobe acquires Canadian makers of PhoneGap, Nitobi. In: *IT World Canada* [online]. IT World Canada, 2011 [cit. 2023-04-26]. Dostupné z: https://www.itworldcanada.com/article/adobe-acquires-canadian-makers-of-phonegap-nitobi/44606.

26. *Ionic Framework — The Cross-Platform App Development Leader* [online]. Ionic, 2023 [cit. 2023-04-26]. Dostupné z: https://ionicframework.com/.
27. *NativeScript* [online]. OpenJS Foundation, 2023 [cit. 2023-04-26]. Dostupné z: https://nativescript.org/.
28. VAILSHERY, Lionel Sujay. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. In: *Statista* [online]. Statista, 2022 [cit. 2023-04-22]. Dostupné z: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.
29. *Flutter: Flutter apps in production* [online]. Google, [b.r.] [cit. 2023-01-23]. Dostupné z: https://flutter.dev/showcase.
30. LOUGHEED, Parker; NGUYEN, Khanh; MORGAN, Brett et al. Flutter architectural overview. In: *Flutter documentation* [online]. Google, 2023 [cit. 2023-04-06]. Dostupné z: https://docs.flutter.dev/resources/architectural-overview.
31. Anatomy of an app. In: *Flutter documentation* [online]. Google, 2023 [cit. 2023-04-22]. Dostupné z: https://docs.flutter.dev/resources/architectural-overview#anatomy-of-an-app.
32. Architectural diagram. In: *Flutter documentation* [online]. Google, 2023 [cit. 2023-04-26]. Dostupné z: https://docs.flutter.dev/resources/architectural-overview#architectural-layers.
33. *React Native* [online]. Meta Platforms, Inc., 2022 [cit. 2022-12-09]. Dostupné z: https://reactnative.dev/.
34. NABORS, Rachel; GERLEMAN, Nick; KASZUBOWSKI, Bartosz et al. React Fundamentals. In: *React Native docs* [online]. Meta Platforms, Inc., 2023 [cit. 2023-04-09]. Dostupné z: https://reactnative.dev/docs/intro-react.
35. KASZUBOWSKI, Bartosz; JUN, Teik; NABORS, Rachel et al. Core Components and Native Components. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-09]. Dostupné z: https://reactnative.dev/docs/intro-react-native-components.
36. KASZUBOWSKI, Bartosz; RAMOS, Héctor; JU, Teik et al. Navigating Between Screens. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-09]. Dostupné z: https://reactnative.dev/docs/navigation.
37. KASZUBOWSKI, Bartosz; CORTI, Nicola; NAKAZAWA, Christoph et al. Using Hermes. In: *React Native docs* [online]. Meta Platforms, Inc., 2023 [cit. 2023-04-10]. Dostupné z: https://reactnative.dev/docs/hermes.
38. LORBER, Sébastien; KASZUBOWSKI, Bartosz. Glossary. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-10]. Dostupné z: https://reactnative.dev/architecture/glossary.
39. Render pipeline 5. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-22]. Dostupné z: https://reactnative.dev/architecture/render-pipeline#phase-1-render-1.
40. LORBER, Sébastien; KASZUBOWSKI, Bartosz. Render, Commit, and Mount. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-10]. Dostupné z: https://reactnative.dev/architecture/render-pipeline.
41. LORBER, Sébastien; KASZUBOWSKI, Bartosz. View Flattening. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-04-11]. Dostupné z: https://reactnative.dev/architecture/view-flattening.
42. React Native's New Architecture. In: *Medium* [online]. Medium, 2022 [cit. 2023-04-26]. Dostupné z: https://medium.com/coox-tech/deep-dive-into-react-natives-new-architecture-fb67ae615ccd#ca32.

43. PAVLOV, Danil; PACHKOV, Nikolay; POLYAKOV, Andrey. Kotlin Multiplatform. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2022-12-06]. Dostupné z: https://kotlinlang.org/docs/multiplatform.html.
44. *Kotlin Multiplatform for Cross-Platform Mobile Development | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2022-12-09]. Dostupné z: https://kotlinlang.org/lp/mobile/.
45. PETROVA, Ekaterina. Kotlin Multiplatform Mobile Is in Beta – Start Using It Now! In: *The Kotlin Blog* [online]. JetBrains, 2022 [cit. 2022-12-09]. Dostupné z: https://blog.jetbrains.com/kotlin/2022/10/kmm-beta/.
46. KALIUZHNYI, Illia. React Native vs. Flutter: Choosing Your Framework. In: *Freshcode Blog* [online]. Freshcode, 2022 [cit. 2023-04-16]. Dostupné z: https://freshcodeit.com/blog/react-native-vs-flutter.
47. NAGY, Róbert. *Simplifying Application Development with Kotlin Multiplatform Mobile*. Birmingham: Packt Publishing, 2022. ISBN 978-1-80181-258-0.
48. GOOGLE. Android's Kotlin-first approach. In: *Android Developers* [online]. Google, 2022 [cit. 2022-12-09]. Dostupné z: https://developer.android.com/kotlin/first.
49. MANSOUR, Nezar. Flutter vs Kotlin Multiplatform: The 2022 Guide. In: *Instabug Blog* [online]. Instabug, 2020 [cit. 2022-12-13]. Dostupné z: https://blog.instabug.com/flutter-vs-kotlin-mutliplatform-guide/.
50. ZAKHOUR, Shams; TOBIN, Brian; PIERRE-LOUIS et al. Add Flutter to existing app. In: *Flutter documentation* [online]. Google, 2022 [cit. 2023-01-23]. Dostupné z: https://docs.flutter.dev/development/add-to-app.
51. RAMOS, Héctor; ALISON, Nat; NABORS, Rachel et al. Integration with Existing Apps. In: *React Native docs* [online]. Meta Platforms, Inc., 2022 [cit. 2023-01-23]. Dostupné z: https://reactnative.dev/docs/integration-with-existing-apps.
52. PAVLOV, Danil; POLYAKOV, Andrey; LI, Ying et al. Make your Android application work on iOS – tutorial. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-23]. Dostupné z: https://kotlinlang.org/docs/multiplatform-mobile-integrate-in-existing-app.html.
53. SINGH, Saurabh. What It Takes To Successfully Publish Your App On AppStore? In: *Appinventiv* [online]. Appinventiv, 2022 [cit. 2023-01-27]. Dostupné z: https://appinventiv.com/blog/takes-successfully-publish-app-appstore/.
54. STENING, Helena. Material Design v.s. iOS 11. In: *helenastening.com* [online]. Helena Stening, 2017 [cit. 2023-04-14]. Dostupné z: https://www.helenastening.com/material-design-vs-ios-11-hui.
55. SOMMERHOFF, Peter. Introducing Kotlin. In: *Kotlin for Android app development* [online]. Boston: Addison-Wesley, 2019 [cit. 2023-01-06]. ISBN 978-0-13-485419-9. Dostupné z: https://books.google.cz/books?id=WX5_DwAAQBAJ&pg=PT46&hl=cs#v=onepage&q&f=false.
56. GRIN, Tatiana. *Kotlin Media Kit* [online]. JetBrains, 2021 [cit. 2023-01-04]. Dostupné z: https://kotlinlang.org/assets/kotlin-media-kit.pdf.
57. PETRAKOVICH, Victoria; SHADRINA, Anastasia; POLYAKOV, Andrey et al. What's new in Kotlin 1.2. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/whatsnew12.html.
58. PETRAKOVICH, Victoria; PAVLOV, Danil. What's new in Kotlin 1.8.20. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-04-19]. Dostupné z: https://kotlinlang.org/docs/whatsnew18.html.

59. KOTLIN BY JETBRAINS. What Everyone Must Know About The NEW Kotlin K2 Compiler. In: *YouTube* [online]. 2021 [cit. 2023-01-11]. Dostupné z: https://www.youtube.com/watch?v=iTdJJq_LyoY.
60. How Kotlin Multiplatform works. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-04]. Dostupné z: https://kotlinlang.org/docs/multiplatform.html#how-kotlin-multiplatform-works.
61. JETBRAINS. Kotlin Standard Library. In: *Kotlin API reference* [online]. JetBrains, 2023 [cit. 2023-04-19]. Dostupné z: https://kotlinlang.org/api/latest/jvm/stdlib/.
62. JETBRAINS. Kotlin Census 2020. In: *JetBrains* [online]. JetBrains, 2020 [cit. 2023-01-05]. Dostupné z: https://www.jetbrains.com/lp/kotlin-census-2020/.
63. DOLGIKH, Alina. Results of the Kotlin Features Survey 2021. In: *The Kotlin Blog* [online]. JetBrains, 2021 [cit. 2023-01-05]. Dostupné z: https://blog.jetbrains.com/kotlin/2021/12/kotlin-features-survey-2021-results/.
64. Android compilation process. In: *Medium* [online]. Medium, 2020 [cit. 2023-01-06]. Dostupné z: https://medium.com/@banmarkovic/process-of-compiling-android-app-with-java-kotlin-code-27edcfcce616.
65. *Case Studies | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2023-04-19]. Dostupné z: https://kotlinlang.org/lp/mobile/case-studies/.
66. NETFLIX. GraphQL Made Easy for Spring Boot. In: *DGS Framework* [online]. Netflix, 2023 [cit. 2023-04-19]. Dostupné z: https://netflix.github.io/dgs/.
67. ARHIPOV, Anton. Productive Server-Side Development With Kotlin: Stories From The Industry. In: *The Kotlin Blog* [online]. JetBrains, 2020 [cit. 2023-04-19]. Dostupné z: https://blog.jetbrains.com/kotlin/2020/11/productive-server-side-development-with-kotlin-stories/.
68. PAVLOV, Danil; POLYAKOV, Andrey; SEMYONOV, Pavel et al. Kotlin for JavaScript. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/js-overview.html.
69. SEMYONOV, Pavel; POLYAKOV, Andrey; SHADRINA, Anastasia et al. What's new in Kotlin 1.1. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/whatsnew11.html.
70. HAGGARTY, Sarah. Use JavaScript code from Kotlin. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/docs/js-interop.html.
71. OGANDZHANIAN, Shagen; TOLSTOY, Egor. README. In: *GitHub — Kotlin/dukat* [online]. GitHub, 2022 [cit. 2023-04-20]. Dostupné z: https://github.com/Kotlin/dukat.
72. VOLODKO, Ekaterina. Dynamic type. In: *Kotlin Documentation* [online]. JetBrains, 2021 [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/docs/dynamic-type.html.
73. PETRAKOVICH, Victoria. Use Kotlin code from JavaScript. In: *Kotlin Documentation* [online]. JetBrains, 2021 [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/docs/js-to-kotlin-interop.html.
74. SEMYONOV, Pavel; PETRAKOVICH, Victoria; POLYAKOV, Andrey et al. Kotlin/JS IR compiler. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/js-ir-compiler.html.
75. PETRAKOVICH, Victoria; HAGGARTY, Sarah; UDALOV, Alexander et al. What's new in Kotlin 1.8.0. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/whatsnew18.html.

76. JETBRAINS. Mirego. In: *Case Studies | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/lp/mobile/case-studies/mirego/.
77. JETBRAINS. Memrise. In: *Case Studies | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/lp/mobile/case-studies/memrise.
78. JETBRAINS. Down Dog. In: *Case Studies | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/lp/mobile/case-studies/down-dog.
79. RUIKAR, Onkar; CHEN, Joshua; SCHONNING, Nick et al. WebAssembly Concepts. In: *MDN Web Docs* [online]. Mozilla, 2023 [cit. 2023-04-21]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.
80. POLYAKOV, Andrey; PAVLOV, Danil. Interoperability with JavaScript. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-04-21]. Dostupné z: https://kotlinlang.org/docs/wasm-js-interop.html.
81. PAVLOV, Danil; POLYAKOV, Andrey; SEMYONOV, Pavel et al. Kotlin Native. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-05]. Dostupné z: https://kotlinlang.org/docs/native-overview.html.
82. JETBRAINS. Autodesk. In: *Case Studies | Kotlin Multiplatform Mobile* [online]. JetBrains, [b.r.] [cit. 2023-04-25]. Dostupné z: https://kotlinlang.org/lp/mobile/case-studies/autodesk/.
83. POLYAKOV, Andrey; PAVLOV, Danil; SHADRINA, Anastasia et al. Kotlin/Native as an Apple framework — tutorial. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-04-24]. Dostupné z: https://kotlinlang.org/docs/apple-framework.html.
84. POLYAKOV, Andrey; SCHERBINA, Svyatoslav; PAVLOV, Danil et al. Kotlin/Native as a dynamic library — tutorial. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-04-24]. Dostupné z: https://kotlinlang.org/docs/native-dynamic-libraries.html.
85. PETRAKOVICH, Victoria; POLYAKOV, Andrey; SEMYONOV, Pavel et al. Interoperability with C. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-04-24]. Dostupné z: https://kotlinlang.org/docs/native-c-interop.html.
86. PAVLOV, Danil; POLYAKOV, Andrey; SHADRINA, Anastasia et al. Interoperability with Swift/Objective-C. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-04-03]. Dostupné z: https://kotlinlang.org/docs/native-objc-interop.html.
87. KOTLIN. [The compiler is often divided into two parts…] In: *Twitter* [online]. 2021 [cit. 2023-01-11]. Dostupné z: https://nitter.cz/kotlin/status/1453741120899457029#m.
88. COOPER, Keith D.; TORCZON, Linda. Chapter 1 – Overview of Compilation. In: *Engineering a Compiler* [online]. Second Edition. Boston: Morgan Kaufmann, 2012, s. 1–23 [cit. 2023-01-10]. ISBN 978-0-12-088478-0. Dostupné z DOI: https://doi.org/10.1016/B978-0-12-088478-0.00001-3.
89. KOTLIN. [The new #k2compiler…] In: *Twitter* [online]. 2021 [cit. 2023-01-11]. Dostupné z: https://nitter.cz/kotlin/status/1453741469148270593#m.
90. POLYAKOV, Andrey; PAVLOV, Danil; SHADRINA, Anastasia et al. What's new in Kotlin 1.4. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-12]. Dostupné z: https://kotlinlang.org/docs/whatsnew14.html.
91. SEMYONOV, Pavel; POLYAKOV, Andrey; PETRAKOVICH, Victoria. What's new in Kotlin 1.5.0. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-12]. Dostupné z: https://kotlinlang.org/docs/whatsnew15.html.

92. POLYAKOV, Andrey; SEMYONOV, Pavel; PAVLOV, Danil. Kotlin releases. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-01-12]. Dostupné z: https://kotlinlang.org/docs/releases.html.
93. Expect/actual declarations in common and platform-specific modules. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-24]. Dostupné z: https://kotlinlang.org/docs/multiplatform-connect-to-apis.html.
94. LI, Ying; POLYAKOV, Andrey; PAVLOV, Danil. Connect to platform-specific APIs. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2022-12-09]. Dostupné z: https://kotlinlang.org/docs/multiplatform-connect-to-apis.html.
95. Source sets. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-24]. Dostupné z: https://kotlinlang.org/docs/multiplatform-discover-project.html#source-sets.
96. PAVLOV, Danil; PETRAKOVICH, Victoria; POLYAKOV, Andrey et al. Share code on platforms. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-24]. Dostupné z: https://kotlinlang.org/docs/multiplatform-share-on-platforms.html.
97. PAVLOV, Danil; POLYAKOV, Andrey; HAGGARTY, Sarah et al. Adding dependencies on multiplatform libraries. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-25]. Dostupné z: https://kotlinlang.org/docs/multiplatform-add-dependencies.html.
98. Code sharing between platforms. In: *Kotlin Documentation* [online]. JetBrains, 2022 [cit. 2023-01-24]. Dostupné z: https://kotlinlang.org/docs/multiplatform.html#code-sharing-between-platforms.
99. LEE, Derek. The iOS Engineer's Guide to Beginning Kotlin Multiplatform Development. In: *Art and Science of Coding* [online]. Art a Science of Coding, 2022 [cit. 2023-04-27]. Dostupné z: https://artandscienceofcoding.com/science/kmm-for-ios-engineers/.
100. LEE, Derek. Why iOS Engineers Should Avoid This Glorified KMM Technique. In: *Art and Science of Coding* [online]. Art a Science of Coding, 2022 [cit. 2023-04-27]. Dostupné z: https://artandscienceofcoding.com/science/avoid-this-kmm-technique/.
101. LEE, Derek. Code organization using expect/actual. In: *Art and Science of Coding* [online]. Art a Science of Coding, 2022 [cit. 2023-04-27]. Dostupné z: https://artandscienceofcoding.com/science/kmm-for-ios-engineers/#option-1-using-kmms-expect-and-actual-syntax.
102. LEE, Derek. Code organization using a common interface and native implementation. In: *Art and Science of Coding* [online]. Art a Science of Coding, 2022 [cit. 2023-04-27]. Dostupné z: https://artandscienceofcoding.com/science/kmm-for-ios-engineers/#option-2-swift-implementation-for-a-common-kotlin-interface.
103. KHOKHLOV, Evgeny. KMM: writing Kotlin API for Swift - 7 things you need to know. In: *DEV Community* [online]. DEV Community, 2023 [cit. 2023-04-27]. Dostupné z: https://dev.to/ttypic/kmm-writing-kotlin-api-for-swift-7-things-you-need-to-know-4io6.
104. GOLEMIO. Public Transport | Golemio Output Gateway API. In: *Golemio API documentation* [online]. Golemio, [b.r.] [cit. 2023-01-31]. Dostupné z: https://api.golemio.cz/v2/pid/docs/openapi/.
105. ROPID. Otevřená data PID. In: *Pražská integrovaná doprava* [online]. ROPID, 2023 [cit. 2023-02-01]. Dostupné z: https://pid.cz/o-systemu/opendata/.
106. CIBULKA, Jan. Výzva: Praha má (skoro) veřejné jízdní řády, aplikace využívající data zatím chybí. In: *Hospodářské Noviny* [online]. Economia, a.s., 2013 [cit. 2023-02-01]. Dostupné z: https://hn.cz/c1-60401210-vyzva-praha-ma-skoro-verejne-jizdni-rady-aplikace-vyuzivajici-data-zatim-chybi.

107. MCCALLUM, Scott Christian. Background. In: *General Transit Feed Specification* [online]. MobilityData, 2022 [cit. 2023-02-01]. Dostupné z: https://gtfs.org/background/.

108. MCCALLUM, Scott Christian. GTFS: Making Public Transit Data Universally Accessible. In: *General Transit Feed Specification* [online]. MobilityData, 2022 [cit. 2023-02-01]. Dostupné z: https://gtfs.org/.

109. ANTRIM, Aaron; FRACHET, Leo; MILLET, Tim et al. GTFS Schedule Reference. In: *General Transit Feed Specification* [online]. MobilityData, 2022 [cit. 2023-02-01]. Dostupné z: https://gtfs.org/schedule/reference/.

110. DAVIS, Martin. Data model diagrams for GTFS. In: *Lin.ear th.inking* [online]. 2011 [cit. 2023-02-01]. Dostupné z: https://lin-ear-th-inking.blogspot.com/2011/09/data-model-diagrams-for-gtfs.html.

111. GOOGLE. Guide to app architecture. In: *Android Developers* [online]. Google, 2023 [cit. 2023-03-14]. Dostupné z: https://developer.android.com/topic/architecture.

112. PAVLOV, Danil; PETRAKOVICH, Andrey Polyakov Victoria. Set up an environment. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-03-16]. Dostupné z: https://kotlinlang.org/docs/multiplatform-mobile-setup.html.

113. RYZHENKOV, Ilya; HARIRI, Hadi; STASHEVSKY, Leonid et al. Ktor. In: *GitHub* [online]. GitHub, 2022 [cit. 2023-03-22]. Dostupné z: https://github.com/ktorio/ktor.

114. POLYAKOV, Andrey; TOLSTOPYATOV, Vsevolod; CHERNENKO, Konstantin et al. Coroutines guide. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-03-22]. Dostupné z: https://kotlinlang.org/docs/coroutines-guide.html.

115. POLYAKOV, Pavel Semyonov Andrey; SHADRINA, Anastasia et al. Serialization. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-03-22]. Dostupné z: https://kotlinlang.org/docs/serialization.html.

116. GOOGLE. Build better apps faster with Jetpack Compose. In: *Android Developers* [online]. Google, 2023 [cit. 2023-03-23]. Dostupné z: https://developer.android.com/jetpack/compose.

117. BELLINI, Anna-Chiara; BUTCHER, Nick. Jetpack Compose is now 1.0: announcing Android's modern toolkit for building native UI. In: *Google Developers Blog* [online]. Google, 2021 [cit. 2023-03-23]. Dostupné z: https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html.

118. VERHOEF, Jolanda. What's new in the Jetpack Compose March '23 release. In: *Google Developers Blog* [online]. Google, 2023 [cit. 2023-04-30]. Dostupné z: https://android-developers.googleblog.com/2023/03/whats-new-in-jetpack-compose-march-23-release.html.

119. *Kotest* [online]. Kotest Team, 2023 [cit. 2023-05-02]. Dostupné z: https://kotest.io/.

120. PAVLOV, Danil; PETRAKOVICH, Victoria; POLYAKOV, Andrey et al. Create and publish a multiplatform library – tutorial. In: *Kotlin Documentation* [online]. JetBrains, 2023 [cit. 2023-05-02]. Dostupné z: https://kotlinlang.org/docs/multiplatform-library.html.

121. GOOGLE. Automate UI tests. In: *Android Developers* [online]. Google, 2022 [cit. 2023-04-29]. Dostupné z: https://developer.android.com/training/testing/instrumented-tests/ui-tests.

122. GOOGLE. Testing your Compose layout. In: *Android Developers* [online]. Google, 2023 [cit. 2023-04-29]. Dostupné z: https://developer.android.com/jetpack/compose/testing.

Obsah přiloženého média

- `README` ..... stručný popis obsahu média
- `master-thesis/` ..... zdrojová forma práce ve formátu LaTeX
- `master-thesis-matous-dlabal.pdf` ..... text práce ve formátu PDF
- `prototyp`
  - `src/` ..... zdrojové kódy
  - `PIDtotype.apk` ..... instalační balíček prototypu