

Řešení optimalizačních problémů v nativním a hybridním vývoji mobilních aplikací

Bc. Filip Václavík



ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Filip Václavík
Osobní číslo: A23449
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Kombinovaná
Téma práce: Řešení optimalizačních problémů v nativním a hybridním vývoji mobilních aplikací
Téma práce anglicky: Solving Optimization Problems in Native and Hybrid Mobile App Development

Zásady pro vypracování

- Provedte rešerši na téma vývoje mobilních aplikací.
- Zaměřte se na nativní a hybridní vývoj mobilních aplikací.
- Popište vhodné typy testů pro otestování mobilních aplikací.
- Vytvořte vzorovou mobilní aplikaci ve třech verzích – pro OS Android, iOS a hybridní.
- Aplikace vhodně otestujte, vizualizujte výsledky testů a rozeberte dosažené výsledky.

Seznam doporučené literatury:

1. DANIEL, Knott. Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. Financial Times Prentice Hall, 2015. ISBN 9780134191713.
2. SILLARS, Doug. High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing. O'Reilly Media, 2015. ISBN 9781491912515.
3. MIOLA, Alberto. Flutter Complete Reference: Create beautiful, fast and native apps for any device. Independently Published, 2020. ISBN 9798691939952.
4. SKEEN, Josh a GREENHALGH, David. Kotlin Programming: The Big Nerd Ranch Guide (Big Nerd Ranch Guides). Big Nerd Ranch Guides, 2018. ISBN 9780135161630.
5. RANDHAWA, Tejinder S. Mobile Applications: Design, Development and Optimization. Springer International Publishing, 2022. ISBN 9783030023904.

Vedoucí diplomové práce: **doc. Ing. Jiří Vojtěšek, Ph.D.**
Ústav řízení procesů

Datum zadání diplomové práce: **27. října 2024**

Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce se zaměřuje na analýzu a porovnání výkonu mobilních aplikací s ohledem na využití procesoru, operační paměti a spotřebu energie. Cílem je posoudit a porovnat efektivitu nativního a hybridního vývoje. Práce popisuje typické problémy spojené s optimalizací pro jednotlivé platformy a navrhuje způsoby jejich řešení. Pro porovnání byly vytvořeny tři funkčně identické mobilní aplikace, dvě nativní (pro Android a iOS) a jedna hybridní pomocí Flutter frameworku. Tyto aplikace byly podrobeny testování, jehož výsledky jsou graficky zpracovány a následně vzájemně porovnány. Praktická část se věnuje technické analýze těchto implementací z pohledu výkonu, plynulosti uživatelského rozhraní a celkové zátěže systému.

Klíčová slova: mobilní aplikace, mobilní vývoj, nativní vývoj, hybridní vývoj, android, iOS, výkon, rychlost, flutter, dart, kotlin, swift

ABSTRACT

This thesis focuses on the analysis and comparison of the performance of mobile applications with regard to CPU usage, memory consumption, and energy efficiency. The aim is to evaluate and compare the effectiveness of native versus hybrid development approaches. The work outlines common optimization challenges specific to each platform and proposes potential solutions. For the purposes of comparison, three functionally identical mobile applications were developed: two native (for Android and iOS) and one hybrid using the Flutter framework. These applications were subjected to testing, with the results presented graphically and compared with each other. The practical section provides a technical analysis of the implementations, focusing on performance, user interface smoothness, and overall system load.

Keywords: mobile application, mobile development, native development, hybrid development, android, iOS, performance, speed, flutter, dart, kotlin, swift

Tímto bych chtěl poděkovat mému vedoucímu panu doc. Ing. Jiří Vojtěšek, Ph.D. za jeho ochotu a pomoc při spolupráci na této diplomové práci.

OBSAH

ÚVOD	13
I TEORETICKÁ ČÁST	15
1 TYPY MOBILNÍCH OPERAČNÍCH SYSTÉMŮ	16
1.1 ANDROID	16
1.2 IOS	17
1.3 DALŠÍ OPERAČNÍ SYSTÉMY	18
2 MOŽNOSTI VÝVOJE MOBILNÍCH APLIKACÍ	20
2.1 NATIVNÍ VÝVOJ MOBILNÍCH APLIKACÍ	20
2.2 HYBRIDNÍ VÝVOJ MOBILNÍCH APLIKACÍ	20
2.2.1 Flutter	22
2.2.2 Další hybridní frameworky	23
2.3 MULTIPLATFORMNÍ VÝVOJ MOBILNÍCH APLIKACÍ	24
2.4 POROVNÁNÍ NATIVNÍHO A HYBRIDNÍHO VÝVOJE	24
3 FUNKCE MOBILNÍCH APLIKACÍ	26
3.1 ROZŠÍŘENÁ REALITA	26
4 OPTIMALIZAČNÍ STRATEGIE	29
4.1 CÍLE OPTIMALIZAČNÍ STRATEGIE	29
4.2 ZVIDITELNĚNÍ APLIKACE	32
4.3 OPTIMALIZACE VÝKONU	32
4.3.1 Start aplikace	33
4.3.2 Offline mód	33
4.3.3 Procesor	33
4.3.4 Paměť	34
4.3.5 Baterie	35
4.4 MONITOROVÁNÍ APLIKACE	36
5 METODOLOGIE TESTOVÁNÍ VÝKONU	37
5.1 ZÁKLADY TESTOVÁNÍ	37

5.2	PROCES TESTOVÁNÍ	39
5.3	TYPY TESTŮ	40
5.3.1	Cílené testování.....	40
6	TESTOVACÍ NÁSTROJE	44
6.1	TESTOVACÍ ZAŘÍZENÍ.....	44
6.1.1	Emulátor	44
6.1.2	Simulátor.....	45
6.2	NATIVNÍ TESTOVACÍ NÁSTROJE	45
6.2.1	Android Studio	46
6.2.2	XCode a XCode Instruments	48
6.3	NÁSTROJE TŘETÍCH STRAN	49
II	PRAKTICKÁ ČÁST	50
7	NÁVRH TESTOVACÍ APLIKACE	51
7.1	NÁVRH A CÍL APLIKACE	51
7.2	FUNKCIONALITY APLIKACE	52
7.3	POUŽITÉ TECHNOLOGIE A ARCHITEKTURA.....	57
7.3.1	Zvolené technologie	58
7.3.2	Založení projektu.....	58
7.3.3	Nastavení aplikací.....	59
7.4	NAČÍTÁNÍ A ZPRACOVÁNÍ DAT	59
7.4.1	Nativní Android realizace	61
7.4.2	Nativní iOS realizace.....	62
7.4.3	Flutter realizace.....	64
7.5	POŘIZOVÁNÍ FOTOGRAFIÍ A LOKÁLNÍ ÚLOŽIŠTĚ	65
7.5.1	Nativní Android realizace	66
7.5.2	Nativní iOS realizace.....	67
7.5.3	Flutter realizace.....	68
7.6	ROZŠÍŘENÁ REALITA	69
7.6.1	Nativní Android realizace	69
7.6.2	Nativní iOS realizace.....	70

7.6.3	Flutter realizace.....	71
7.7	SHRNUTÍ	72
8	TESTOVACÍ SCÉNÁŘE.....	77
8.1	MĚŘENÍ RYCHLOSTI	77
8.1.1	Návrh	77
8.1.2	Nativní Android testování.....	78
8.1.3	Nativní iOS testování.....	81
8.1.4	Flutter testování	83
8.1.5	Závěr.....	86
8.2	MĚŘENÍ VÝKONU	91
8.2.1	Návrh	91
8.2.2	Nativní Android testování.....	92
8.2.3	Nativní iOS testování.....	94
8.2.4	Flutter testování	96
8.2.5	Závěr.....	99
8.3	SHRNUTÍ	103
	ZÁVĚR.....	106
	SEZNAM POUŽITÉ LITERATURY	109
	SEZNAM OBRÁZKŮ	114
	SEZNAM TABULEK	116
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	117
	SEZNAM PŘÍLOH.....	118

ÚVOD

Trh s mobilními aplikacemi každým dnem roste. Stále častěji vznikají mobilní verze již existujících softwarových řešení, které jsou pro uživatele pohodlnější a přehlednější na ovládání. Při vývoji takových aplikací se vývojáři často setkávají s rozhodnutím, zda zvolit nativní přístup, tedy vývoj samostatné aplikace pro každou platformu, nebo využít hybridní řešení, které umožňuje cílit na více platform s jedním sdíleným kódem. Právě na tuto otázku se tato práce zaměřuje – zda je výhodnější vývoj pomocí nativních nástrojů, nebo zvolit hybridní přístup.

Nejpoužívanějšími mobilními operačními systémy jsou Android a iOS, přičemž každý z nich má výrazně odlišné prostředí a architekturu. Aplikace vyvíjené hybridně mohou mít problémy s optimalizací, zejména pokud se jedná o náročnější funkce. Na druhou stranu, pokud aplikace není příliš složitá, může být hybridní přístup výhodnější díky rychlejšímu vývoji, jednodušší údržbě a nižším nákladům. Nativní vývoj sice obvykle nabízí vyšší úroveň optimalizace a lepší výkon, ale vyžaduje vytváření a správu dvou samostatných aplikací, což přináší vyšší časové i finanční náklady.

Tato práce se zabývá porovnáním tří funkčně identických aplikací – hybridní verze a dvou nativních verzí (pro Android a iOS). Cílem je zhodnotit, jak si hybridní a nativní přístupy vedou z hlediska výkonu a rychlosti v rámci definovaných testovacích scénářů.

V rámci praktické části byly všechny tři aplikace testovány na základě identických scénářů. Bylo měřeno využití procesoru, paměti RAM, spotřeba baterie a plynulost uživatelského rozhraní. Výsledky byly zaznamenány pomocí vývojářských nástrojů příslušných platform (Android Studio, Xcode Instruments) a analyzovány s cílem zjistit, která implementace je z hlediska výkonu nejefektivnější. Práce tak poskytuje nejen technické srovnání, ale i praktické doporučení pro vývojáře při rozhodování o výběru platformy a přístupu k vývoji.

Pro vývoj hybridní aplikace byl použit framework Flutter a jazyk Dart. Nativní aplikace pro Android a iOS byly vyvíjeny v prostředích Android Studio a Xcode, v jazycích Kotlin a Swift. Součástí práce je rovněž popis rozdílů v implementaci na jednotlivých platformách, včetně toho, jak každé prostředí přistupuje k integraci hardwarových prvků zařízení.

Měření výkonu aplikací probíhalo na základě předem definovaných scénářů, pro které byly sledovány konkrétní metriky jako například doba vykreslení prvků nebo zatížení proce-

soru. Každý scénář byl testován v pěti až deseti iteracích, aby bylo možné získat průměrné hodnoty včetně odchylky. Výsledky těchto měření jsou prezentovány ve formě tabulek, grafů a ilustračních obrázků. Použité nástroje a komponenty jsou v textu zvýrazněny tučně, zatímco názvy funkcí z kódu jsou uvedeny odlišným písmem pro snadnější orientaci.

I TEORETICKÁ ČÁST

1 TYPY MOBILNÍCH OPERAČNÍCH SYSTÉMŮ

Kapitola se převážně bude zabývat systémy Android a iOS jelikož se jedná o nejrozšířenější mobilní operační systémy. Podle statistik z článku, které uvádí inženýrka Disha Shah [1], od roku 2024 se použití Androidu pohybuje kolem 70 % a použití a iOS kolem 28 %. Nativní vývoj mobilních aplikací na jiné operační systémy je minimální. Kapitola obsahuje výčet typů operačních systémů. Ke každému operačnímu systému je zde napsán souhrn jeho klíčových vlastností, v čem se aplikace na daný operační systém vyvíjí a jaké jsou výhody a nevýhody těchto operačních systémů [1].

1.1 Android

Operační systém Android je jeden z nejpoužívanějších operačních systémů na mobilních zařízeních. Operační systém je vyvíjen společností Google a jedná se o open-source operační systém založen na Linuxovém jádru. Android je operační systém, který je velice všestranný a lze ho uplatnit nejen na mobilních zařízeních, ale i na chytrých televizích, hodinkách, domácích spotřebičích, a i na jednoduchých obslužných stanicích. Jedna z mnoha unikátních vlastností oproti třeba iOS operačního systému je ta, že uživatelé si mohou přizpůsobovat rozhraní aplikace dle jejich libosti. Android zařízení také obsahují virtuálního asistenta, který využívá umělou inteligenci a jedná se o populární Google Assistant. Na Android je možné instalovat aplikace přes Google Play Store, který nejenže obsahuje aplikace, ale i obsah ke stažení jako filmy, knihy a muziku [1][2][3][4].

Aplikace na zařízení s operačním systémem Android jsou primárně programovány v softwarové aplikaci Android Studio a k jejich vývoji se používá programovací jazyk Kotlin, který nahradil jazyk Java. Vývojové nástroje pro Android jsou dostupné na všech platformách jako je například Windows, macOS nebo Linux. Ačkoliv se Android může pro mnohé zdát jako správná volba díky jeho rozšířenosti, tak pro vývojáře je díky této skutečnosti náročné udržovat aplikace v aktualizovaném stavu a to převážně díky mnoha různým výrobcům rozdílných zařízení, které využívají Android jako například Samsung a Sony. Testování android aplikací je proto zdlouhavé a rozsáhlé. Díky náročné údržbě také Android aplikace nepatří mezi nejlevnější. Jelikož se jedná o open-source operační systém, tak je i náchylnější k získání virů. Nativní programování aplikací pro Android je i pro vývojáře náročnější než například v případě iOS, i když s rychlým vývojem se prostředí pro Android vývojáře zlepšuje. Pro

publikování aplikace na Google Play je nutné, aby si vývojář založil vývojářský účet za 25 dolarů. Na tento jeden účet s jednorázovou platbou může vývojář publikovat mnoho aplikací [1][2][3][4].

1.2 iOS

Operační systém iOS je druhým nejrozšířenějším operačním systémem hned za Androidem. Tento operační systém je vyvíjen společností Apple a na rozdíl od operačního systému Android, se jedná o uzavřený ekosystém, který má za cíl běžet pouze na zařízeních vyvinutých společností Apple jako například iPhone, iPad a podobně. Tento operační systém je převážně oblíbený v USA, zatímco Android je více populární mimo USA. Firma Apple namísto rozšiřování svého produktu mezi nejvíce lidí, se zaměřuje na budování stálých zákazníků, kteří jim budou loajální a budou kupovat zařízení převážně od nich. Mezi hlavní výhody iOS zařízení patří velice jednoduché UI a UX. UI je zkratka pro uživatelské rozhraní, do kterého patří vzhled aplikace a prvky aplikace jako tlačítka. UX je zkratka pro uživatelskou zkušenost, která definuje, jak by se například daná tlačítka měla chovat napříč celou aplikací. Rozhraní Apple zařízení je jednotné přes všechny typy zařízení a sdílí tak spolu tzv. Cocoa vrstvu, která definuje ten jednoduchý vzhled, kterým jsou iOS aplikace tak známé. Jedna z hlavních vlastností je také mnohem větší bezpečnost díky uzavřenému ekosystému. Aplikace na zařízení iOS jsou k dispozici v aplikaci App Store, jedná se téměř o přímý ekvivalent aplikace Google Play [1][2][4][5][6][7].

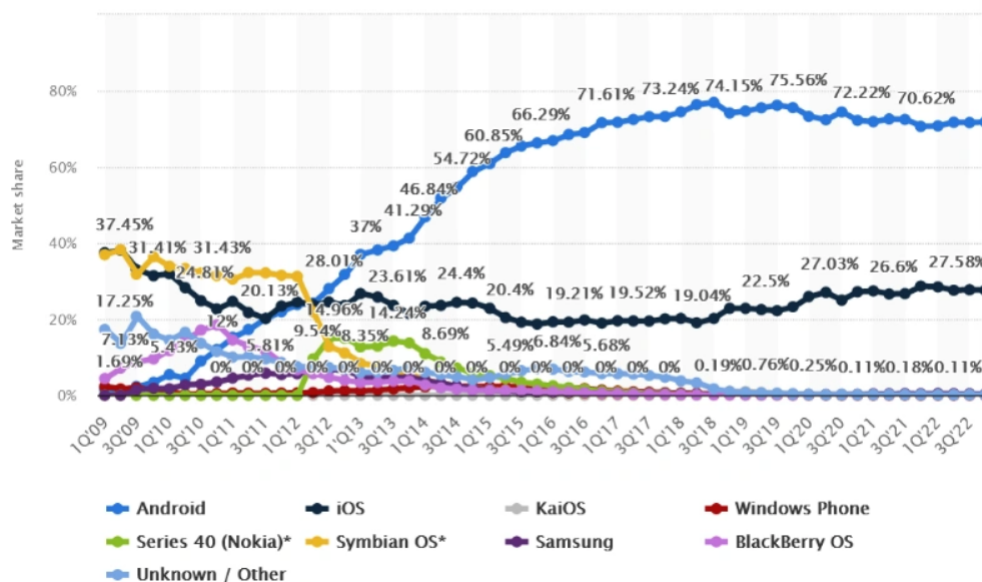
Aplikace na iOS jsou programovány ve vývojovém prostředí XCode a využívají programovací jazyk Swift, který nahradil zastaralý Objective-C. Toto vývojové prostředí je dostupné pouze na zařízeních s operačním systémem od společnosti Apple, jako je například macOS. Vývoj a testování je tedy mnohem více limitované. Pro testování je také nutné vlastnit fyzické zařízení, jelikož vývojové prostředí obsahuje pouze iOS simulátor, který není schopný replikovat hardwarové funkce mobilního zařízení. I přes tyto potíže je ovšem vývoj iOS aplikací v mnoha ohledech jednodušší. Vývojové prostředí je pro vývojáře mnohem přehlednější a křivka učení jazyka je mnohem strmější než v případě nativního Android vývoje. Testování je také zjednodušené, jelikož pro testování existuje mnohem menší počet zařízení, které se od sebe moc neliší. Pro publikování aplikací na App Store je nutné, aby si vývojář pořídil vývojářský účet. Aktivní vývojářský účet, který slouží k prodeji aplikací stojí 99 dolarů ročně. Publikace aplikací na App Store taky může být zdlouhavější než v případě

Google Play, jelikož obsahuje více ověřovacích procesů [1][2][4][5][6][7].

1.3 Další operační systémy

Android a iOS patří mezi nejpoužívanější operační systémy na mobilních zařízeních. Nejsou ovšem jediné a mezi další známé značky patří třeba Windows Mobile. Tento operační systém je vyvinutý společností Microsoft. Tento operační systém nabízel vše, co klasický operační systém na počítač a k tomu navíc technologii pro podporu mobilních zařízení. Obsahoval tzv. dlaždicový layout elementů, které si uživatel mohl libovolně přeskupit. I přes inovativní vlastnosti, Windows telefony nezískaly na popularitě a měly problém se získáním zákazníků, obzvláště kvůli limitovanému ekosystému pro mobilní aplikace [8].

Jedním z dříve populárních operačních systémů, který dominoval na trhu byl operační systém BlackBerry, který se zaměřoval na bezpečnost a produktivitu. Díky jeho zabezpečeným funkcím nabral na popularitě. Také na těchto zařízeních bylo velice přirozené používat fyzickou klávesnici, které již mnoho zařízení nemají. S postupným vývojem Androidu a iOS ovšem popularita BlackBerry klesala a nyní již i BlackBerry zařízení využívají operační systém Android [8]. Pokles popularity operačního systému BlackBerry je patrný obrázku 1.1.



Obrázek 1.1 Použití mobilních OS v čase

(zdroj: [9])

Ještě stojí za zmínku operační systém Harmony. Tento operační systém byl vyvinut společností Huawei a nebyl vyvinut pouze na mobilní zařízení, ale i chytré televize a hodinky.

Jeho cílem bylo mít jednotné rozhraní přes všechny zařízení pro snadné interakce. Tento operační systém je stále globálně používán. Přičemž nejvíce tento operační systém využívá Čína. Podíl tohoto operačního systému na trhu je ovšem tak malý, že ve většině případů se nevyplatí na tento operační systém cílit nativní aplikace [8].

Na závěr dalším částečně používaným operačním systémem je Magic OS. Jedná se o proprietární operační systém vyvinutý společností HONOR, který měl za cíl zlepšit uživatelskou interaktivitu skrz všechny HONOR zařízení. Nejnovější verze tohoto operačního systému představují multitasking, který umožňuje uživatele přepínat pracovní prostředí mezi více zařízeními. Mezi dobré vlastnosti, kterými se tento operační systém chlubí, patří kvalitní zabezpečení a ochrana osobních údajů. Firma HONOR také ke svému operačnímu systému nabízí fit aplikaci pro zlepšení osobního zdraví s personalizovanými nabídky [8].

2 MOŽNOSTI VÝVOJE MOBILNÍCH APLIKACÍ

Tato kapitola obsahuje a rozebírá, jakým způsobem lze vyvíjet mobilní aplikace. Mobilní aplikace se dají vyvíjet třemi způsoby, nativně, hybridně a multiplatformě. Pro tyto způsoby tato kapitola obsahuje rozbor výhod či nevýhod a výčet dostupných nástrojů.

2.1 Nativní vývoj mobilních aplikací

Nativní vývoj mobilní aplikace znamená, že daná aplikace poběží pouze na jednom specifickém zařízení a platformě a to na tom, na které je určena. Nativní aplikace je možné sestavit pro více zařízení jako například počítače a chytré televize, ale pojem nativní vývoj je mnohem více spojován s mobilními aplikacemi. Nativní aplikace neběží ve webovém prohlížeči, ale vyžadují instalaci na konkrétní platformě jako například z App Store na iOS nebo na Google Play na Android. Jelikož nativní aplikace jsou vyvíjeny pro konkrétní platformu, operační systém nebo zařízení, tak jsou pro tu platformu mnohem lépe optimalizovány a jsou tak lépe integrovány s prvky daného zařízení. Tyto aplikace využívají efektivněji dostupné softwarové a hardwarové funkce daného zařízení. Nejen to, ale nativní aplikace jsou také schopné dodržovat nativní rozložení a vzhled elementů svázaných s operačním systémem. Také s rychlým vývojem mobilních zařízení jsou na trhu neustále představovány nové a inovativní funkcionality. V nativním vývoji je tyto nové funkce možné ihned začít používat, jelikož k nim mají přímý přístup [9][10].

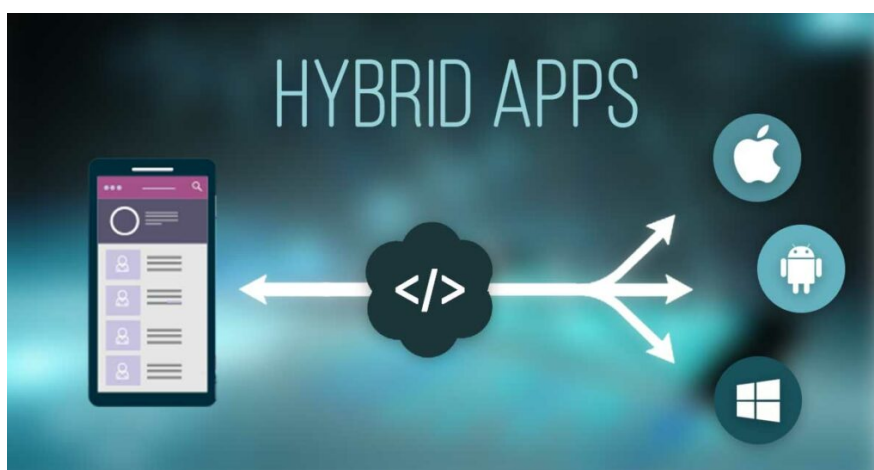
Nativní aplikace lze vyvíjet například pro Android v jazyce Kotlin ve vývojovém prostředí Android Studio a pro iOS lze tyto aplikace vyvíjet v jazyce Swift a vývojovém prostředí XCode. Nevýhoda nativního vývoje převážně spočívá v tom, že na trhu mobilních zařízení právě dominují dva operační systémy, a to Android a iOS. Když je nutné oslovit velkou škálu zákazníků nebo vyvíjená aplikace vyžaduje, aby byla dostupná všem, tak je při nativním vývoji nutné vyvinout dvě aplikace. Vývoj dvou nativních aplikací je velice zdoluhavý proces, který může stát příliš mnoho úsilí a peněz [9][10].

2.2 Hybridní vývoj mobilních aplikací

Hybridní aplikace jsou stavěny tak, aby běžely na více platformách, jak je znázorněno na obrázku 2.1. Hybridní vývoj kombinuje nativní vývoj a vývoj webových aplikací. Hybridní aplikace tedy mívají webové zobrazení a jsou programovány pomocí front-end jazyků. Mezi

nejvýraznější výhodou hybridních aplikací patří to, že je možné, aby běžely na více platformách a je tak tedy možné programovat pouze jednu aplikaci, nebo je i možné překlomit již stávající software do mobilní aplikace. Tato možnost velmi krátí čas ve vývoji aplikací za cenu ne úplně plynulého běhu aplikace, obzvláště co se týče používání nativních prvků zařízení. Hybridní aplikace mají tu nevýhodu, že pro odemčení jistých funkcionalit je nutné stahovat externí knihovny třetích stran. Tato nevýhoda se hlavně projeví, když vyjde nějaká nová funkcionalita, jelikož tyto nové funkce nejsou ihned dostupné na hybridních aplikacích a vývojáři tak musí čekat na vyvinutí externích knihoven.

Při vývoji hybridní aplikace je také velký rozdíl v UI a UX vyvinuté aplikace. Zde se již nemusí jednat o nevýhodu, ale je to také něco na co je nutné si dávat pozor. Android i iOS mají pevně definované styly pro své elementy skrz zařízení, které se nativně používají. Může se tedy stát, že hybridně vyvinuté aplikace budou vzhledově vyčnívat, jelikož tento design rozbíjí. Nativní aplikace jak pro Android tak iOS mají stanovený nějaký vzhled a rozložení prvků. Hybridně vyvinuté aplikace, tento nativní vzhled nedodržuje a nebude tak odpovídat prvkům, na které jsou již uživatelé daného operačního systému zvyklí, jako například vzhled notifikací. Toto je možné řešit importováním externích knihoven a přidáním kontroly na používané zařízení.



Obrázek 2.1 Cíl hybridní aplikace

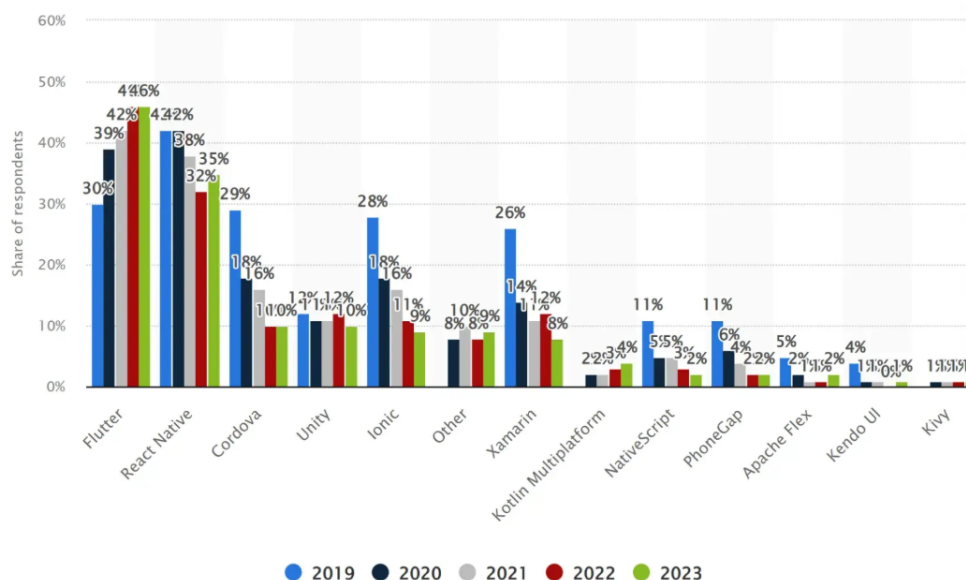
(zdroj: [11])

Hybridní aplikace lze vyvíjet ve frameworkcích jako je například Apache Cordova nebo Ionic, ovšem jiné zdroje mohou uvádět i frameworky jako Xamarin a Flutter jako jejich součástí. Hybridní aplikace lze vyvíjet na všech zařízeních a pro mnoho z nich stačí jedno-

duché vývojové prostředí jako například Visual Studio nebo Visual Studio Code [12][13][14][11][15][16].

2.2.1 Flutter

Flutter je open-source framework vyvíjený společností Google, který byl oficiálně představen v roce 2018. Od svého uvedení si získal velkou popularitu díky své jednoduchosti, rychlosti vývoje a schopnosti nativně kompilovat aplikace pro více platforem z jednoho kódu. Nárůst jeho popularity lze sledovat na obrázku 2.2. Původně byl určen pro vývoj mobilních aplikací, avšak postupem času se rozšířil i na další oblasti, jako je vývoj webových a desktopových aplikací [17][18][19].



Obrázek 2.2 Nárůst popularity frameworku Flutter

(zdroj: [17])

Jednou z hlavních výhod Flutteru je jednoduchý proces instalace a široká podpora napříč různými vývojovými prostředími jako je například Visual Studio Code nebo i Android Studio. Díky tzv. **hot reloadu** mohou vývojáři okamžitě vidět změny v kódu přímo v běžící aplikaci, což výrazně urychluje vývojový cyklus. **Hot reload** umožňuje rychle vývojáři přenačíst změny, bez nutnosti sestavování celé aplikace. Tato vlastnost byla dlouhodobě nedostupná v nativních vývojových prostředích, zejména u Androidu, kde bylo nasazení změn zdouhavější [17][18][19].

Framework využívá programovací jazyk Dart a staví na systému widgetů, které tvoří sta-

vební prvky uživatelského rozhraní. Vývoj aplikací ve Flutteru je intuitivní a umožňuje přizpůsobení uživatelského rozhraní konkrétní platformě pomocí různých rozšíření. Flutter je navíc vysoce optimalizovaný pro mobilní zařízení a v porovnání s jinými frameworky dosahuje velmi dobrých výsledků v oblasti výkonu a spotřeby paměti [17][18][19].

Mezi další výhody patří rozsáhlý ekosystém balíčků a pluginů, které výrazně usnadňují implementaci běžných funkcionalit. Vývoj vlastních rozšíření je přitom poměrně snadný. Pro firmy je Flutter atraktivní zejména díky nižším nákladům, jelikož vývojáři se mohou soustředit pouze na jeden kódový základ namísto odděleného vývoje pro Android a iOS [17][18][19].

Na druhou stranu je třeba zmínit i některé nevýhody. Ne všechny nativní funkcionality jsou ve Flutteru dostupné ihned, a výběr vhodného pluginu může být v některých případech komplikovaný – například při práci s fotoaparátem nebo mapovými službami. Flutter také nemá pevně definovaný architektonický vzor, a proto je správa dat mezi obrazovkami závislá na použití knihoven třetích stran, jako je například provider. Obecně se však ve Flutter komunitě často uplatňuje architektonický vzor MVC (Model-View-Controller), kde je logika aplikace rozdělena do tří základních částí – modelu, uživatelského rozhraní a řídicí logiky [17][18][19].

2.2.2 Další hybridní frameworky

Do oblasti hybridního vývoje vstoupila také společnost Microsoft se svou .NET platformou. Pro tvorbu aplikací nabízí různé nástroje, jako jsou ASP.NET, Xamarin, MAUI a nově také Blazor. Tyto technologie využívají jazyk C# a často i deklarativní značkovací jazyk XAML, který slouží k popisu uživatelského rozhraní a uchovávání vlastností objektů. Ačkoliv vývoj v rámci těchto nástrojů může být finančně výhodnější než čistě nativní přístup, často naráží na problémy spojené s optimalizací a výkonem, což vyžaduje jednodušší návrh aplikace. Přesto patří Microsoft řešení mezi rychlejší hybridní frameworky, které se z hlediska flexibility a výkonu blíží možnostem Flutteru [15].

Za zmínku stojí i další multiplatformní frameworky, které si v minulosti získaly velkou popularitu, jako například Ionic Framework, Apache Cordova nebo React Native. Tyto technologie původně vznikly pro vývoj webových aplikací, ale postupně se rozšířily i do oblasti mobilního vývoje. Ionic a Cordova umožňují vytvářet aplikace pomocí HTML, CSS

a JavaScriptu, které se následně zabalují do nativního kontejneru a spouštějí na mobilních zařízeních. React Native od společnosti Meta naopak umožňuje psát aplikace v JavaScriptu (případně TypeScriptu) s využitím nativních komponent. Tyto frameworky stále mají své výhody jako například jejich široká komunita a rychlý vývoj [16].

2.3 Multiplatformní vývoj mobilních aplikací

Multiplatformní vývoj aplikací má stejný cíl jako hybridní vývoj, a to vyvinout jednu aplikaci pro více platform se znovupoužitelným kódem. Cíl je sice stejný, ale postup se liší, jelikož tato aplikace nemá komponenty pro webové rozhraní a snaží se více podobat nativním aplikacím. Multiplatformní vývoj se dost využívá v případě, kdy je nutné vyvinout jednu aplikaci pro Android a iOS a to rychle a za nízkou cenu. Vyvinuté aplikace tímto způsobem také mají více nativní vzhled než hybridní aplikace. Aplikacím vyvinutým tímto způsobem ovšem chybí dostatečná rychlost a nejsou porovnatelné s nativními aplikacemi, jelikož stejně jako u hybridních aplikací jim chybí správná optimalizace s nativními prvky zařízení a operačního systému. Také tyto multiplatformní aplikace jsou limitovány novými funkcionalitami na zařízeních a funkcemi pro přizpůsobení. Pro vývoj multiplatformních aplikací se používají frameworky jako React Native nebo .NET MAUI. Některé zdroje, ale také uvádí do tohoto typu vývoje i frameworky jako Ionic, Xamarin a Flutter [11][20].

2.4 Porovnání nativního a hybridního vývoje

Nativní vývoj je vhodné použít, pokud je žádoucí vytvořit kvalitní aplikaci, která využívá plný potenciál daného operačního systému. Nativní aplikace je mnohem lépe optimalizována pro daný operační systém. Využívá mnohem lépe softwarové i hardwarové funkce zařízení než aplikace, která byla vyvinuta hybridně. Při představení nových funkcionalit pro dané zařízení, jsou tyto funkcionality ihned dostupné vývojářům. Tedy aplikace, které jsou vyvinuty nativně jsou rychlejší, efektivněji využívají paměť, mají k dispozici nové funkcionality v nejnovějších verzích, a vzhled dané aplikace přirozeně zapadá do prostředí operačního systému. Nevýhoda nativního vývoje spočívá v tom, že mnohdy je potřeba vyvinout více než jen jednu aplikaci a cílový produkt je nutné směřovat na více platform jako je například Android a iOS. Toto zpravidla vyžaduje, aby do vývoje bylo investováno mnohem více času a více financí. Vývoj na Android i iOS se v mnoha ohledech liší a pro jednoho vývojáře nebo jeden konkrétní tým by vývoj byl až příliš složitý. V tomto případě je obvykle také dobré

alokovat zdroje pro více týmů nebo najmout externí specialisty, kteří jsou schopni naprogramovat optimalizovanou aplikaci, aby bylo možné tyto dvě aplikace vyvíjet zároveň.

Hybridní vývoj mnoho těchto problémů řeší, protože v hybridním vývoji se vyvíjí pouze jedna aplikace, kterou je možné spustit na více platformách. Nejen to, ale pokud je řeč o mobilních aplikacích, tak aplikace vyvinuté hybridně ve webovém rozhraní, je pak také i velice snadné překlomit, aby běžely v mobilní aplikaci, a to na většině dostupných operačních systémech. Tyto výhody ušetří při vývoji mnoho vývojového času, což má velký vliv na snížení ceny za vývoj. S tím bohužel přichází jistá negativa jako například snížení výkonu. Aplikace vyvinuté hybridně také mohou hůře komunikovat s nativními funkcemi operačního systému a pro dostupnost k některým funkcionalitám je nutné instalovat aplikace třetích stran. Při vývoji hybridních aplikací taky není zcela možné dodržovat nejnovější trendy v mobilních zařízeních, protože nové funkcionality nejsou ihned dostupné pro hybridní frameworky. Uživatelé, kteří jsou zvyklí na práci s jistým operačním systémem, mohou být vyvedeni z míry vzhledem a prací s hybridní aplikací, jelikož UX těchto aplikací se dost odlišuje od zavedeného standartu nativních aplikací.

Hybridní aplikace na mobilní zařízení je vhodné vytvářet, pokud firma nemá k dispozici dostatečné finance nebo tým pro realizaci dvou nativních aplikací pro Android a pro iOS. Hybridní aplikace ovšem může být v jistých ohledech i preferovaný přístup vývoje, pokud firma ví, že bude chtít vyvíjet jednotnou aplikaci jak pro mobilní zařízení, tak i pro webové rozhraní.

3 FUNKCE MOBILNÍCH APLIKACÍ

Tato kapitola slouží především k popisu pokročilejší mobilních funkcí, na které lze v mobilních aplikacích často narazit a které jsou v této práci také použity.

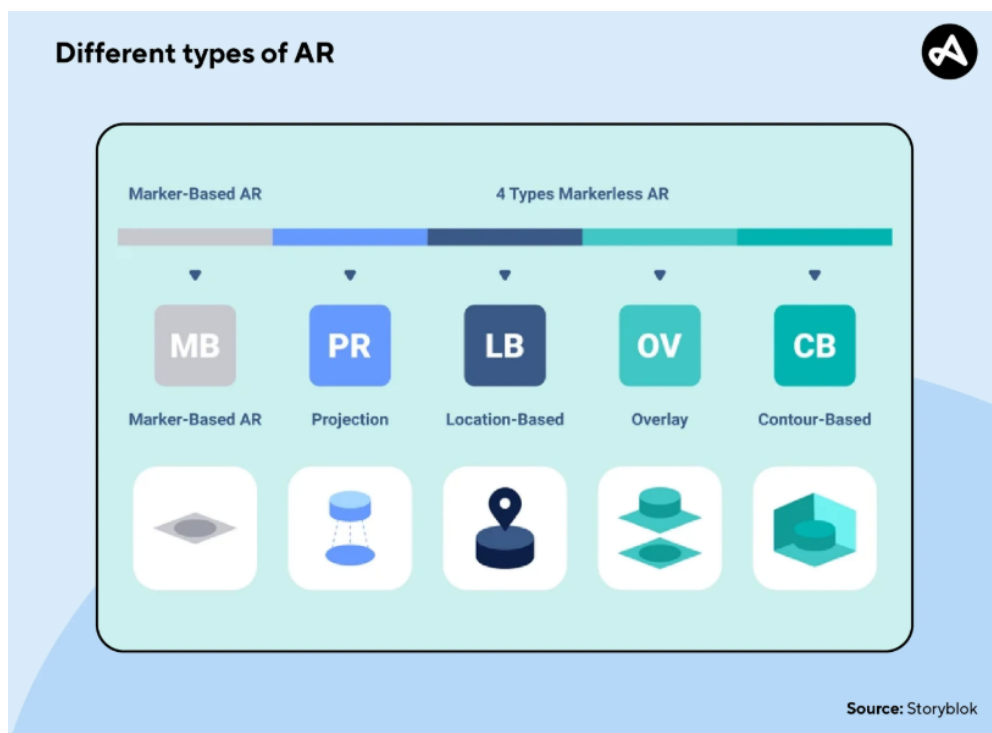
3.1 Rozšířená realita

Rozšířená realita, známá také pod anglickým označením Augmented Reality (AR), je technologie, která propojuje digitální obsah jako je text, grafika, audio nebo video s reálným prostředím, které uživatel pozoruje prostřednictvím zařízení, například mobilního telefonu nebo specializovaných brýlí. AR kombinuje fyzický a digitální svět, čímž vytváří pohlcující zážitek, při němž mohou uživatelé interagovat s 3D objekty zobrazenými ve svém okolí [21].

Tato technologie funguje na základě zařízení vybavených kamerou a senzory, které umožňují detekci objektů a lokalit v reálném prostředí. Detekované vzory, jako jsou jednoduché geometrické tvary nebo například lidský obličej, jsou snímány kamerou a dále zpracovávány AR softwarem. Jakmile je určen tvar, velikost a vzdálenost objektu, může být tato oblast doplněna o 3D model nebo jiný digitální prvek, čímž vzniká iluze jeho fyzické přítomnosti [21].

Existují různé přístupy k realizaci AR. Tyto přístupy je možné vidět na obrázku 3.1. Jedním z nejjednodušších je využití fixních značek (např. podobný QR kódům), které mají jednoznačně definované okraje a referenční body pro umístění digitálního obsahu. Pokročilejší systémy pak fungují bez těchto značek tzv. markerless AR, kde je nutné zapojit další senzory, jako jsou GPS, gyroskop, akcelerometr a další. Tato forma AR umožňuje širší škálu využití, avšak je technicky náročnější na implementaci [21].

AR technologie bez fixních značek se dále dělí podle principu fungování například na AR založenou na projekci, překrytí, konturách nebo lokalizaci. V souvislosti s rozšířenou realitou se často zmiňují i příbuzné pojmy jako VR (Virtual Reality), MR (Mixed Reality) a XR (Extended Reality). Virtuální realita představuje kompletní ponoření uživatele do simulovaného prostředí, kde nevidí reálný svět. Smíšená realita pak spojuje principy AR a VR. Umožňuje souběžné vnímání i interakci s digitálními prvky v reálném čase. Pojem XR je zastřešující termín, který zahrnuje všechny výše uvedené technologie a obecně označuje rozšíření vnímání reality pomocí digitálních prvků [21].



Obrázek 3.1 Typy rozšířené reality

(zdroj: [21])

Mobilní zařízení dnes disponují všemi potřebnými senzory pro efektivní využití technologií rozšířené reality. Jednou z nejjednodušších a nejběžnějších forem AR v mobilních aplikacích je zobrazení doplňkových informací přímo přes kameru. Typickým příkladem může být čtení čárových nebo QR kódů, při kterém se místo pouhého textového výstupu zobrazí uživateli na displeji rozšířené informace, například název produktu, obrázek či další detaily, které se překrývají s reálným obrazem snímaným kamerou. Taková forma AR je pro uživatele snadno pochopitelná a praktická [22].

Mezi nejznámější aplikace využívající rozšířenou realitu patří bezesporu hra Pokémon GO, která kombinuje reálnou polohu uživatele s digitálními prvky zobrazenými na mapě. Při přiblížení k virtuálnímu objektu (např. Pokémonovi) se na obrazovce zařízení zobrazí jeho 3D podoba zasazená do reálného prostředí. AR technologie se tak osvědčila zejména v mobilních hrách, ale její potenciál sahá mnohem dál [22].

Rozšířená realita nachází uplatnění také v oblasti logistiky (např. při skenování produktů a sledování zásilek), v technickém servisu (např. zobrazování návodů pro opravy), nebo v návrhářství a interiérovém designu (např. vizualizace nábytku či oblečení ve skutečném prostoru). Očekává se, že trh s mobilními aplikacemi využívajícími AR bude i nadále výrazně

růst [22]. Případné uplatnění této rozšířené reality lze vidět na obrázku 3.2.



Obrázek 3.2 Použití rozšířené reality pro skenování zásilek

(zdroj: [22])

4 OPTIMALIZAČNÍ STRATEGIE

Tato část práce se zaměřuje na optimalizační strategie mobilních aplikací. Vysvětluje, proč je optimalizace důležitá, jaké cíle sleduje a jakými prostředky jich lze dosáhnout. Dále jsou představeny klíčové aspekty, které se týkají jak výkonu aplikace, tak i zviditelnění aplikace pro koncové uživatele. Závěrem je popsáno, jak lze výkon aplikace sledovat i po jejím nasazení do produkčního prostředí s cílem jejího dalšího zlepšování.

4.1 Cíle optimalizační strategie

Optimalizační strategie pro mobilní aplikace zahrnuje dosažení určitých předem stanovených cílů. Tyto cíle mohou být výkonové, ale i obchodní. Optimalizační strategie z hlediska používání aplikace zahrnuje metriky jako měření technického výkonu zaměřující se na rychlost, stabilitu a efektivitu, UX, vlastností aplikace a zabezpečení. Hlavním cílem efektivní mobilní aplikace je, aby se dala snadno vyhledat. Po nainstalování, aby se rychle zapíнала, rychle pracovala a nezasekávala se, a to napříč všemi možnými zařízeními, na které může být nainstalována. Aplikace musí zaujmout uživatele a uživatel s ní musí rád pracovat, aby ji v průběhu používání neodinstaloval. Na závěr, dobrá aplikace by měla směřovat k vysokým hodnocením v obchodu s aplikacemi [23].

Aplikace musí být viditelná na trhu jako například na App Store nebo na Google Play. Dobře optimalizované aplikace mají vyšší hodnocení a jsou tak viditelnější v obchodě s aplikacemi díky tomu, že mají více stažení. Uživatelé odinstalují aplikace z mnoha důvodů, ale mezi nejčastější důvod patří právě špatný výkon aplikace. Správná optimalizace redukuje šanci na to, že uživatel aplikace odinstaluje. To, jak si aplikace obstará také dost dokáže ovlivnit značku samotné firmy. Pokud je produkt přijat negativně, tak uživatelé můžou ve firmu ztratit důvěru [23].

Za správnou aplikací stojí také hlavně skvělé UX, na které jsou uživatelé zvyklí z předchozího používání podobných aplikací. Uživatelé předpokládají, že aplikace bude rychlá, responzivní a snadná na používání. Pokud je aplikace jen trochu matoucí, je dost velká šance, že uživatel bude frustrovaný a aplikaci odinstaluje. Správně laděné UX vede k udržení si uživatelů [23].

Je však nutné, aby před publikací aplikace proběhlo výkonové testování. Výkonové testování pomůže identifikovat, kde jsou v aplikaci problémy, kde se zasekává a co ji zpomaluje.

Díky těmto testům je možné měřit dobu odezvy, použití paměti a zatížení procesoru. Toto testování by mělo být provedeno ve více fázích vývoje a mělo by být provedeno zavčasu, aby se stihly opravit chyby před nasazením verze do produkce [24].

Velikost aplikace dost ovlivňuje její samotný výkon. Velká velikost dost zvyšuje dobu načítání, což vede k frustraci uživatelů. Snížit velikost aplikace je možné odstraněním redundantního kódu, popřípadě komprimováním obrázků nebo video či audio souborů. Pro menší aplikace je typické, že se načtou mnohem rychleji [24].

To, že samotný kód aplikace by měl být optimalizovaný je samozřejmostí. Kód by měl obsahovat, co nejméně http dotazů a měl by tak redukovat spotřebu zdrojů. Často používaná data by měla být cachována pro rychlý přístup. Tato optimalizační technika dokáže dost snížit dobu načítání a dokáže zrychlit aplikaci. Tyto optimalizace v kódu, také dokážou zredukovat konzumaci baterie [24].

Minimalizace spotřeby baterie je kritický krok ke zlepšení výkonu. Pro snížení spotřeby baterie je nutné limitovat procesy, které běží na pozadí, redukovat počet a složitost animací a efektů v aplikaci. Tyto opatření drasticky zlepší UX [24].

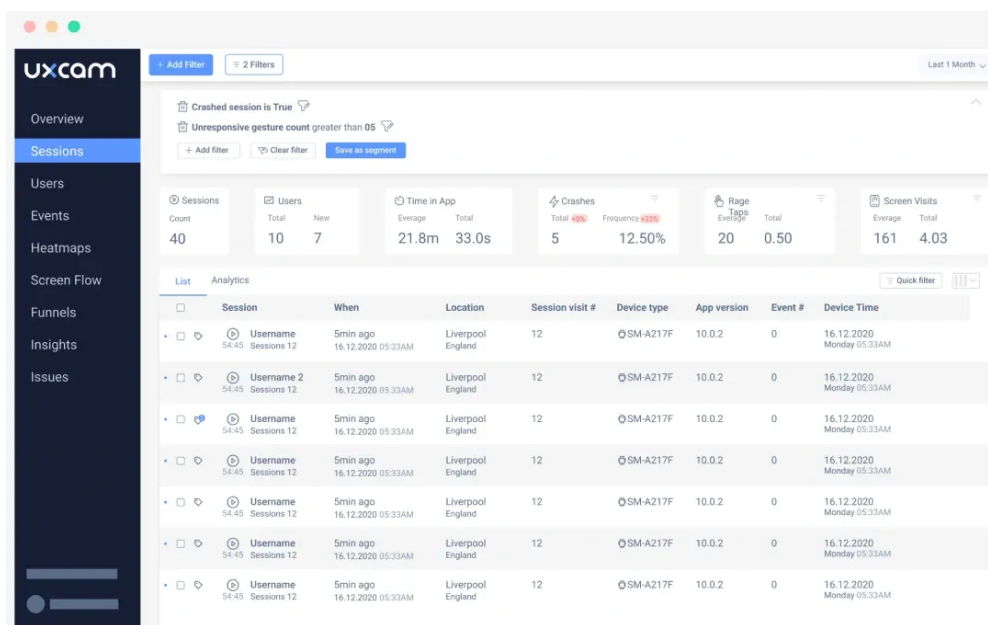
Ačkoliv je to náročný proces, je nezbytné aplikaci otestovat a optimalizovat pro různá zařízení. Každé zařízení má jiné specifikace, např. displej a paměť. Jakožto minimální opatření pro bezproblémové použití aplikace, by měla být aplikace otestována na různých zařízeních s rozdílnými displeji [24].

Další důležitá část, která dokáže dost odradit uživatele od používání aplikace, jsou pády. Aplikace by měla být ošetřena vůči pádům aplikace, aby k nim nejlépe nedocházelo a pokud již dojde, tak je nutné tyto pády monitorovat a ihned opravovat [23].

Když již nastanou nějaké problémy s publikovanou aplikací, tak je nutné na tyto problémy rychle reagovat. Uživatelé očekávají, že jakékoliv chyby budou v aplikaci rychle opraveny a že budou pravidelně vydávány aktualizace [24].

Pro již běžící aplikace je také nutné stanovit monitoring, který bude sledovat běžící procesy. Monitorování je neustále běžící proces, který probíhá nad nasazenou aplikací. Tento proces dokáže proaktivně identifikovat problémy v aplikaci, kterým uživatelé čelí. I malé chyby či občasné záseky by měly být logovány [23]. Proces monitorování je znázorněn na obrázcích 4.1 a 4.2.

Pro mnoho uživatelů je v aplikaci důležitá personalizace. Uživatel předpokládá, že se jim



Obrázek 4.1 Monitorování návštěvnosti a pádu aplikace

(zdroj: [25])

aplikace přizpůsobí, že tam bude mít vše potřebné a nic navíc. V běžných aplikacích tohoto lze dosáhnout pomocí doporučení obsahu, který uživatel nejčastěji navštěvuje [23].

Jako jeden z dalších cílů by měla být bezpečnost aplikace. Aplikace by měla poskytovat dostatečné zabezpečení a šifrování dat. Ukládání citlivých informací by také nemělo být děláno na klientské straně. Bývá zvykem, že klientská strana je méně zabezpečená a snadno napadnutelná. Vydaná aplikace by také měla být neustále monitorována kvůli potenciálním hrozbám [23].

Na závěr ještě jeden z cílů by měly být push notifikace. Tento typ notifikací, přichází uživatelům buď automaticky díky podmínkám v kódu nebo jsou spuštěny manuálně programátorem. Mají podobu jednoduchých upozornění, které se zobrazují uživatelům na mobilním zařízení v horní liště. Push notifikace jsou velice často používány v mobilních aplikacích a dokážou být jak k dobru, tak i na škodu. V mnoha případech notifikace zlepšují interakci mezi uživatelem a aplikací, ale pokud aplikace bude odesílat push notifikace příliš často a příliš mnoho, tak budou spíše uživatele frustrovat. Je nutné tedy odesílat převážně personalizované a relevantní notifikace. V případě notifikací je nutné se soustředit také převážně na urgentnost dané notifikace a jak častý je jejich příchod. Hlavní problém je, když se notifikace opakují [23].

4.2 Zviditelnění aplikace

Zviditelnění aplikace je jeden z klíčových cílů k úspěchu. Aplikace zveřejněné na trhu s aplikacemi by měly být snadno vyhledatelné. Při tvorbě nových aplikací se dost vyplácí sledovat novodobé trendy a predikovat tak uživatelské chování. Vývojáři, kteří vydávají aplikaci, by měli vhodně zvolit klíčová slova pro svou aplikaci, aby byla snáze vyhledatelná a aby při vyhledávání byl její název napovídán. Klíčová slova hrají důležitou roli při vyhledávání aplikací v obchodě a měla by být vhodně zakomponována do tzv. metadat aplikace, mezi která patří například její název a popis. Již publikovanou aplikaci je nutné monitorovat a měřit tak její návštěvnost a stahování, na výsledky tohoto monitorování by se pak mělo řádně odpovídat [23].

Pro větší dopad a získání většího publika by aplikace měla být přeložená do více jazyků. Stránka dané aplikace by pak měla být podpořena o výbornou grafiku s ikonami, obrázky a videi z aplikace. Pro zvýšení počtu stažení je vhodné využít také A/B testování, při kterém jsou porovnávány různé varianty vizuálních prvků a popisů aplikace s cílem zjistit, které z nich dosahují nejlepších výsledků. A/B testování je metoda, která umožňuje rozdělit uživatele do několika skupin a každé z nich zobrazit jinou verzi zvoleného prvku, například pomocí ikony, screenshotu nebo popisu. Na základě měření chování uživatelů (např. míra prokliků nebo instalací) lze poté vyhodnotit, která varianta je efektivnější. Aplikace by také měla v jistých intervalech uživatelům připomínat, aby aplikaci ohodnotili a přidali zpětnou vazbu [23].

Pro promování aplikace je nutné vytvořit marketingové kampaně, nejlépe ze začátku krátkodobou kampaň, která v krátkém období pouze pár dnů, donutí skupinu uživatelů si stáhnout a nainstalovat vyvíjenou aplikaci. Tento náhlý nárůst ve stahování podpoří vyhledávací algoritmy v obchodě a zajistí, aby tak aplikace byla lépe viditelná [23].

Zviditelňování aplikace je nepřetržitý proces, který jen tak neskončí. Aplikaci je nutné stále testovat, obohacovat a optimalizovat jak z pohledu výkonu, tak i pro vyhledávací algoritmy [23].

4.3 Optimalizace výkonu

Optimalizace výkonu přichází ve více formách. Je nutné se soustředit na zátěž procesoru, spotřebu paměti a baterie. I přes to, že novodobé aplikace a nové technologie vyžadují mít

silnější procesor a specifikace zařízení, tak to neznamena, že by se aplikace neměly optimalizovat. Každé zdroje, které procesor využívá navíc je nutné redukovat [25][26].

4.3.1 Start aplikace

Start aplikace je jedna ze zásadních vlastností, která dokáže ovlivnit, jak uživatel vnímá aplikaci. Pro zredukování rychlosti náběhu aplikace je možné odstranit přebytečné frameworky, které aplikace stahuje. Je vhodné zajistit, aby aplikace stahovala při prvním spuštění pouze to, co je nutné. Náročné procesy na pozadí by se také nejlépe měly provádět asynchronně pro zlepšení výkonu. Když již byla aplikace jednou spuštěna a znovu se načítá, je nutné, aby si zařízení uchovala v paměti stav aplikace, ve kterém se nacházela při ukončení. Uložený stav v paměti pak umožní tomu, aby se některé prvky v aplikaci nevykreslovali znovu [25].

4.3.2 Offline mód

Offline mód je velice důležitý pro mnoho uživatelů, protože mohou takto přistoupit ke svým datům i přesto, že nemají zrovna k dispozici internetové připojení. Offline mód většinou i zajistí, že uživatelé danou aplikaci budou používat častěji a budou ji vnímat pozitivněji. Aplikace by si tak měla ukládat uživatelský postup v aplikaci pro případ, že nebude mít přístup k internetu. Offline mód nejenže zvyšuje uživatelský komfort, ale také rychlost odezvy v aplikaci, jelikož aplikace je navržena tak, aby si uchovávala potřebná data lokálně [25].

4.3.3 Procesor

Při používání aplikace může občas dojít k náhlému nárustu využití procesoru, což může způsobit krátkodobé zpomalení nebo zaseknutí aplikace. To se stane, pokud aplikace v daný moment vyžaduje více zdrojů procesoru než obvykle. Stává se to běžně při hraní her, zpracování fotek, pořizování videa, streamování nebo spuštění složitých algoritmů. Tyto aktivity také mohou procesor zahřát a snížit jeho výkon. Optimalizování výkonu procesoru dokáže být velice náročné, protože ne vždy je jisté proč dojde k náhlému nárustu využití procesoru. Toto chování dokáže být často nepředvídatelné a je těžké identifikovat přesně příčinu. Tyto výkony také drasticky mohou zvýšit spotřebu baterie zařízení a pokud tento nárůst energie trvá, zařízení se může zahřát a všechny procesy v aplikaci se můžou zpomalit nebo i hůř, aplikace může spadnout. Důvod proč na některých zařízeních je výkon procesoru takto negativně ovlivněn může také samozřejmě být frekvencí procesoru a počtem jader v procesoru. Ovšem

pokud procesor využívá příliš mnoho zdrojů tak je vždy nutné hledat příčiny [25][26].

První věc, co může programátor udělat, pokud chce tento problém řešit, je zjednodušení algoritmů a optimalizace kódu. Je nutné analyzovat, co kód vykonává za operace, jelikož ne každý kód bude moci využít efektivně zdroje každého zařízení. Na některých zařízeních může aplikace jet plynule a na některých ne. Je nutné tedy kód přizpůsobit pro dané zařízení a zbytečně nepřetěžovat procesor, pokud to není nutné. Procesy, které běží na pozadí při startu aplikace také mohou zpomalit načítání aplikace. Testování výkonu nyní tradičně probíhá na emulátorech, ale ty nejsou vždy schopné zachytit vady skutečného zařízení. Pokud je zařízení až příliš přetěžováno, další věc, co programátor může udělat je zbavit se přebytných funkcionalit. Tyto funkce sice mohou vypadat hezky, ale zbytečně ovlivňují výkon. Pokud nejsou nutné pro funkcionalitu aplikace, tak je nejlepší se jich zbavit [25][26].

4.3.4 Paměť

Klíčová vlastnost každé aplikace, by měla být malá spotřeba paměti. Žádná aplikace by si neměla zbytečně alokovat zdroje, které nepotřebuje. Jeden z nejefektivnějších způsobů, jak redukovat využití paměti je odstranit přebytná data jako obrázky a videa, které nejsou potřeba. V kódu by převážně měly být použity správně optimalizované datové struktury, ve kterých vývojáři jsou schopni ukládat data jako například pole a slovníky. Jedna z užitečných praktik při načítání stránky, nebo jakýchkoliv dat, je **lazy loading**. Díky tomuto typu načítání dat jsou při inicializaci stránky, načteny pouze data, která jsou potřeba a zbývající data se načítají postupně až jsou třeba. Při vývoji aplikace by se průběžně mělo provádět profilování aplikace, jestli náhodou aplikace neobsahuje memory leaky, které by bylo nutné ihned ošetřit [27]. Memory leak se objeví, když program není schopen uvolnit místo v paměti, které nepotřebuje.

Správa paměti spočívá na procesu alokace a dealokace paměti na zařízení. Efektivní správa paměti definuje, jak plynule aplikace poběží a jak je kritická pro vývoj. Pro snížení využití paměti by v aplikaci měly být implementovány cachovací mechanismy, které redukovat využití paměti a zlepšují výkon. Díky cache paměti je možné data získat i mnohem rychleji. Vyskytující se memory leak rapidně zvětší využití paměti a drasticky to sníží výkon aplikace. Aplikace by tak měly mít implementován tzv. garbage collector. Garbage collector je používán v mnoha programovacích jazycích a automaticky získává paměť, která již

není dostupná. Implementace těchto algoritmů může pomoci vyčistit paměť a zlepšit výkon aplikace. Na co je nutné si ještě dávat pozor při správě paměti je fragmentace. Fragmentace může vést k neefektivnímu použití paměti, a tedy poklesu výkonu [27].

4.3.5 Baterie

Neefektivní využívání baterie je jedním z častých důvodů, proč uživatelé aplikace opouštějí. Při vývoji je proto důležité dělat vše pro to, aby byla spotřeba energie co nejnižší. Spotřebu baterie je možné pozorovat na monitorovacích zařízeních jak pro Android, tak iOS. Tyto nástroje jsou dostupné v nativních vývojových prostředích pro dané platformy a poskytují podstatné informace o spotřebě baterie, které jsou nejlépe testované na fyzickém zařízení [28].

Dost často baterii zatěžují mobilní aplikace, které načítání data ze sítě. Jak tomu bývá zvykem, aplikace, která nepoužívá žádné cachování je nejvíce náročná na baterii zařízení. Jsou tedy tři způsoby, jak implementovat cachování na zařízení a předejít tak zbytečnému plýtvání baterie. Pokud se jedná o data načítaná přes síť, tak je možné implementovat tzv. http cachování, které dokonce nativní programovací jazyky nabízejí a je velice jednoduché. Mezi další způsoby, jak snížit spotřebu baterie při načítání dat, patří průběžné ukládání dat do lokálního úložiště nebo využití cache paměti zařízení. Tyto způsoby ovšem jsou již náročnější [28].

Baterie velmi často právě bývá zatěžována zpracováním velkých datasetů, které konzumují příliš paměti. Konzumace paměti, společně s memory leaky, přispívá k plýtvání baterie. Pro správu objektů je tedy i vhodné použít tzv. object pooling pro efektivnější správu paměti. Díky této metodě se objekty budou znovu používat, místo toho, aby se neustále inicializovaly nové objekty [28].

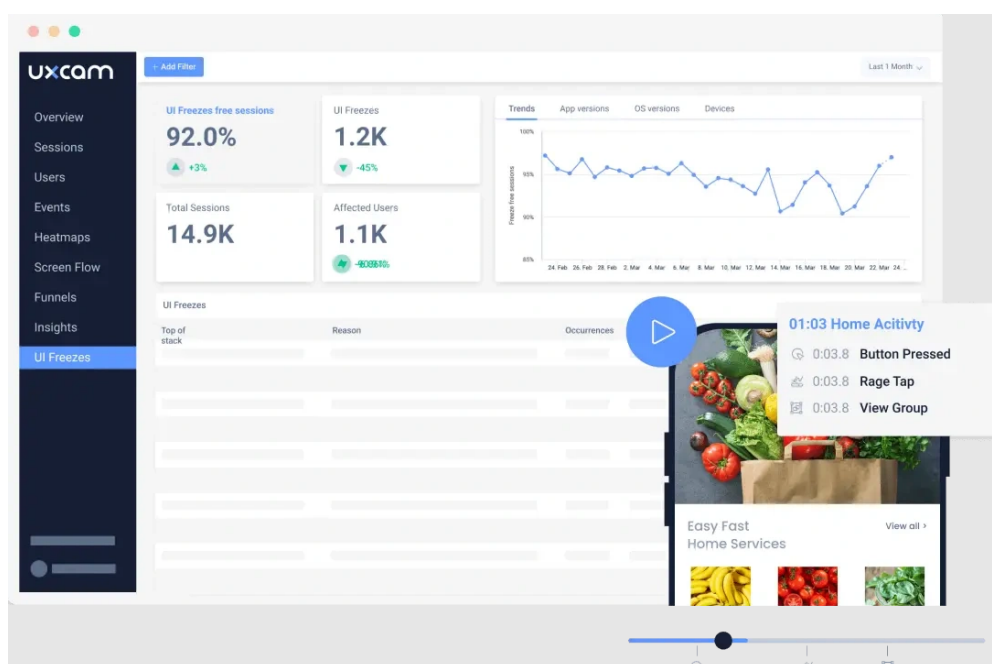
Vzhled aplikace a renderování prvků má také velký dopad na konzumaci baterie a je tedy lepší, aby na některých displejích aplikace běžela v temném módu, při kterém šetří energii [28].

Mezi velmi náročné operace na paměť patří například rozšířená realita, blockchain nebo 3D skenování. Pro tyto specifické operace je nutné použít již specifické optimalizační strategie, jako například podmínkové aktivování u rozšířené reality, selektivní procesování v případě použití blockchainu nebo pokročilé zpracovávací algoritmy v případě 3D skenování

[28].

4.4 Monitorování aplikace

Při nasazení aplikace je nutné monitorovat, co se v ní děje. Prioritou by mělo být sledování pádů aplikace, různých chyb a případných zpomalení. Tyto logy jsou hodně důležité k tomu, aby vývojáři mohli na dané problémy reagovat a opravit je včas. Takto lze monitorovat různé chyby, ale i výkon a UI. Také je možné pozorovat chování uživatele, na co kliká a jak se naviguje v aplikaci. Pro monitorování aplikace se využívají softwary, či webové aplikace a servery třetích stran. Mezi tyto monitorovací služby pro mobilní aplikace patří například software UXCam, kde lze aplikace zaregistrovat a mít tak přehled o chybách a výkonu. Logovací zprávy s výjimkami lze propojit k dalším službám jako například Slack, kde vývojáři ihned dostanou notifikace, že něco neběží. Mezi další monitorovací nástroje patří například Google Analytics, Mixpanel a Firebase [23][25].



Obrázek 4.2 Monitorování UI aplikace

(zdroj: [25])

5 METODOLOGIE TESTOVÁNÍ VÝKONU

Tato část práce se zaměřuje na základy testování softwaru. Vysvětluje, co testování znamená, jak probíhá celý proces a jaké typy testů se v jeho rámci uplatňují.

5.1 Základy testování

Testování je systematický proces vyhodnocování mobilních funkcionalit, výkonu, rychlosti a použitelnosti. Mobilní testy mají také vyhodnotit uživatelskou zkušenost, bezpečnost a kompatibilitu s ostatními zařízeními. Cílem testování je splnit požadavky, které jsou kladeny ve finální aplikaci [29].

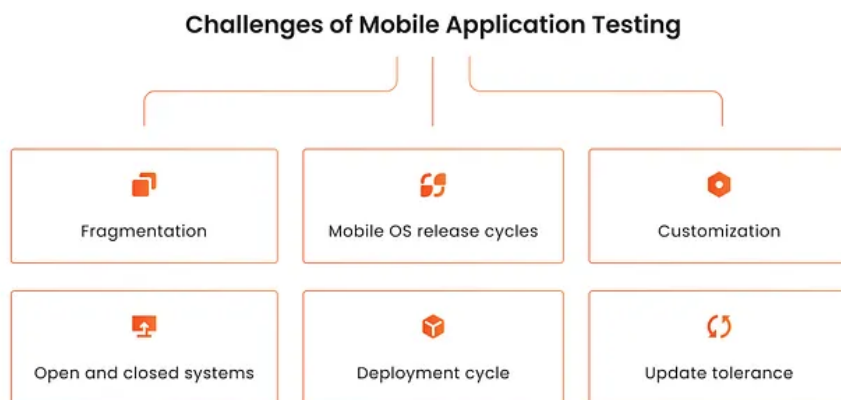
Testování je proces, který nelze z jakéhokoliv vývoje odstranit. Nejdůležitější funkce testování spočívá v odhalení chyb, které mají negativní vliv na chod aplikace a znepríjemňují chod aplikace uživatelům. Důvodem testování je objevit tyto chyby předčasně, abychom mohli těmto situacím předejít. Je také nutné provést testování na sadě příkladů, kde je nejen nutné otestovat chyby, ale i kolísavost internetového připojení nebo i zátěž na konkrétních zařízeních, aby si pak vývojáři byli skutečně jisti, že se neobjeví konkrétně nepříjemnosti. Špatně otestovaná aplikace může vyústit v potenciální ztrátu zákazníků, horší hodnocení a samozřejmě pokles ve výdělcích [29].

Mnohé začínající firmy krok testování přeskakují, kvůli nedostatku zdrojů, ale otázka o přeskočení testování by neměla ani vyvstat. Otestovaná mobilní aplikace dává lidem jistotu, že aplikace bude fungovat plynule. Na základě statistiky, která uvedla Parková [30], 70 % uživatelů opouští aplikaci, pokud je aplikace pomalá. Z celkové množiny uživatelů kolem 65 % uživatelů v danou firmu nebo značku ztrácí důvěru, pokud doručí špatný produkt. Už jen z těchto čísel je možné usoudit, že testování se nedá podceňovat. Velký zřetel by se měl brát na výkon aplikace, aby aplikace běžela co nejplynuleji. Nezáleží na velikosti aplikace, ať už je malá nebo velká, jelikož i malé aplikace je nutné testovat a optimalizovat [30].

Brzké testování ve vývoji je mnohem levnější než opravování chyb v produkci na poslední chvíli. Pro malé firmy se může jednat o velmi katastrofální věc, která může danou firmu stát velké množství peněz a hlavně zákazníky. Všem chybám v aplikaci se dá předejít s kvalitním testováním, ve kterém jsme schopni detekovat chyby před nasazením aplikace do produkčního prostředí. Oprava detekované chyby je vždy levnější hned v moment, kdy je objevena, místo neustálého odkládání. Při obrovské konkurenci zde není místo na to si

dovolit mít v produkci chyby. Je nutné si uvědomit, že koncový uživatelé nejsou beta testéři aplikace. Pro mnoho firem je důležité se zviditelnit, co nejdříve. Ať se o produktu ví a ať se může propagovat. A ačkoliv je tato skutečnost opravdu důležitá, tak by neměla přijít za cenu neotestovaného produktu. Nelze vždy doufat, že se aplikace spraví v pozdější fázi vývoje, protože na prvním dojmu záleží a špatný výkon aplikace může poškodit reputaci celé firmy [30].

Aplikaci je nutné testovat v raných fázích vývoje, aby se předešlo k rozšíření chyb a aby se velké problémy brzo zachytily. Testování by nejlépe mělo probíhat na skutečných případech, nejlépe na takových, se kterými se uživatelé nejčastěji potýkají. Priorita by především měla být uživatelská zkušenost se zařízením a mělo by se nejlépe testovat rozhraní a navigace v aplikaci. Pokud aplikace obsahuje nějaká uživatelská data, je nutné tyto data ošetřit zabezpečujícími testy. Za účelem častých aktualizací a přidávání nových funkcionalit aplikace by se mělo přidat do vývoje regresní testování. To pomáhá zajistit, že nové změny v aplikaci nenaruší již existující a funkční části systému. Regresní testování je forma softwarového testování, která se zaměřuje na ověřování, že nově provedené úpravy (např. nové funkce, opravy chyb) neovlivnily negativně dosavadní funkcionalitu aplikace. Obvykle se provádí opakovaně a automatizovaně po každé změně kódu [29]. Na jaké překážky při testování je možné narazit, je vidět z obrázku 5.1.



Obrázek 5.1 Překážky v testování

(zdroj: [30])

Na jednu z věcí, na kterou si testéři musí dát pozor, je v jakých intervalech jsou vydávány aktualizace operačních systémů jednotlivých zařízení. Především na trhu s Androidem

existuje mnoho různých vydavatelů se zařízeními, které mají různé hardwarové specifikace. Výrobci se snaží přijít s inovativními technologiemi co nejdříve. Při vývoji mobilní aplikace to může přinést mnoho výzev, obzvláště, když výrobce představí doposud neviděnou funkcionalitu [30].

5.2 Proces testování

Pro začátek testování, místo toho, aby vývojář rozuměl potřebám zákazníka, je zcela nutné, aby byl vývojář seznámen s funkční aplikací a jak by se měla chovat. Je důležité vědět, co bude aplikace vykreslovat, co bude vykonávat a je tedy nutné identifikovat potenciální hrozby, které mohou nastat na různých typech zařízení. Aby byl vývojář schopen správně otestovat aplikaci, je nejdůležitější stanovit si cíle a pro tyto cíle stanovit testy. Zároveň s těmito testy je nutné stanovit několik testovacích případů. Tyto případy slouží jako sada systematických scénářů a podmínek, o které aplikace musí být testována. Vývojář musí podchytit každé místo aplikace, nejlépe, kde uživatel pracuje s aplikací. Na základě těchto testovacích případů je pak vhodné vybrat testovací nástroje, které budou odpovídat žádaným potřebám. Dále už je potřeba pouze spustit dané testy a metodicky je provádět. Za každým spuštěným testem by měl být výsledek zdokumentován a měly by se pozorovat jakékoliv abnormality. Jakožto jeden z posledních kroků je zde nutná přesnost. Některé typy testů naštěstí mohou být, a i měli by být, automatizovány. Automatizace určitých kroků při testování, dokáže razantně zvýšit produktivitu. Je dobrou praxí do vývoje integrovat jak automatické, tak i manuální testy. Každý typ testování má své výhody i nevýhody. Manuální testování se nejlépe hodí například na testování skutečných událostí, které mohou nastat. Pokud automatické testy mají vliv na běh aplikace, tak by nejlépe měly být vykonávány v době, kdy je nejméně provozu v aplikaci, pokud je aplikace již nasazená [29][31].

Na testování aplikace není vývojář sám a nedílnou součástí testování jsou také testeři, kdy v některých případech testery jsou také skuteční uživatelé. Součástí testování je tedy i sbírání zpětné vazby od uživatelů [31].

Součástí závěrečného QA testování (Quality Assurance – zajištění kvality) je i testování spotřeby baterie zařízení, při kterém tester sleduje, jak se spotřeba energie mění v různých scénářích používání aplikace. Dále testování hardwarové konfigurace, při kterém je nutné zkontrolovat kompatibilitu aplikace s nastavením různých zařízení. Základní je i pohoto-

vostní testování, při kterém se zkontroluje, jak rychle se aplikace dokáže probudit z pohotovostního režimu mobilního zařízení. Na závěr probíhá tzv. beta testování, kdy je aplikace otestována v reálném prostředí pro získání zpětné vazby ohledně UI a funkcionality [30].

5.3 Typy testů

Globálně se testování mobilních aplikací dělí na dva typy, hardwarové a softwarové. Hardwarové testování spočívá především v testování interních komponent, jako fotoaparát, paměť, úložiště, Wi-Fi, a dokonce i rozložení UI aplikace na různých rozlišeních zařízení. Softwarové testování se soustředí již na samotnou funkcionalitu aplikace [30].

K testování se dá přistoupit třemi způsoby a spočívá tak v teorii testování buď bílé skřínky, testování černé skřínky nebo testování šedé skřínky. Testování bílé skřínky spočívá v tom, že daný vývojář zná interní logiku a strukturu kódu, se kterou pracuje. Naopak v případě černé skřínky se aplikace testuje ne z pohledu vývojáře, ale uživatele, který nemá ponětí o tom jak daný kód a architektura vypadá. Ještě existují i testování šedé skřínky, které kombinuje tyto dva přístupy [30].

5.3.1 Cílené testování

Pro testování mobilní aplikace se rozlišuje několik typů testování a způsobů. Existuje funkcionální testování, výkonnostní testování, testování zabezpečení a testování kompatibility. Dále je možné ještě provádět testy použitelnosti, test instalace, průzkumné testy, testy úložiště, testy přerušení a testy vstupních polí neboli input testy [29][30].

- Ve **funkcionálním testování** jde o testování vlastností aplikace, zda pracují tak jak je definováno v cílech, co má aplikace splňovat. Mezi to spadá testování UI, databáze, volání API, zpracování dat apod. [29].
- Do **testování výkonu** spadá testování rychlosti a testování výkonu aplikace za určitých podmínek, jako například za nízkého stavu baterky, špatného připojení k internetu nebo testování vysoké zátěže [29].
- **Testování zabezpečení** je o určení a detekování potenciálních slabín v aplikaci, která mohou ohrozit uživatelská data [29].

- **Testování kompatibility** slouží převážně k tomu otestovat aplikaci na více zařízeních, různých velikostech obrazovky nebo i operačních systémech [29].
- **Testy použitelnosti**, jak jméno napovídá, slouží k otestování, jak moc je aplikace uživatelsky přívětivá a zda je bez chyb [30].
- **Instalační testy** slouží pouze k otestování, zda hladce probírá proces instalace a aktualizace aplikace [30].
- **Průzkumné testy** se zaměřují na detekci klíčových chyb [30].
- **Testy úložiště** slouží k převážně k otestování, jak je aplikace schopna vykonávat funkce týkající dat a úložného prostoru, jako například ukládání dat, čtení dat a mazání dat. Testy by měly být provedeny i na zařízeních, kde je například málo úložného prostoru, aby bylo zjištěno chování aplikace v hraničních případech [30].
- **Testy přerušení** slouží především k identifikaci, kolik průměrně za den dostane uživatel zpráv a notifikací z aplikace. Tyto narušující zprávy je nutné držet na minimu a tester nejen musí brát v potaz výkon a chyby, ale i zda jsou vhodné pro uživatele [30].
- Další testování je **testování vstupních polí**, kdy tester by měl otestovat nejlépe krajové případy, které by mohly nastat při vyplňování takových polí. Netýká se to ovšem jen formulářových políček, ale také i gest, hlasu a jakýchkoliv senzorů, které aplikace nabízí [30].

Při testování mobilní aplikace je nezbytné, aby každá firma vyhodnotila klíčové ukazatele výkonnosti, které jsou zásadní pro posouzení efektivity aplikace. Tyto ukazatele, označované jako KPI (Key Performance Indicators – klíčové ukazatele výkonnosti), poskytují přehled o výkonnostních aspektech aplikace a pomáhají vývojářům při její optimalizaci. Mezi tyto body ve výkonnostním testování patří například doba odezvy, pro kterou obecně platí, že uživatelské prvky by se měly rychle načítat. Dále mezi tyto KPI patří pády aplikace, propustnost, čas prvního bitu, latence, spotřeba baterie, spotřeba paměti, doba načítání, DNS lookup a start aplikace [32].

Z těchto KPI se vyplatí rozebrat, třeba propustnost, u které je nutné si měřit kolik dat se přesouvá přes síť v daném čase. Díky naměřeným hodnotám je snazší identifikovat problémy

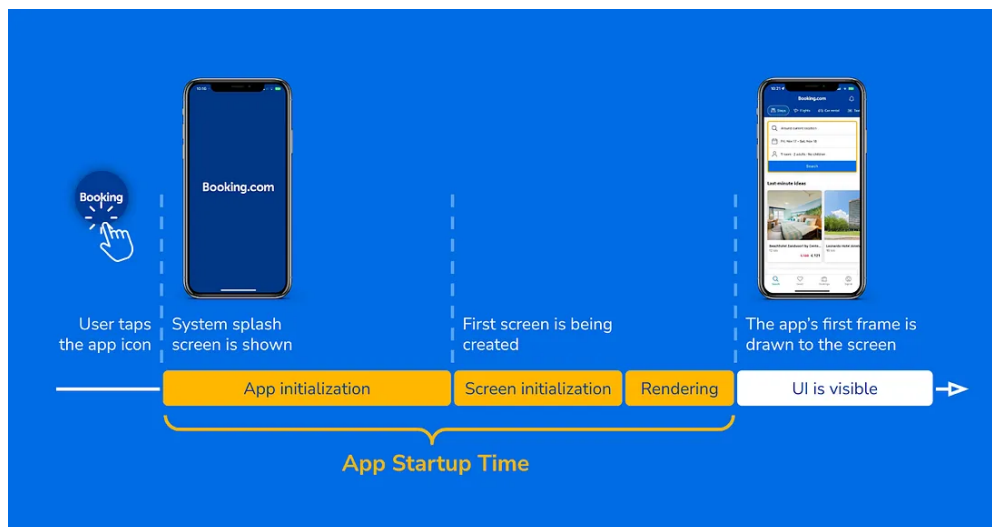
v různých situacích. Čas prvního bajtu neboli TTFB (Time To First Byte) je měřený čas od spuštění požadavku do prvního bajtu odpovědi, který je přijat. Slouží to jako indikátor pro identifikování, jak rychle probíhá komunikace mezi serverem nebo infrastrukturou na backendu [32].

Dále je vhodné monitorovat, jak aplikace spotřebovává baterii nebo jak moc aplikace využívá paměť zařízení. Příliš vysoká spotřeba paměti aplikace může způsobit pády, nebo může negativně ovlivnit celkový výkon a běh aplikace. S tímto souvisí doba načítání, při které se u měření soustředí na to, jak dlouho trvá, než se uživateli obsah plně načte na obrazovce. Pro zlepšení doby načítání se vyplatí optimalizovat dotazy na načítání dat, minimalizovat úkony, které jsou náročné na zdroje aplikace při startu a také optimalizovat render elementů [32].

Další klíčová věc je měření DNS lookupu, neboli doby, po jakou trvá DNS rezoluce, jakmile uživatel spustí aplikaci. Tento faktor také může ovlivnit rychlost načítání aplikace. DNS lookup je proces, při kterém dochází k vyhledávání IP adres na základě doménových jmen [32].

Jedním z klíčových ukazatelů, které je důležité sledovat, je doba, za kterou se aplikace načte do svého výchozího stavu poté, co uživatel klikne na její ikonu. Při měření startu aplikace rozlišujeme tři stavy, **cold start**, **warm start** a **hot start**. **Cold start** je stav, kdy se aplikace načítá po kliknutí na ikonu do stavu vykreslení prvotního snímku a nemá v mezi paměti uloženy žádné informace, které by zrychlily načítání. Tento stav je více znázorněn na obrázku 5.2. **Warm start** funguje na podobném principu jako běžné spuštění aplikace, ale s tím rozdílem, že aplikace má již některá data nebo stav uložený v paměti. Díky tomu je její načtení rychlejší. Tento typ spuštění se obvykle neměří, protože neodráží reálný výkon při prvním spuštění aplikace. Na závěr **hot start** je stav, kdy aplikace již běží a jen se přesune z procesu, co běží na pozadí do popředí. Tento čas je nejrychlejší, jelikož aplikace má již vše potřebné načtené [33].

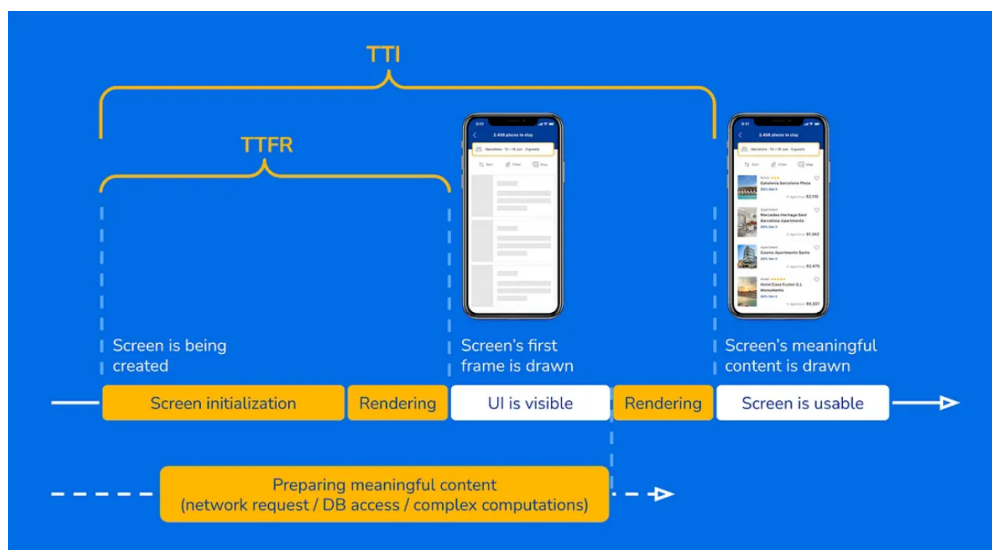
Další parametry, které je nutné testovat se týkají renderovaných elementů. Již bylo zmíněno, že je nutné měřit a optimalizovat dobu od startu aplikace do jejího výchozí stavu. Další parametry, které je potřeba měřit jsou TTI a TTFR. Přesné rozdíly těchto dvou časů je možné pozorovat na obrázku 5.3. TTI je time to interactivity, neboli čas v milisekundách, který je měřen od vytvoření obrazovky až po první zobrazený obsah. Po uplynutí tohoto času, je zajištěno, že interní stav zařízení je načten, UI prvky pro uživatele jsou vyrenderovány a hlavní



Obrázek 5.2 Cold start aplikace

(zdroj: [33])

vlákno naslouchá pro uživatelský vstup. Tento měřený čas se také odvíjí od výkonu zařízení, úložiště, paměti a sítě. Pokud je tato doba příliš krátká, tak pro přesnou identifikaci se používá metrika TTFR neboli time to first render. TTFR je měřený čas, který začíná ve stejném bodě jako TTI, ale končí dříve, a to v okamžik vykreslení prvního snímku. Po dokončení TTFR většinou je na obrazovce zobrazováno nějaký načítání, pod kterým je schována komplexnější logika [33].



Obrázek 5.3 TTI a TTFR

(zdroj: [33])

6 TESTOVACÍ NÁSTROJE

Tato část práce se zaměřuje na testovací nástroje určené pro mobilní aplikace. Popisuje, na jakých platformách lze mobilní aplikace testovat, a představuje základní nástroje dostupné v rámci nativního vývoje. Zároveň se věnuje i nástrojům třetích stran, které mohou nabídnout rozšířené možnosti a efektivnější testování v různých scénářích.

6.1 Testovací zařízení

Mobilní aplikace lze nejlépe testovat na fyzickém zařízení, na kterém je možné otestovat, jak se aplikace skutečně bude chovat. Pro některé scénáře je to například i žádoucí, jako například při testu aplikace, kdy je zařízení skoro vybité, nebo plně nabité. Jenže testování na fyzickém zařízení dokáže být náročné obzvláště v případě testování Android aplikací, jelikož existuje velké množství zařízení, které je třeba otestovat. Nejen to, ale také vývojáři při rychlém vývoji potřebují vědět, jak jejich aplikace vypadá a jak se chová, bez nutnosti neustále zapínání na fyzickém zařízení. Tento problém řeší emulátory a simulátory. Ačkoliv by se mohlo zdát, že se jedná o tu stejnou věc tak je mezi nimi rozdíl [30][34].

6.1.1 Emulátor

Emulátor je typ softwaru, který slouží vývojovým, ale i testovacím týmům k otestování výkonu aplikace na základě hardwaru mobilního zařízení. Emulátory neslouží pouze k testování softwarových funkcí, ale do testování na emulátoru spadá i testování operačního systému, procesoru, paměti a baterie. Při testování na emulátoru je téměř jisté, že pokud je aplikace pomalá, tak bude i pomalá na fyzickém zařízení stejného typu, který emulátor replikuje. Emulátory jsou schopné zcela replikovat vybrané fyzické zařízení i po hardwarové stránce a jsou tak dobré na všechny typy testů. Emulátory jsou dostupné pro testování Android zařízení. Při instalaci vývojového prostředí jako Android Studio je zároveň také nainstalován Android emulátor pro uživatelem vybrané konkrétní zařízení. Emulátory však bývají náročné na operační paměť počítače, a protože se snaží co nejvěrněji napodobit chování fyzického zařízení, může být testování pomalejší. Tyto emulátory nejsou dostupné pro Apple zařízení [30]. Příklad emulátoru je možné pozorovat na obrázku 6.1.



Obrázek 6.1 Android emulátor

6.1.2 Simulátor

Simulátor je typ softwaru, který slouží k otestování pouze softwarových funkcí a vzhledu. Simulátory jsou velice jednoduché a rychlé, ale nejsou schopny replikovat hardwarové funkce mobilního zařízení. Simulátory pouze napodobují chování mobilního zařízení a zobrazují běh aplikace na počítači bez nutnosti skutečného hardwaru simulovaného zařízení. Simulátory jsou tedy především používány k otestování nově přidaných funkcí. Také jsou hlavně používány při responzivním testování, při kterém se testuje, jak aplikace vypadá na různých rozlišeních. Tyto simulátory jsou dostupné na zařízeních od společnosti Apple pro testování iOS produktů. Na rozdíl od Android emulátoru, tyto simulátory je možné nainstalovat pouze na zařízeních společnosti Apple a Apple také nenabízí žádnou emulátorovou variantu. Hardwarové prvky zařízení je v případě iOS nutné testovat pouze na fyzickém zařízení [30]. Existují i simulátory pro Android zařízení, ale ty se zřídka využívají. Příklad simulátoru je možné pozorovat na obrázku 6.2.

6.2 Nativní testovací nástroje

Pro testování a měření zátěže aplikace v mnoha případech stačí použít testovací nástroje, které jsou dostupné v nativním vývojovém prostředí pro danou platformu. Tím je Android



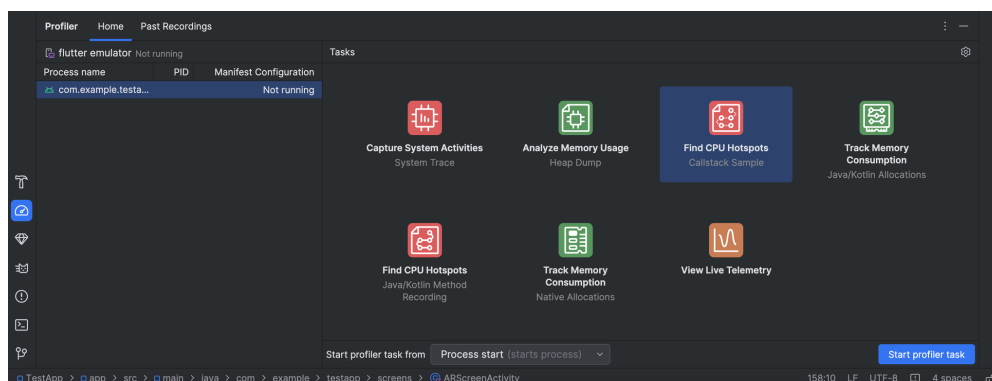
Obrázek 6.2 iOS simulátor

Studio pro Android nebo XCode a XCode Instruments pro iOS. I přesto, že je aplikace vyvíjena například jiným než nativním přístupem, tak je možné tyto testovací nástroje i tak použít. Multiplatformní a hybridní frameworky, také občas poskytují své testovací nástroje, ale nemusí být vždy kompletní a testování se tak vyplácí provádět v těchto nativních nástrojích [30][34].

6.2.1 Android Studio

Pro testování na Android zařízení se používá Android Studio. Jedná se o vývojové prostředí spravované společností Google a JetBrains. Android Studio je multiplatformní vývojové prostředí, které zvládá mnoho testovacích akcí a je snadné pro správu velkých projektů. Android studio je nativní vývojové prostředí pro tvorbu nativních Android aplikací. Toto vývojové prostředí obsahuje zabudovaný profiler, díky kterému je možné měřit několik výkonostních metrik jako například paměť, CPU, provoz na síti a spotřebu baterie. Pokud je Android Studio připojené k emulátoru nebo fyzickému zařízení, tak toto vývojové prostředí vypíše všechny procesy, které momentálně na daném zařízení běží. Běžící procesy je možné profilovat a testovat. Pro testování je nutné spustit aplikaci, následně ji vybrat v seznamu procesů a poté je nutné vybrat takzvaný profilovací úkon, pro měření konkrétní výkonostní metriky.

Tyto profilovací úkony je možné vidět na obrázku 6.3. Po vybrání úkonu je nutné spustit nahrávání a následně už stačí pracovat s aplikací jako běžný uživatel. Při používání aplikace jsou výsledky zaznamenávány. Všechny naměřené metriky je možné následně exportovat a porovnávat mezi sebou [35][36].

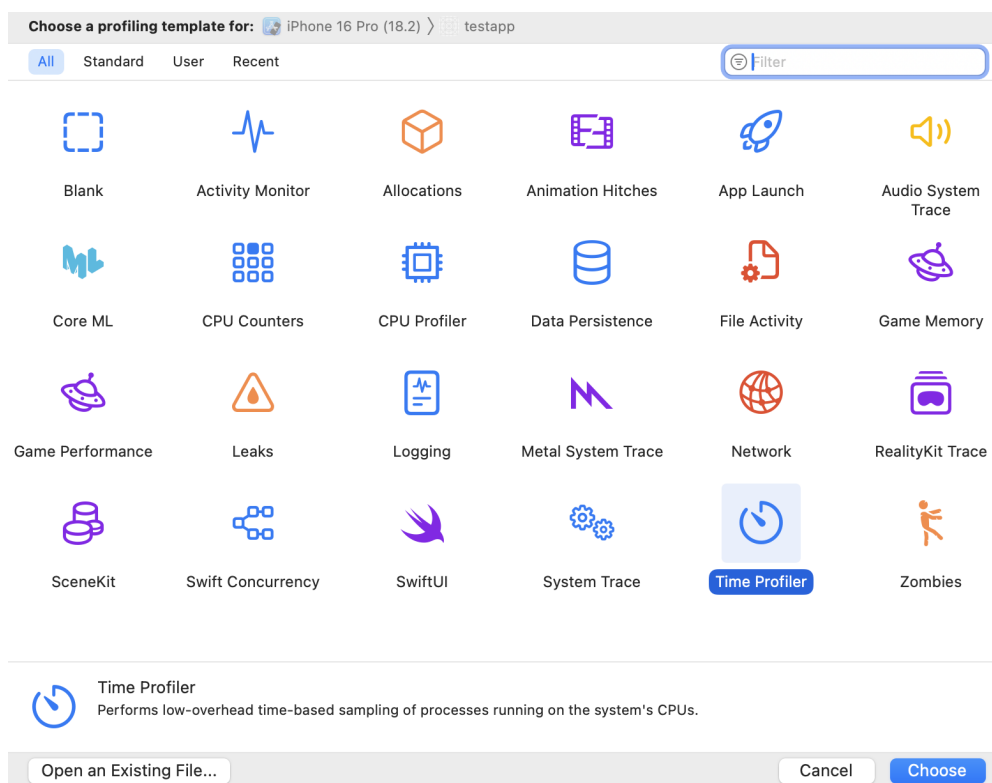


Obrázek 6.3 Android studio profiler

V Android studiu je možné měřit různé metriky. Monitorování CPU se hodí, pokud aplikace se zasekává nebo trvá dlouho na načtení v určitých momentech. Toto testování není dobré jen na sledování výkonu, ale i na hledání memory leaků. Android Studio je schopné ukázat spotřebu paměti a pokud využití konstantně roste, je téměř jisté, že se může v aplikaci nacházet memory leak. Ve vývojovém prostředí, jde vidět, kdy objekty byly vytvořeny, a kolik jejich inicializace nebo znovupoužití, spotřebovalo paměti. Vývojové prostředí také dokáže vizualizovat využití sítě. Prostedí také umožňuje měřit, jak dlouho trvá vykreslení prvků na obrazovce a zároveň dokáže měřit GPU. Tyto hodnoty je možné vypisovat do logů. Měření, jak dlouho trvá vykreslit prvky na displeji, je ovšem možné již provádět na daném zařízení. Jestli je na připojeném zařízení ve vývojářských možnostech povoleno profilování GPU, zobrazí se při běhu aplikace sloupcový graf, který znázorňuje, jak rychle se v čase vykreslují jednotlivé snímky. Pokud je vykreslování příliš náročné, lze i přímo v aplikaci zapnout náhled na prvky, které jsou vykreslovány více než jedenkrát. Důvod, proč jsou vykreslovány některé prvky vícekrát je například ten, že se prvky překrývají, a to může zdržovat vykreslování a vynucovat vícero vykreslení. Celý proces tohoto testování, lze také vidět i v Android studiu. Vývojové prostředí také obsahuje časovou osu používání aplikace s voláními systémových a uživatelských funkcí [35][36][37].

6.2.2 XCode a XCode Instruments

Pro testování iOS zařízení se používá software XCode a XCode Instruments. Jedná se o specifický software vyvinutý firmou Apple pouze pro zařízení Applu. XCode slouží převážně pouze jako vývojové prostředí a pro správu projektů, i přesto XCode přichází s nainstalovaným simulátorem. Základní měření je možné provést v aplikaci XCode přímo v kódu aplikace pomocí measure metod, které je možné přidat novou konfigurační sadou v projektu. Tyto metody jde poskládat do testovacích bloků a pak s nimi jde zacházet jako s obyčejnými testy, kdy při spuštění dané úlohy, se spustí daný blok a ten následně vypíše, jak dlouho trvalo tento blok kódu vykonat a pokud bylo stanoveno, tak i výsledky porovná s výsledky, kterých se mělo docílit [38][39].



Obrázek 6.4 XCode Instruments

Ovšem k čistému testování se zpravidla využívá software XCode Instruments. Tento software poskytuje komplexní náhled a obrovskou množinu testovacích úkonů pro aplikaci. XCode Instruments podobně jako Android Studio nabízí sadu testovacích balíčků, jako například pro měření výkonu, správu paměti, provozu na síti, sledování memory leaků nebo spotřebu baterie. Vizuál XCode Instruments společně s testovacími balíčky je možné vidět na obrázku 6.4. Tyto úkony jsou zde nazývány jako instrumenty a jsou mnohem podrobnější

rozčleněny na jednotlivé úlohy, než tomu je například v případě Android Studia. Pro základní testování ovšem mnoho z těchto instrumentů vůbec není potřeba a jsou spíše užitečné pro specifické nebo komplikované úlohy. Většinu těchto instrumentů také není možné spustit na simulátoru, a i pro ty, které je možné spustit na simulátoru je i tak doporučeno použít fyzické zařízení kvůli lepším a přesnějším výsledkům. Práce s instrumenty probíhá stejně jako v Android studiu. Vývojář si vybere konkrétní metriku, kterou chce měřit, spustí aplikaci, spustí nahrávání a následně se mu vypisují informace o výkonu aplikace, volání funkcí apod. ve stejném stylu [38][39].

6.3 Nástroje třetích stran

Testovací nástroje třetích stran se dosti liší od nativních nástrojů jako Android Studio a XCode Instruments a to převážně proto, že cílí konkrétně na něco jiného, jako například na vstupy od uživatele. Tyto nástroje se také dají použít k měření výkonu v aplikaci. Ale ve většině případů už se jedná o nasazenou aplikaci, třeba jenom ve fázi beta testování, kdy uživatelé aplikaci prochází a testují. Tyto nástroje pomáhají monitorovat a logovat chyby již v běžících aplikacích. Nejen to, ale i dokážou monitorovat s čím uživatel interaguje. Jejich kvalita také spočívá v nastavování automatických testů nově přidaných funkcionalit. Některé nové a moderní testovací nástroje se dokonce snaží, aby nastavení bylo co nejjednodušší za použití, co nejméně kódu. Lidé, co tedy tyto nástroje obsluhují, nepotřebují až tak rozsáhlé znalosti v kódování. Tyto pokročilé nástroje, stejně jako i nástroje pro monitorování bývají placené a mají pouze zkušební verzi, která je zdarma. Mezi známé varianty nástroje patří například Katalon, Appium, Espresso, XCUI Test a Robotium [40].

II PRAKTICKÁ ČÁST

7 NÁVRH TESTOVACÍ APLIKACE

Součástí praktické části bylo vyvinout třikrát stejnou aplikaci, jednou hybridně a dvakrát nativně pro Android a iOS. Tato část práce vysvětluje, jakým stylem byly mobilní aplikace navrženy. Popisuje, proč aplikace obsahují konkrétní funkcionality a za jakým účelem byly tyto funkce zvoleny. Také obsahuje ukázky z aplikací, popisuje architekturu a zvolené nástroje k realizaci.

7.1 Návrh a cíl aplikace

Cílem práce bylo navrhnout jednotnou aplikaci, která bude implementována třemi odlišnými způsoby. První dvě varianty představují samostatný nativní vývoj pro platformy Android a iOS. Třetí variantou je hybridní aplikace navržená tak, aby byla spustitelná na obou platformách.

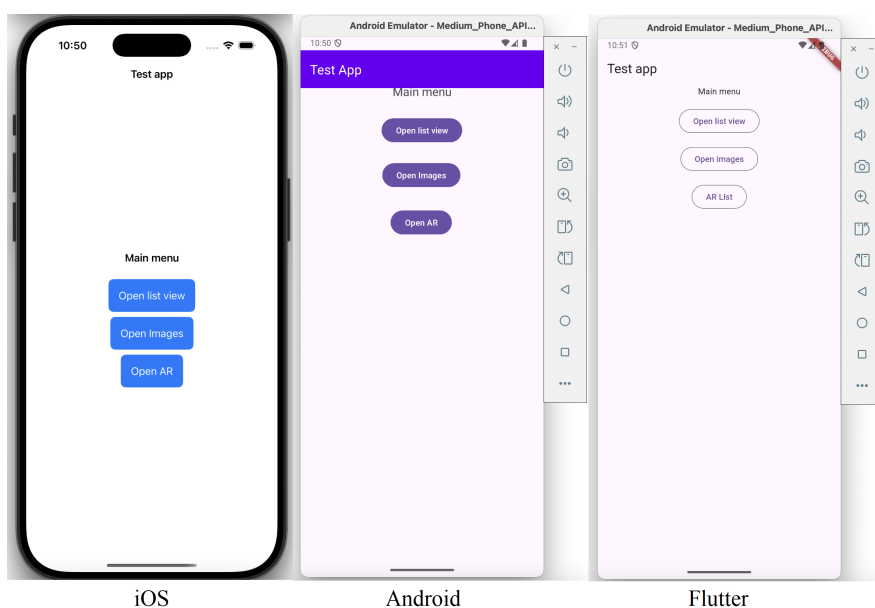
Tyto tři způsoby vývoje byly zvoleny z toho důvodu, že nativní přístup je v současnosti dominantní pouze na platformách Android a iOS. Vývoj mobilních aplikací pro jiné operační systémy je dnes zanedbatelný. Pokud je cílem oslovit uživatele na více platformách současně a zároveň minimalizovat náklady na vývoj, volí se často hybridní přístup, který umožňuje vytvářet aplikaci pro více systémů z jednoho společného kódu.

Aplikace ve svém finálním návrhu zůstává záměrně jednoduchá a má za cíl otestovat všechny základní funkcionality, se kterými se běžný uživatel mobilní aplikace obvykle setkává. Jednotlivé funkce na sebe nemusí nutně navazovat a ve skutečnosti ne všechny musí být současně obsaženy v jedné konkrétní aplikaci. Přesto se jedná o nejčastější prvky, které se v mobilních aplikacích objevují. Z hlediska funkcionality odpovídá návrh běžným jednoduchým aplikacím využívaným například zaměstnanci firem pro zobrazení nových požadavků nebo pro přidání nového požadavku.

Aby bylo možné výsledky testování mezi jednotlivými platformami objektivně porovnat, musí být všechny tři verze aplikace navrženy co nejvíce konzistentně. To se týká jak vizuálního zpracování a rozmístění prvků uživatelského rozhraní, tak i toku obrazovek a způsobu interakce. I přes rozdíly v technickém zpracování jednotlivých přístupů je důležité, aby uživatelská zkušenost zůstala co nejvíce podobná, a bylo tak možné vyhodnotit rozdíly ve výkonu, stabilitě a odezvě nezávisle na konkrétní implementaci.

7.2 Funkcionality aplikace

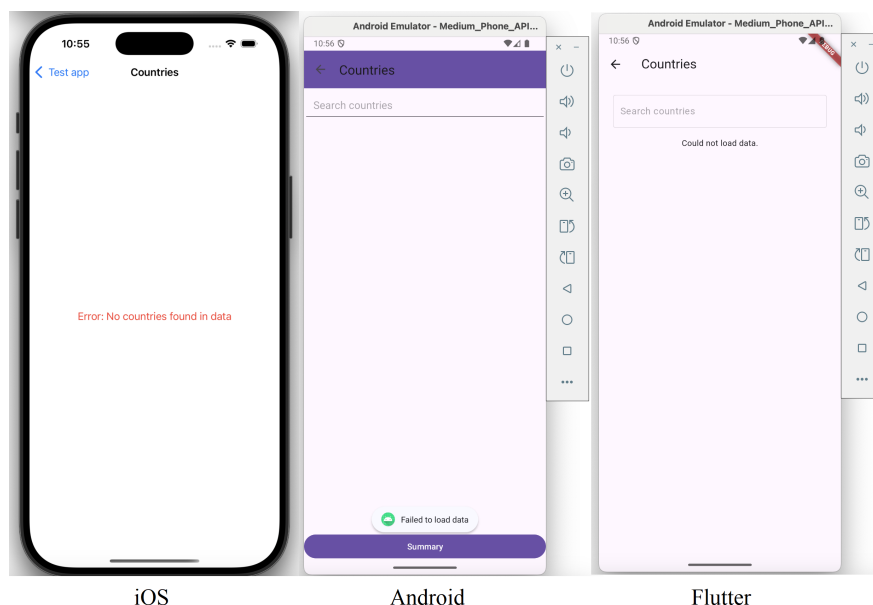
Navržená mobilní aplikace má základní strukturu s domovskou stránkou, která obsahuje tři tlačítka sloužící k navigaci mezi jednotlivými funkcemi. První možnost vede na obrazovku se seznamem dat načítaných z externího API. Položky v seznamu lze rozkliknout pro zobrazení detailního přehledu a k dispozici je také sumarizační stránka se souhrnnými informacemi. Druhá sekce zobrazuje seznam dat uložených lokálně v zařízení, přičemž každá položka obsahuje i miniaturu obrázku. Tento seznam umožňuje přechod na editační obrazovku, kde lze pořídit fotografii s aktuální polohou získanou pomocí geolokace. Stejně jako u předchozího seznamu je možné jednotlivé položky zobrazit v detailu. Třetí navigační možnost vede k obrazovce s rozšířenou realitou, která umožňuje vykreslení vybraného objektu na reálné ploše snímané kamerou zařízení.



Obrázek 7.1 Domovská stránka aplikace (iOS, Android, Flutter)

Aplikace začíná úvodní obrazovkou se třemi tlačítky, z nichž každé vede k jiné funkcionalitě a testovacímu scénáři. Navigace mezi obrazovkami je zajištěna především pomocí horního nástrojového panelu (toolbar), který uživateli umožňuje návrat zpět na domovskou stránku. Na obrazovkách, kde toolbar není přítomen, lze k návratu využít standardní funkci zpět dostupnou na mobilním zařízení. Domovskou obrazovku lze vidět na obrázku 7.1.

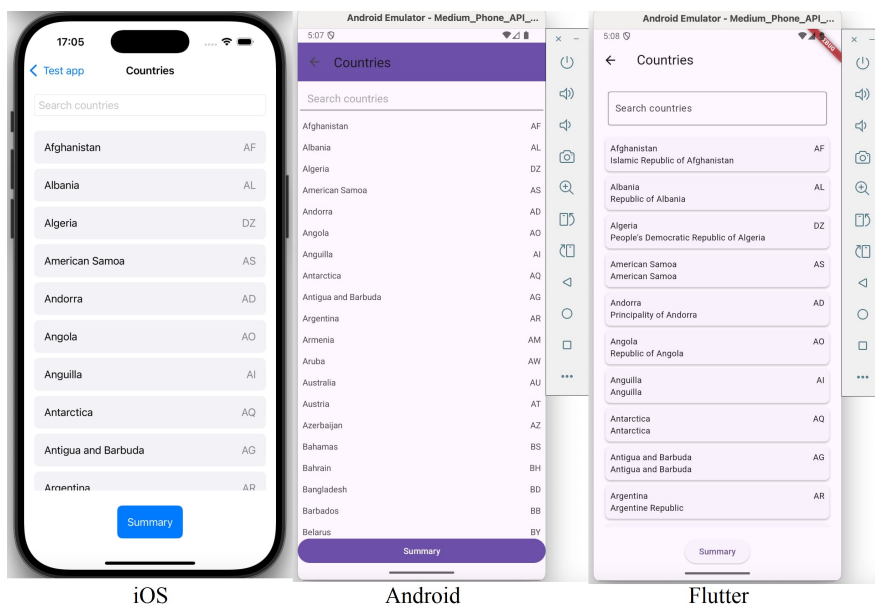
První tlačítko vede na obrazovku, na které jsou zobrazena data načítaná z externího REST API s názvem Countries [41]. Toto API poskytuje informace o jednotlivých státech světa, při-



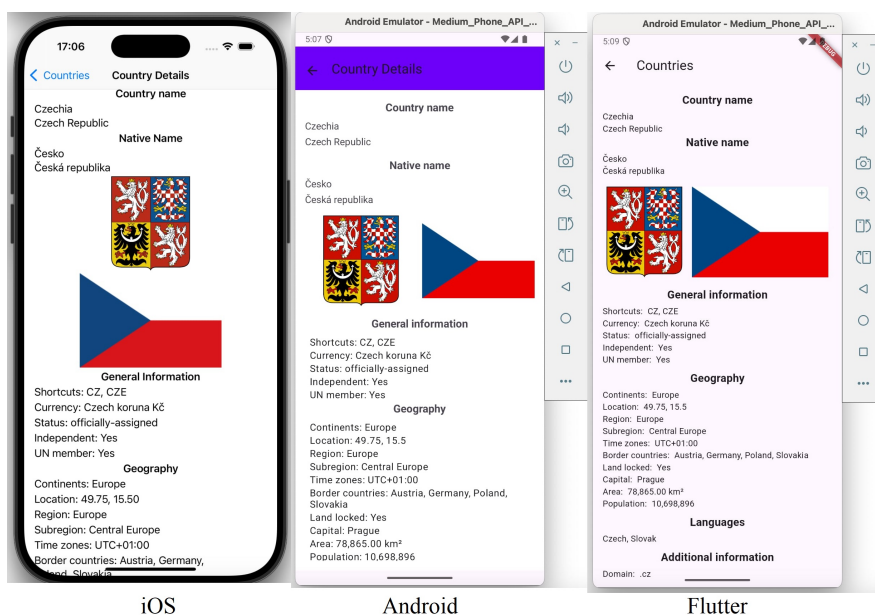
Obrázek 7.2 Chybová hláška aplikace (iOS, Android, Flutter)

čemž všechna dostupná data jsou načtena naráz při vstupu na stránku. Při vstupu na stránku, která načítá data z externího API, se na stránce objeví ukazatel průběhu načítání. Každý stát je zobrazen jako samostatná karta v přehledném seznamu. Na jednotlivé karty je možné kliknout, přičemž tento klik vede na stránku s detailním přehledem o státu. Uživatel má také k dispozici fulltextové vyhledávací pole, které umožňuje filtrovat státy podle celého názvu nebo i jejich zkratk. Stránka také obsahuje navigaci v podobě toolbaru, který vede zpátky na domovskou obrazovku. Kromě toho je na stránce k dispozici tlačítko, které vede na sumarizační obrazovku. Technické pozadí načítání dat a vysvětlení použitých pojmů, jako je REST či API, je podrobněji rozebráno v následující podkapitole 7.4. Tento seznam a i chybový stav seznamu je možné vidět na obrázcích 7.2 a 7.3.

Kliknutí na kartu konkrétního státu vede na novou obrazovku s detailním přehledem informací o dané zemi. Tato stránka znovu načítá data z externího API, tentokrát však pouze pro jeden konkrétní stát a případně i jeho sousední státy. Zobrazované informace jsou přehledně rozděleny do několika kategorií, mezi které patří globální název státu, název v rodném jazyce, obecné a geografické údaje, používané jazyky a dodatečné informace jako například údaj o tom, na které straně silnice se ve státě jezdí, nebo jakou má doménu nejvyšší úrovně. Pokud má stát k dispozici vlajku a erb, jsou tyto grafické prvky zobrazeny přímo na stránce. Obrázky se načítají přes internet pomocí odkazu na externí zdroj, nikoliv ze souborů uložených přímo v aplikaci. Tento detail je možné vidět na obrázku 7.4.



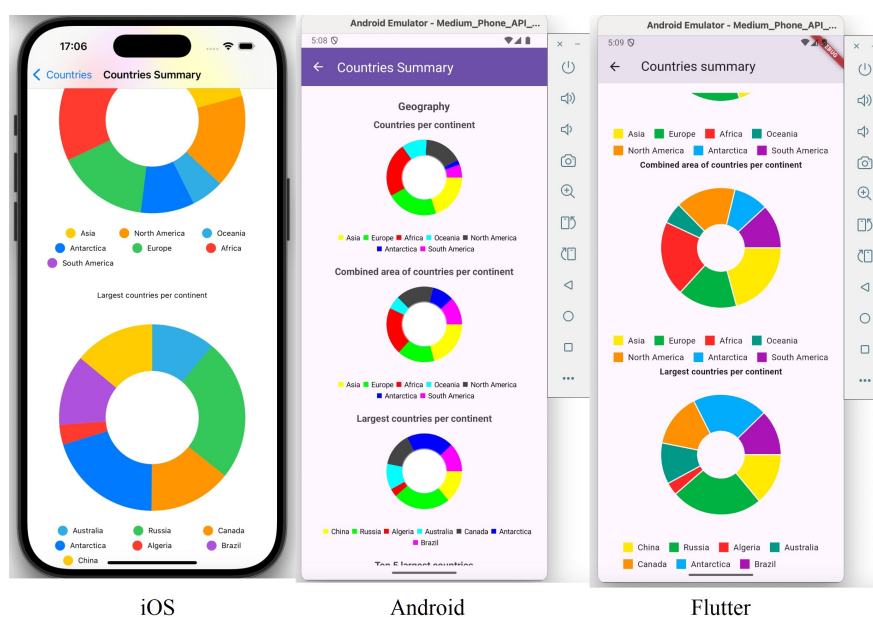
Obrázek 7.3 Seznam států v aplikaci (iOS, Android, Flutter)



Obrázek 7.4 Detail stránky státu (iOS, Android, Flutter)

Ze stránky se seznamem států je možné přejít na sumarizační obrazovku, která nabízí přehled agregovaných dat napříč všemi státy. Tato stránka si znovu načítá informace z externího API, tentokrát s cílem vizuálně znázornit různé globální statistiky. Získaná data jsou zpracována a zobrazena pomocí koláčových grafů, které umožňují rychlou orientaci v porovnávaných údajích. Prezentované statistiky jsou rozděleny do několika tematických kategorií, jako jsou obecné informace, geografické údaje, demografická data a jazyková rozmanitost. Mezi konkrétními vizualizacemi lze nalézt například graf s pětici největších států podle roz-

lohy, přehled počtu států rozdělených podle kontinentů, zastoupení nejčastěji používaných jazyků nebo informaci o tom, kolik států je členem Organizace spojených národů (OSN). Grafy slouží jako doplňkový přehled pro porovnání vybraných vlastností napříč zeměmi a zároveň zjednodušují interpretaci rozsáhlých dat, která by jinak byla obtížněji srozumitelná v čistě textové podobě. Toto je možné vidět na obrázku 7.5.

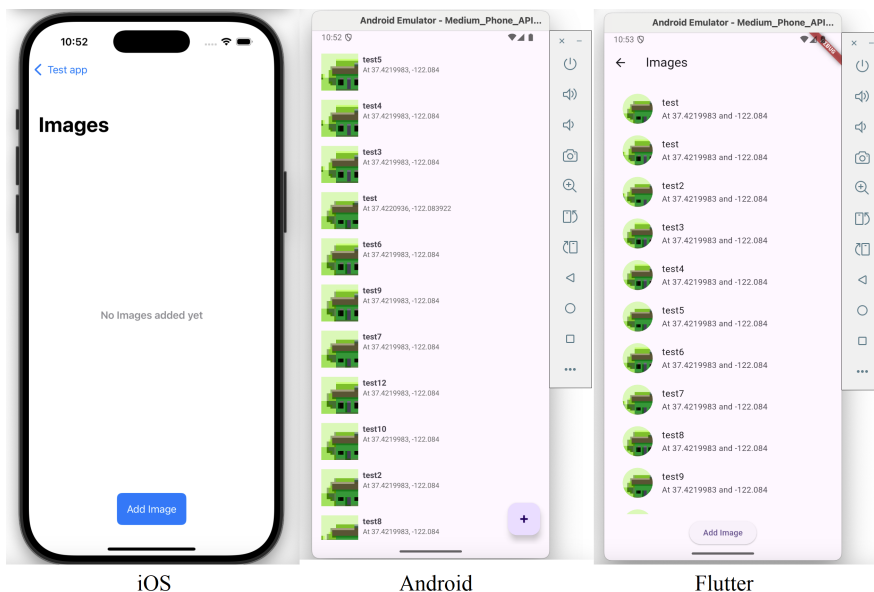


Obrázek 7.5 Sumarizační obrazovka (iOS, Android, Flutter)

Druhá sekce, do které se lze navigovat z domovské obrazovky, pracuje s obrázkovými daty uloženými lokálně v zařízení. Hlavní stránka této části aplikace zobrazuje seznam existujících záznamů, přičemž každá položka obsahuje miniaturu fotografie, její název a GPS souřadnice. Tento seznam je možné vidět na obrázku 7.6. Seznam umožňuje procházení již pořízených záznamů a po kliknutí na libovolnou položku je uživatel přesměrován na detailní obrazovku s rozšířenými informacemi. Na rozdíl od předchozích částí aplikace jsou data v tomto případě načítána výhradně z lokálního úložiště zařízení, bez potřeby připojení k internetu. Stránka zároveň obsahuje tlačítko, které uživateli umožňuje vytvořit nový záznam. Po jeho stisknutí je přesměrován na obrazovku, kde může pořídit novou fotografii a doplnit k ní další informace.

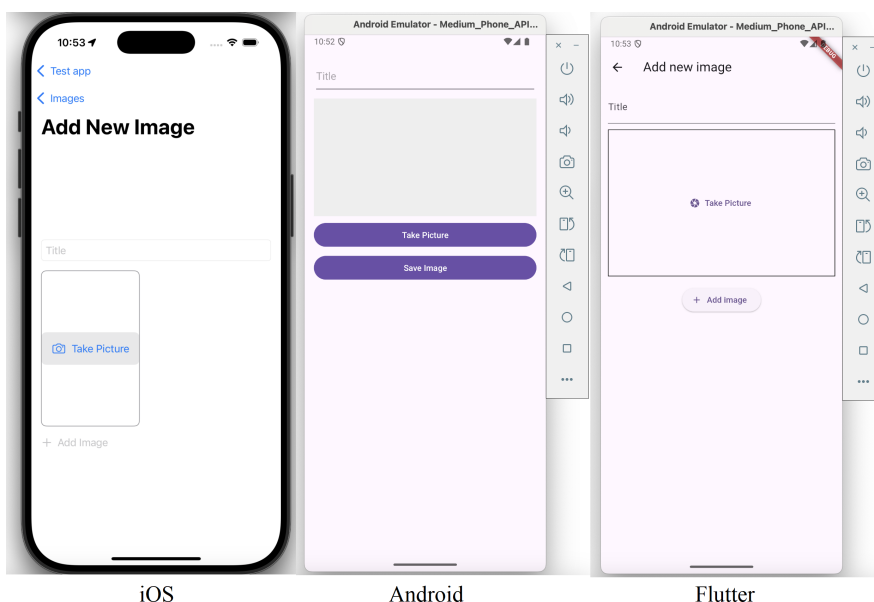
Detailová stránka je velmi jednoduchá, jelikož zobrazuje pouze samotnou fotografii. Na rozdíl od seznamu je zde fotografie zobrazována v plné velikosti, nikoliv jako miniatura. Obrázek se při přechodu nenačítá znovu, ale je předán přímo ze seznamu.

Ze seznamu je možné přejít na stránku pro přidání nové fotografie. Tato obrazovka ob-



Obrázek 7.6 Seznam obrázků v aplikaci (iOS, Android, Flutter)

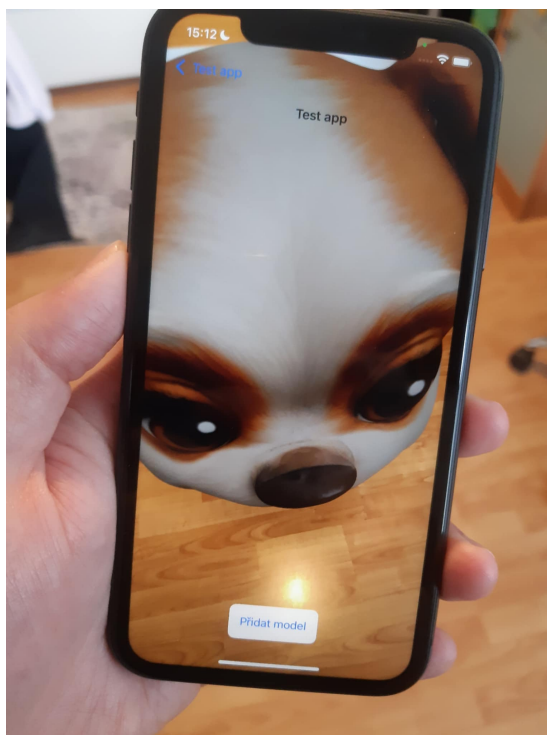
sahuje vstupní pole pro zadání názvu fotografie a pole, které po kliknutí umožňuje pořízení snímku prostřednictvím fotoaparátu zařízení. Zadaná data lze následně uložit pomocí tlačítka umístěného ve spodní části obrazovky. Po úspěšném uložení záznamu je uživatel automaticky přesměrován zpět na předchozí stránku. Přidávání obrázků lze vidět na obrázku 7.7.



Obrázek 7.7 Vytváření obrázků v aplikaci (iOS, Android, Flutter)

Třetí a zároveň poslední sekci, na kterou se lze z domovské obrazovky dostat, je rozšířená

realita neboli AR (Augmented Reality). Po přechodu na tuto obrazovku si uživatel může vybrat jeden z dostupných 3D modelů, který chce zobrazit ve scéně rozšířené reality. Po výběru modelu se aktivuje fotoaparát zařízení. Jakmile aplikace rozpozná rovnou plochu v reálném prostředí, zobrazí se na ní zmenšený 3D model vybraného objektu. Tuto funkcionalitu je možné vidět na obrázku 7.8.



Obrázek 7.8 Použití rozšířené reality v aplikaci

Na závěr byl záměrně zachován co nejjednodušší vzhled uživatelského rozhraní, bez použití grafických úprav či vizuálních efektů. Ve všech verzích aplikace byly využity výchozí prvky uživatelského rozhraní, které nabízí daná vývojová platforma. Tento přístup umožňuje jednak udržet rozhraní přehledné a nenáročné na implementaci, ale především slouží k pozorování přirozených rozdílů v chování a vzhledu nativních komponent napříč platformami.

7.3 Použité technologie a architektura

Všechny tři verze aplikace byly navrženy tak, aby uživatelům poskytovaly totožnou funkcionalitu, přestože jejich samotná implementace se v některých případech liší v závislosti na specifikách dané vývojové platformy. Pro nativní vývoj byly využity oficiální technologie jednotlivých systémů, zatímco hybridní řešení bylo realizováno s využitím moderního frameworku zaměřeného na vývoj z jednoho sdíleného kódu.

7.3.1 Zvolené technologie

Pro vývoj nativní aplikace pro operační systém Android byl použit programovací jazyk Kotlin. Jedná se o moderní, staticky typovaný jazyk podporovaný společností Google, který je plně kompatibilní s Javou, ale nabízí čistší a stručnější zápis. Vývoj probíhal v prostředí Android Studio, které poskytuje širokou podporu pro vývoj, ladění i testování aplikací na platformě Android.

Nativní verze aplikace pro iOS byla vytvořena v jazyce Swift za použití frameworku SwiftUI. Swift je výkonný jazyk vyvíjený společností Apple, navržený pro moderní vývoj aplikací s důrazem na bezpečnost a čitelnost kódu. SwiftUI pak představuje deklarativní nástroj pro tvorbu uživatelského rozhraní, který umožňuje efektivní návrh a údržbu vzhledu aplikace. Vývoj probíhal v prostředí Xcode, oficiálním vývojovým prostředím pro platformy Apple.

Pro vývoj hybridní verze aplikace byl zvolen framework Flutter, který umožňuje vytvářet aplikace pro Android i iOS z jednoho sdíleného kódu. Flutter využívá jazyk Dart a umožňuje sestavit celé uživatelské rozhraní pomocí widgetů. Flutter také umožňuje vytváření webových a softwarových aplikací. Byl zvolen především kvůli své rostoucí popularitě a schopnosti nativní kompilace, která z něj činí jeden z nejrychlejších frameworků. Díky těmto vlastnostem představuje vhodného kandidáta pro porovnání s nativními přístupy, zejména z hlediska výkonu a rychlosti běhu aplikace.

7.3.2 Založení projektu

Pro vytvoření nativní Android aplikace s názvem **testApp** bylo použito vývojové prostředí Android Studio. Projekt byl založen pomocí průvodce *New Project*, kde byla vybrána šablona *Empty Activity* a jako programovací jazyk zvolen Kotlin.

Projekt **testApp** pro iOS byl vytvořen v prostředí Xcode výběrem šablony App v sekci iOS. Při zakládání byl jako jazyk zvolen Swift a pro tvorbu rozhraní byl použit framework SwiftUI.

Hybridní projekt **testApp** byl založen pomocí příkazu `flutter create testApp` v terminálu. Poté byl otevřen v editoru Visual Studio Code, kde probíhala další konfigurace a vývoj jednotlivých obrazovek.

7.3.3 Nastavení aplikací

Při práci se soubory, fotoaparátem nebo dalšími prvky zařízení je nezbytné provést v každém projektu odpovídající konfigurační úpravy. V případě práce se soubory může konfigurace určovat například umístění, kde budou soubory ukládány. Pokud aplikace využívá prvky zařízení, jako je fotoaparát nebo lokalizační služby, je nutné uživateli zobrazit systémové upozornění s žádostí o udělení příslušného oprávnění k přístupu.

V souboru **AndroidManifest.xml** byla pro Android aplikaci deklarována oprávnění k přístupu na internet, k poloze, k fotoaparátu a k externímu úložišti. Soubor **AndroidManifest.xml** slouží k definování základních informací o aplikaci a k deklaraci oprávnění, která aplikace vyžaduje pro svůj běh. Pro práci s kamerou a ukládáním fotografií byl přidán **FileProvider**, který bezpečně zpřístupňuje soubory uložené v interním úložišti aplikace jiným aplikacím prostřednictvím sdílených URI adres. Dále bylo třeba deklarovat **ARScreenActivity** a **uses-feature** pro AR kameru a fotoaparát, aby bylo možné využívat rozšířenou realitu a funkce zařízení.

V souboru **Info.plist** byly pro iOS aplikaci doplněny záznamy **NSCameraUsageDescription** a různé varianty **NSLocationUsageDescription**, které definují oprávnění k přístupu ke kameře a poloze. Tyto popisy informují uživatele o důvodu žádosti o oprávnění. Soubor **Info.plist** slouží k zadávání konfiguračních informací a oprávnění, které aplikace požaduje při svém běhu na platformě iOS.

Ve Flutter verzi byly externí knihovny deklarovány v souboru **pubspec.yaml**, a byly to např. **http**, **fl_chart**, **image_picker**, **location**, **ar_flutter_plugin_updated** a další. Pro zpřístupnění funkcí nativních platforem bylo nutné upravit jak **AndroidManifest.xml**, tak **Info.plist**, které Flutter také obsahuje, jelikož obsahuje nativní kompilaci. Každý plugin má vlastní požadavky na oprávnění, které bylo třeba respektovat podle dokumentace daného balíčku.

7.4 Načítání a zpracování dat

Jednou z klíčových funkcionalit testovací aplikace je načítání a zpracování dat z externího zdroje. Pro tento účel bylo využito veřejně dostupné rozhraní REST API Countries. Data států z tohoto API jsou následně zobrazena uživateli ve formě seznamu, detailních obrázků a sumarizačních grafů. Zkratka API (Application Programming Interface) označuje

rozhraní, které umožňuje komunikaci mezi dvěma softwarovými systémy. V tomto případě aplikace odesílá požadavek na vzdálený server a na základě odpovědi získává data o jednotlivých státech. REST (Representational State Transfer) je architektonický styl pro návrh API, který využívá standardní HTTP metody (např. GET, POST) pro přenos dat. REST API je v současnosti nejběžnějším způsobem, jak aplikace získávají data ze vzdálených serverů.

Pro účely načítání a práce s daty byl ve všech verzích aplikace vytvořen model **Country**, který obsahuje vlastnosti odpovídající struktuře dat získaných z REST API Countries. Model byl navržen tak, aby pokrýval všechna potřebná pole vrácená v JSON odpovědi, včetně názvu státu, zkratky, jazyků, počtu obyvatel a dalších. Součástí zpracování dat je tedy také serializace (převedení JSON dat na objekty) a následné vytvoření seznamu instancí třídy **Country**, se kterými aplikace dále pracuje.

Na stránce se seznamem států je implementováno fulltextové vyhledávání, které umožňuje filtrovat seznam podle zadaného textu. Vyhledávání probíhá přímo nad již načtenými daty, bez dalších dotazů na API, a umožňuje hledat jak podle plného názvu státu, tak podle jeho zkratky. Pro zajištění lepší odezvy aplikace a snížení počtu zbytečných výpočtů při psaní bylo vyhledávací pole doplněno o debounce interval 500 ms. To znamená, že filtrování seznamu se spustí až po krátké prodlevě, kdy uživatel přestane psát, díky tomu je vyhledávání plynulejší a efektivnější.

Při přechodu na detailní stránku je dané obrazovce předána zkratka vybraného státu. Na jejím základě je následně odeslán upravený API požadavek, který načítá pouze data pro konkrétní stát, nikoliv celý seznam. Tento přístup zajišťuje úsporu přenesených dat a zároveň poskytuje aktuální informace. Součástí detailu je také zobrazení státní vlajky, která je načítána dynamicky z externího zdroje pomocí odkazu. Obrázek se nenačítá okamžitě při požadavku na data, ale až po vykreslení stránky, což napomáhá plynulejšímu zobrazení obsahu a omezuje prodlevy při přechodu mezi obrazovkami.

Součástí aplikace je rovněž sumarizační stránka, která zpracovává agregované údaje napříč všemi státy. Pro výpočty těchto dat jsou využity agregační funkce daného programovacího jazyka (např. seskupování, sčítání apod.), díky nimž je možné efektivně připravit vstupy pro vizualizaci ve formě grafů.

Na všech třech stránkách, které načítají data z REST API (seznam států, detail státu a sumarizační stránka), je implementována vizuální indikace průběhu načítání. Při odeslání API

požadavku se na obrazovce zobrazí točící se kolečko, které uživateli signalizuje, že probíhá asynchronní operace. Aplikace tak dynamicky mění svůj stav a poskytuje zpětnou vazbu o tom, že data ještě nejsou k dispozici. V případě, že během načítání dojde k chybě například z důvodu nedostupného připojení nebo chyby na straně serveru, je uživatel informován jednoduchou textovou zprávou zobrazenou přímo na obrazovce. Díky tomu je zajištěno, že aplikace v takovém případě neselže, nevygeneruje výjimku a nedojde k jejímu pádu. Tento přístup zvyšuje stabilitu aplikace a zlepšuje uživatelský komfort při práci s daty.

7.4.1 Nativní Android realizace

V nativní Android aplikaci postavené na jazyce Kotlin je načítání dat implementováno asynchronně pomocí knihovny `kotlinx.coroutines`. Data jsou získávána z REST API prostřednictvím třídy **HttpsURLConnection** v metodě `loadData` ve třídě **CountriesActivity**. Po načtení odpovědi ve formátu JSON probíhá serializace na seznam objektů třídy **Country**, který byl vytvořen tak, aby odpovídal struktuře dat poskytnuté API. Serializace probíhá ručně pomocí `org.json.JSONArray` a `getJSONObject`, bez využití externích knihoven.

Aplikace využívá vlastní třídu **Debouncer**, která pomocí **Handler** a **Runnable** implementuje zpoždění vstupu uživatele ve vyhledávacím poli. Vyhledávání je tak optimalizováno pomocí 500ms debounce intervalu a probíhá nad již načtenými daty bez dalšího API volání. Výsledný filtrovaný seznam se zobrazuje pomocí komponenty **RecyclerView** a vlastního adaptéru **CountryAdapter**. Tento adaptér využívá tzv. **ViewHolder** vzor, který efektivně pracuje s opakovaně používanými prvky rozhraní. Každá položka v seznamu zobrazuje běžný název státu a jeho zkratku, přičemž při kliknutí na konkrétní položku je vyvolána událost **onItemClick**, která přesměruje uživatele na detailní stránku daného státu.

Součástí implementace je i obsluha stavů aplikace jako třeba zobrazení načítacího indikátoru (**ProgressBar**) při běhu asynchronního požadavku a zobrazení chybové zprávy pomocí **Toast** při neúspěšném spojení se serverem nebo výjimce během zpracování. Tento přístup zajišťuje robustní chování a příjemný uživatelský zážitek.

Detailní zobrazení informací o konkrétním státě je implementováno ve třídě **CountryDetailActivity**. Při přechodu z předchozí obrazovky je této aktivitě předána zkratka státu prostřednictvím **Intent** objektu. Na jejím základě je sestaven URL dotaz směřující na konkrétní endpoint REST API, který vrací detailní JSON odpověď pouze pro daný stát. Na rozdíl

od seznamové stránky tedy dochází k načtení jen jednoho záznamu, čímž se minimalizuje množství přenášených dat.

Získaná data jsou zpracována ručně pomocí **JSONObject** a přiřazena do připravených **TextView** komponent v layoutu **activity_country_detail.xml**. Obrázek vlajky je načítán asynchronně z URL adresy pomocí knihovny **Glide**, která se stará o efektivní správu stahování, kešování a zobrazování obrázků. Načítání obrázku probíhá až po vykreslení stránky, což zajišťuje plynulejší zobrazení a zamezuje prodávám v UI.

I v této aktivitě je ošetřen stav načítání pomocí **ProgressBar**, který se zobrazí při inicializaci požadavku a následně skryje po úspěšném zpracování dat. V případě chyby je uživatel informován jednoduchou zprávou pomocí komponenty **Toast**.

Sumarizační obrazovka aplikace je implementována ve třídě **CountriesSummaryActivity**. Po spuštění aktivity je odeslán požadavek na stejné REST API jako v případě seznamové stránky, přičemž je opět načten celý seznam států ve formátu JSON. Získaná data jsou následně převedena na seznam objektů třídy **Country** a zpracována pomocí funkcí jazyka Kotlin pro agregaci a filtrování.

Výpočty zahrnují například identifikaci pěti největších států podle rozlohy, počet států na jednotlivých kontinentech, jazykovou rozmanitost nebo podíl členských států Organizace spojených národů (UN). Tyto statistiky jsou zobrazovány vizuálně pomocí knihovny **MPAndroidChart**, která umožňuje snadné vykreslení koláčových grafů přímo v aplikaci. Grafy jsou generovány pomocí komponenty **PieChart** a jejich datové sady (**PieDataSet**) jsou konfigurovány v závislosti na výsledku agregace. Agregovaná data plní seznam datového typu **ChartItem**. Jedná se o vlastní model, který se transformuje na model typu **PieDataSet**.

I v této aktivitě je použit **ProgressBar** pro vizuální indikaci probíhajícího načítání a výpočtů. Případné chyby v průběhu operace jsou zachyceny pomocí try-catch bloku a uživatel je informován zobrazením jednoduché chybové zprávy prostřednictvím **Toast**.

7.4.2 Nativní iOS realizace

Základní obrazovka se seznamem států je implementována jako komponenta **CountriesScreen** ve SwiftUI. Data jsou načítána z REST API pomocí **URLSession** a zpracována ve funkci **parseCountriesForList**, která převádí JSON odpověď na seznam objektů typu **Country**. Tento model reprezentuje jednotlivé státy a je navržen tak, aby odpovídal

datové struktury poskytované API. Po načtení jsou data seřazena podle názvu a zobrazena v komponentě **LazyVStack** s využitím **ForEach**.

Vyhledávání ve státech je realizováno pomocí textového pole s **onChange** listenerem, který dynamicky filtruje záznamy podle názvu, oficiálního názvu nebo zkratky. Filtrování probíhá přímo nad lokálně uloženými daty a výsledek se okamžitě promítá do UI. Součástí obrazovky je také indikátor stavu – komponenta **ProgressView** během načítání a jednoduchá textová chybová zpráva při selhání.

Detailní obrazovka pro jednotlivý stát je definována v **CountryDetailScreen**. Při otevření stránky je na základě předané hodnoty zkratky státu sestaven požadavek na konkrétní API endpoint, který vrací detailní data pouze pro vybraný stát. Získaná data jsou zpracována pomocí asynchronní funkce `parseCountries`, která využívá **async/await** pro zajištění neblokujícího chodu aplikace.

Zobrazované informace zahrnují vše od základních názvů až po geografická, politická a kulturní data. Obrázky vlajky a erbu jsou načítány asynchronně pomocí komponenty **AsyncImage**, která zahrnuje vlastní **ProgressView** jako placeholder. Výsledný design je uspořádán do sekcí s nadpisy a využívá jednoduché SwiftUI komponenty bez složitých rozvržení. V případě chyby během načítání se zobrazí chybová zpráva a obsah není vykreslen.

Sumarizační přehled o všech státech je realizován v **CountriesSummaryScreen**. Po načtení kompletního seznamu států z REST API dochází k agregaci dat pomocí funkcí jako **map**, **reduce**, **sorted** či vlastních metod pro práci s datovými strukturami. Data jsou seskupována například podle kontinentu, rozlohy, počtu obyvatel, členství v OSN, nebo počtu používaných jazyků.

Vizualizace výsledků je realizována pomocí komponenty **ChartView**, která využívá framework **Swift Charts**. Každý graf je reprezentován strukturou **ChartItem** a zobrazuje se jako koláčový graf (**SectorMark**) s legendou. Výběrem segmentu lze zobrazit hodnotu přímo v grafu. Celá stránka je zabalena do **ScrollView**, což umožňuje pohodlné procházení jednotlivých sekcí. Načítací stav je indikován pomocí **ProgressView** a při absenci dat je uživatel informován textovou zprávou.

7.4.3 Flutter realizace

V aplikaci vytvořené pomocí Flutteru je seznam států implementován ve třídě **CountriesScreen**, která využívá **StatefulWidget** a externí knihovnu **http** pro načítání dat z REST API. JSON odpověď je manuálně parsována na instanci modelu **Country**, jehož struktura přesně odpovídá formátu API. Pro dynamické filtrování dat při psaní do textového pole je použita vlastní implementace třídy **Debouncer**, která zajišťuje zpoždění (debounce) o 500 ms před aktualizací výsledků – čímž se optimalizuje výkon při vyhledávání.

Zobrazování seznamu je realizováno pomocí komponenty **ListView.builder** s využitím widgetů **Card**, **Text** a **InkWell**. Při kliknutí na libovolnou položku je uživatel přesměrován na obrazovku s detailem státu (**CountryDetailScreen**). Vizualní zpětná vazba během načítání je zajištěna komponentou **CircularProgressIndicator**, případné chyby se zobrazují prostřednictvím jednoduchého textového výstupu.

Obrazovka **CountryDetailScreen** využívá komponentu **StatefulWidget** a přebírá jako parametr název státu. Na základě něj je vytvořen API požadavek, který načítá detailní informace o daném státě. Model **Country** zde obsahuje rozsáhlejší množinu vlastností, včetně geografických údajů, jazyka, časových pásem či informací o měně. Čísla jsou formátována pomocí externí knihovny **intl**.

Obrázky vlajek a erbů jsou zobrazovány prostřednictvím **FadeInImage.assetNetwork**, což umožňuje elegantní zobrazení s placeholderem, a obrázky se načítají až po vykreslení stránky. Detailní informace jsou strukturovaně rozděleny do tematických bloků a zobrazeny pomocí widgetů **Row**, **Text**, **Wrap** a dalších nativních prvků Flutteru.

Sumarizační obrazovka **CountriesSummaryScreen** je zodpovědná za agregaci dat a jejich následné vizuální zobrazení. Po načtení všech států z API probíhá lokální zpracování dat v metodě `_loadSummary`, kde jsou pomocí běžných funkcí jazyka Dart (např. **map**, **fold**, **sort**, **where**) vytvořeny datové mapy pro vizualizaci například počet obyvatel na kontinent, podíl členů OSN, největší státy podle rozlohy apod.

Vizualizace je řešena pomocí externí knihovny **fl_chart**, konkrétně komponentou **PieChart** ze souboru **chart.dart**. Všechny segmenty grafu jsou barevně odlišeny a doplněny legendou. Interaktivní prvek **pieTouchData** umožňuje zvýraznění vybraného segmentu s vypsáním hodnoty. Stránka je scrollovatelná (**SingleChildScrollView**) a obsahuje i základní

zpětnou vazbu v podobě **CircularProgressIndicator** a textové chybové hlášky.

7.5 Pořizování fotografií a lokální úložiště

Součástí aplikace je modul pro práci s lokálně uloženými daty, který umožňuje zobrazit, spravovat a rozšiřovat seznam uživatelských záznamů. Každý záznam obsahuje název, fotografii pořízenou pomocí zařízení a souřadnice GPS. Data jsou uložena v lokální databázi přímo na zařízení, takže nejsou závislá na síťovém připojení a jsou dostupná i v offline režimu.

Záznamy se zobrazují v seznamu, který má podobu tzv. kartového rozvržení. Každá karta prezentuje náhledovou miniaturu fotografie, textový název a polohu. Seznam je dynamicky načítán z databáze při každém otevření obrazovky, čímž je zajištěno, že vždy reflektuje aktuální stav uložených dat. Kliknutím na libovolný záznam dojde k přesměrování na detailní obrazovku, která zobrazí fotografii v její plné velikosti.

Základní funkcionality pro rozšíření seznamu je dostupná prostřednictvím tlačítka pro přidání nového záznamu. To vede na samostatnou obrazovku s formulářem, který obsahuje vstupní pole pro zadání názvu fotografie a ovládací prvek pro přidání samotného obrázku. Tento prvek po interakci aktivuje fotoaparát zařízení, který umožní uživateli pořídit snímek v reálném čase.

Před uložením záznamu do databáze probíhá validace všech povinných polí – tedy názvu, fotografie i dostupnost GPS polohy. Po úspěšné validaci je zjištěna aktuální pozice zařízení (např. prostřednictvím GPS senzoru) a všechny hodnoty jsou zapsány do lokální databáze. Následně je uživatel automaticky přesměrován zpět na předchozí obrazovku se seznamem, který se obnoví tak, aby zobrazoval nově přidány záznam.

Tento přístup umožňuje uchovávání informací přímo na zařízení bez nutnosti vzdáleného serveru, což je výhodné například při práci v prostředích s omezeným připojením k internetu. Použití fotoaparátu a GPS zároveň poskytuje aplikaci praktické možnosti pro rozšířené evidenční nebo záznamové funkce, které lze využít v celé řadě případů – od dokumentace, po práci v terénu.

Všechny verze aplikace využívají centrální komponentu pro správu lokální databáze, označovanou jako provider. Tento provider zpřístupňuje jednotné rozhraní pro základní operace nad databází jako jsou například založení a inicializaci databáze, načítání uložených záznamů a přidávání nových obrázků s metadaty. Výhodou tohoto přístupu je centralizace

logiky na jedno místo, čímž se snižuje duplikace kódu, zlepšuje čitelnost a usnadňuje údržba aplikace. Provider je zpřístupněn globálně napříč aplikací, takže jej mohou snadno využívat jednotlivé obrazovky a komponenty bez přímé vazby na konkrétní databázovou implementaci.

7.5.1 Nativní Android realizace

Aplikace využívá databázi **Room**, která je definována ve třídě **AppDatabase** jako singleton pomocí vzoru `getDatabase(context)`. **Singleton** je návrhový vzor, který zajišťuje, že v aplikaci existuje pouze jedna instance dané třídy a poskytuje k ní globální přístup. Databáze obsahuje jednu entitu **ImageItemEntity**, která reprezentuje záznam s unikátním ID, názvem, cestou k obrázku, GPS souřadnicemi a adresou. Přístup k datům je zajištěn přes DAO rozhraní **ImageItemDao**, které poskytuje metody pro vložení nového záznamu a načtení všech obrázků. Tyto operace jsou asynchronní a označené jako **suspend**.

Hlavní seznam obrázků je implementován ve třídě **ImagesActivity**. Záznamy jsou zobrazeny v komponentě **RecyclerView** pomocí vlastního adaptéru **ImageListAdapter**, který využívá `layout list_item_image.xml`. Každá položka seznamu zobrazuje název obrázku, GPS lokaci a miniaturu. Kliknutím na záznam se otevře **ImageDetailActivity**, která zobrazí obrázek v plné velikosti.

Data jsou poskytována pomocí architektury ViewModel (**ImageViewModel**), který komunikuje s **ImageRepository** a načítá data asynchronně pomocí coroutines (`viewModelScope.launch`). K pozorování změn slouží **LiveData**, která automaticky aktualizuje seznam při každé změně v databázi. Instanci ViewModelu zajišťuje tovární třída **ImageViewModelFactory**.

Obrazovka pro přidání nového záznamu je realizována ve třídě **AddImageActivity**. Po kliknutí na tlačítko se spustí fotoaparát zařízení pomocí `Intent(MediaStore.ACTION_IMAGE_CAPTURE)`. Výsledná fotografie je uložena do dočasného souboru pomocí **FileProvider**, přičemž cesta k souboru je zachována pro pozdější zpracování. Následně je získána aktuální GPS poloha pomocí služby **FusedLocationProviderClient**.

Po vyplnění názvu a úspěšném pořízení fotografie a získání lokace je záznam vytvořen jako instance **ImageItemEntity** a uložen do databáze přes **ImageViewModel**. Následně je

uživatel přesměrován zpět na seznam, který se automaticky obnoví.

Třída **ImageDetailActivity** slouží k zobrazení celé fotografie. Pomocí **Intent** se na tuto aktivitu předává cesta k obrázku a název, který je použit jako nadpis v horní liště. Samotný obrázek je zobrazen přes **ImageView** komponentu s metodou `setImageURI`.

7.5.2 Nativní iOS realizace

Data o obrázcích jsou v aplikaci reprezentována modelem **ImageItem**, který obsahuje název, cestu k obrázku, GPS souřadnice a adresu. Tyto informace jsou zapouzdřeny ve struktuře **ImageLocation**, která uchovává zeměpisnou šířku, délku a adresní popis lokace. Práce s daty je centralizována ve třídě **UserImagesProvider**, která implementuje správu záznamů pomocí vestavěné databáze **SQLite**. Provider je označen jako **ObservableObject** a zpřístupňuje data napříč aplikací pomocí **EnvironmentObject**.

Při inicializaci provideru dochází k otevření nebo vytvoření databázového souboru **images.sqlite** v aplikační složce **Documents**. Třída automaticky vytvoří databázovou tabulku **user_images**, pokud neexistuje. Metoda `fetchImages` načítá záznamy z databáze, provádí mapování na instanci **ImageItem** a aktualizuje pozorované pole **images**. Nové záznamy se ukládají metodou `addImage`, která vygeneruje unikátní název obrázku, uloží ho do souborového systému a následně vloží odpovídající metadata do **SQLite** databáze.

Obrazovka **ImagesScreen** využívá komponentu **ImagesList**, která zobrazuje seznam záznamů pomocí komponenty **List**. Každá položka zobrazuje miniaturu obrázku (načtenou přes **UIImage(contentsOfFile:)**), název a adresu. Pokud není obrázků k dispozici, zobrazí se alternativní text. Po kliknutí na položku se otevře **ImageDetailScreen**, která vykreslí obrázek v plné velikosti a nastaví jeho název jako titulek stránky.

Přidání nového obrázku probíhá v komponentě **ImageEditScreen**. Součástí obrazovky je vlastní komponenta **ImageInput**, která slouží k interakci s fotoaparátem. Po kliknutí na příslušné tlačítko je prostřednictvím komponenty **ImagePicker** spuštěn nativní fotoaparát zařízení. Komponenta **ImagePicker** využívá **UIKit** třídu **UIImagePickerController** a je implementována jako přemostění do **SwiftUI** pomocí protokolu **UIViewControllerRepresentable**. Po pořízení snímku je tento obrázek zobrazen v komponentě **ImageInput** jako miniatura.

Pořízený obrázek je před uložením převeden do formátu JPEG a uložen do dokumento-

vého adresáře aplikace **Documents**. K tomu slouží pomocná funkce `saveImageToDocuments`, která zároveň vygeneruje unikátní název souboru, čímž se zamezí přepsání již existujících dat. Souběžně s tím je pomocí třídy **LocationManager** získána aktuální poloha zařízení. Tato třída využívá systémový framework **CoreLocation** a slouží k získání zeměpisné šířky a délky ve formátu **CLLocationCoordinate2D**. Lokace je aktualizována automaticky a výsledné souřadnice jsou následně zahrnuty do ukládaného záznamu.

Při kliknutí na tlačítko Add Image dojde k validaci vstupních dat, vytvoření nové instance **ImageItem** a jejímu uložení do databáze pomocí provideru **UserImagesProvider**. Po úspěšném uložení je uživatel automaticky přesměrován zpět na předchozí obrazovku.

7.5.3 Flutter realizace

Data o pořízených fotografiích jsou v aplikaci reprezentována pomocí třídy **ImageItem**, která obsahuje název, soubor obrázku, unikátní ID a lokaci (**ImageLocation**) s GPS souřadnicemi a adresou. Správa dat je realizována pomocí **StateNotifier** s názvem **UserImagesNotifier**, který je poskytován globálně přes **StateNotifierProvider**. Veškerá data jsou uložena v lokální **SQLite** databázi, přístupné pomocí balíčku **sqflite**. Provider je realizován pomocí balíčku **riverpod**.

Při prvním spuštění je databáze **images.db** inicializována metodou `_getDatabase` a v ní je vytvořena tabulka **user_images**. Třída **UserImagesNotifier** poskytuje metody `loadImage` pro načítání a `addImage` pro přidávání nových záznamů. Každý obrázek je nejprve uložen do aplikační složky pomocí balíčku **path_provider**, a poté jsou metadata zapsána do databáze. Záznamy jsou dále ukládány do vnitřního stavu poskytovatele (**state**), což umožňuje jejich okamžité zobrazení v UI.

Hlavní seznam obrázků je zobrazen v komponentě **ImagesList**, která je vložena do **ImagesScreen**. Záznamy se načítají přes **FutureBuilder**, který zobrazuje **CircularProgressIndicator** při načítání a seznam záznamů po načtení. Každý záznam je reprezentován jako **ListTile** s miniaturou obrázku (**CircleAvatar**), názvem a adresou. Po kliknutí na záznam je uživatel přesměrován na **ImageDetailScreen**, kde se zobrazí obrázek v plné velikosti.

Přidání nového obrázku se provádí přes komponentu **ImageEditScreen**, která obsahuje vstupní pole pro název a widget **ImageInput** pro práci s fotoaparátem. Ten používá balíček **image_picker** pro aktivaci kamery a získání obrázku, který se následně zobrazí v náhledu. Po

vyplnění všech údajů a stisknutí tlačítka Add Image se zavolá metoda addImage v provideru a záznam se uloží.

Před uložením fotografie se získá aktuální poloha zařízení pomocí balíčku **location**. Komponenta **_getCurrentLocation** kontroluje oprávnění, aktivuje GPS a získává souřadnice, které se následně spolu s adresou uloží do objektu **ImageLocation**. Po úspěšném uložení je uživatel přesměrován zpět na seznam.

7.6 Rozšířená realita

V této aplikaci byla technologie rozšířené reality využita především za účelem ověření, jak dobře se s ní pracuje v prostředí hybridní aplikace a zda se projeví výraznější rozdíly ve spotřebě baterie mobilního zařízení.

Aplikace načítá 3D modely buď ze vzdáleného zdroje (například z internetu), nebo přímo ze zdrojového kódu. Tyto modely je možné zobrazit na obrazovce s rozšířenou realitou. Při přechodu na AR obrazovku se automaticky aktivuje fotoaparát zařízení a aplikace začne skenovat okolní prostor za účelem detekce rovné plochy. Jakmile je plocha rozpoznána, lze na ni po stisknutí umístit 3D model. V případě, že detekce selže, je uživateli k dispozici tlačítko pro manuální umístění modelu na scénu.

V části s rozšířenou realitou byl použit 3D model „Little Cartoon Dog“ od uživatele *na-diro*, dostupný na **Sketchfab**: <https://sketchfab.com/3d-models/little-cartoon-dog-e0c1413956dc40769ba3bf96dc476bc3>. Licencováno pod CC BY 4.0: <https://creativecommons.org/licenses/by/4.0/>. Tento model byl zvolen, protože je dostatečně jednoduchý pro zobrazení ve všech verzích aplikace a zároveň je dostupný pod bezplatnou licencí.

7.6.1 Nativní Android realizace

V nativní Android aplikaci je rozšířená realita implementována pomocí open-source knihovny **SceneView**, která staví na technologii **ARCore** a poskytuje vyšší úroveň abstrakce pro práci s AR funkcemi. Hlavním zobrazovacím prvkem je **ArSceneView**, který je součástí rozvržení aktivity a slouží jako výchozí plocha pro renderování 3D obsahu. Model ve formátu .glb je načítán asynchronně pomocí metody `loadModelGlbAsync()` třídy **ArModelNode**, která zároveň provede jeho ukotvení ve scéně prostřednictvím metody `anchor()`. Po

úspěšném načtení je model přidán do scény (`sceneView.addChild(node)`) a vizualizace roviny je deaktivována (`planeRenderer.isVisible = false`), čímž se dosáhne realistického zobrazení. Uživatel aktivuje zobrazení modelu ručně kliknutím na plovoucí tlačítko (**FloatingActionButton**).

Použitá knihovna **SceneView** interně využívá **ARCore** je oficiální rozhraní od společnosti Google pro zpracování rozšířené reality na Androidu. Aby bylo možné tyto funkce používat, musí zařízení podporovat **ARCore** a zároveň obsahovat odpovídající softwarové komponenty (např. služby Google Play Services for AR ve verzi kompatibilní s daným zařízením). Bez této podpory aplikace AR funkce nespustí.

Kromě toho je nutné deklarovat oprávnění pro přístup ke kameře v souboru **AndroidManifest.xml**, jelikož kamera slouží jako základní vstupní prvek pro prostorovou orientaci a detekci ploch. Před spuštěním scény aplikace kontroluje kompatibilitu zařízení a v případě nekompatibility AR režim nespustí.

7.6.2 Nativní iOS realizace

Funkcionalita rozšířené reality v nativní iOS aplikaci je realizována pomocí frameworku **RealityKit** v kombinaci s **ARKit**, které umožňují detekci rovinných povrchů a následné vykreslení 3D objektů ve skutečném prostoru skrze kameru zařízení. Funkčnost je rozdělena mezi dvě hlavní komponenty **ARScreen** a **ARViewContainer**.

Soubor **arscreen.swift** obsahuje hlavní SwiftUI komponentu **ARScreen**, která reprezentuje obrazovku s AR funkcionalitou. V rámci této obrazovky se nachází uživatelské rozhraní s tlačítkem pro přidání 3D objektu do prostředí. Pomocí proměnné `placeObject`, deklarované jako **@State**, se uchovává stav, zda má být objekt umístěn do scény. Při stisknutí tlačítka „Přidat model“ se hodnota `placeObject` přepne na `true`, čímž se spustí logika v komponentě **ARViewContainer**, která následně vykreslí 3D model do prostoru před uživatelem.

Komponenta **ARViewContainer** je implementována jako struktura odpovídající protokolu **UIViewRepresentable**, což umožňuje propojení klasického UIKit rozhraní se SwiftUI. Tato komponenta slouží jako kontejner pro samotnou AR scénu. V metodě `makeUIView` je vytvořen objekt **ARView**, do kterého je aplikována konfigurace **ARWorldTrackingConfiguration**, která aktivuje detekci horizontálních rovin. Výsledkem je, že zařízení dokáže rozpoznat rovnou plochu např. stůl nebo podlahu, na kterou může být následně umístěn 3D

objekt. Koordinátor v této komponentě je zodpovědný za logiku přidání modelu. Při změně stavu `placeObject` se metoda `updateUIView` spustí a zavolá metodu `addModel()`. V této metodě probíhá tzv. „raycasting“ neboli vyhledávání rovného povrchu ve středu obrazovky. Pokud je povrch detekován, vytvoří se na tomto místě **AnchorEntity**, ke které je připojen 3D model. Model je načítán ze souboru typu `.usdz`, který je uložen v balíčku aplikace. Tento model je načten pomocí metody `ModelEntity.load(contentsOf:)` a po úspěšném načtení je přidán do scény. Pokud by načtení selhalo, do konzole se vypíše chybová hláška.

7.6.3 Flutter realizace

K implementaci rozšířené reality ve Flutter aplikaci byl použit balíček **ar_flutter_plugin_updated**, který umožňuje zobrazovat 3D objekty v reálném prostoru prostřednictvím nativního **ARCore/ARKit** rozhraní. Dále je použit balíček **vector_math** pro práci s 3D pozicemi, rotacemi a měřítkem. Komponenty AR se zobrazují skrze widget **ARView**, který vytváří přímé propojení s nativní AR vrstvou systému.

Obrazovka **ArListPage** zobrazuje seznam dostupných 3D modelů, které uživatel může vybrat k zobrazení v AR režimu. Modely jsou reprezentovány pomocí třídy **ARObject**, která obsahuje název objektu, cestu k souboru (`uri`) a typ zdroje (**NodeType**). V seznamu se nachází jak lokální modely uložené v `assets` složce, tak vzdálené modely načítané z internetu (např. ve formátu `.glb` nebo `.gltf`). Seznam je vykreslen pomocí **GridView**, kde každá položka otevírá **ARViewPage** s konkrétním objektem.

Stránka **ARViewPage** slouží k samotnému zobrazení objektu v rozšířené realitě. Po inicializaci komponenty **ARView** je pomocí `onARViewCreated` vytvořeno připojení ke čtyřem hlavním správcům: **ARSessionManager**, **ARObjectManager**, **ARAnchorManager** a **ARLocationManager**. Při spuštění se aktivuje kamera zařízení, zapne detekce rovných ploch a inicializuje se scéna.

Uživatel může do scény vložit 3D objekt pomocí plovoucího tlačítka (**FloatingActionButton**), přičemž je rozlišováno, zda se jedná o lokální (**localGLTF2**) nebo webový (**webGLB**) model. Objekt je přidán pomocí `addNode()` z **ARObjectManageru**, kde se definuje jeho pozice, rotace a měřítko (např. `scale: Vector3(0.2, 0.2, 0.2)`). V případě problémů se načtením je uživateli zobrazena chybová hláška pomocí dialogového okna.

Třída **ARObject** obsahuje metodu `toARNode()`, která převádí logický model na instanci třídy **ARNode**, usnadňuje práci při přidávání do scény. Přepínání přidání/odebrání objektu funguje tak, že pokud je objekt již ve scéně, je odstraněn pomocí `removeNode()` a při dalším kliknutí znovu vložen. Komponenta podporuje i více typů objektů, což umožňuje flexibilní testování různých typů vstupních formátů.

7.7 Shrnutí

Zahájení vývoje nativní iOS aplikace s sebou přineslo určité komplikace, a to zejména kvůli uzavřenému ekosystému společnosti Apple. Pro vývoj je nezbytné mít zařízení s operačním systémem macOS, což omezuje dostupnost prostředí. Jakmile však byla tato překážka překonána, instalace vývojového prostředí Xcode i samotné založení projektu bylo velmi jednoduché a intuitivní. Před zahájením programování je však nutné mít vytvořený Apple vývojářský účet, což může celý proces mírně prodloužit. V případě nativního vývoje pro Android stačilo stáhnout Android Studio, které je dostupné pro všechny hlavní operační systémy. Průvodce instalací i založením projektu je uživatelsky přívětivý, stejně jako v případě Xcode. Při instalaci Xcode je automaticky přidán iOS simulátor, zatímco v Android Studiu je třeba emulátor nastavit ručně. Pro základní testování však stačilo vybrat verzi systému a typ zařízení. Instalace Flutter frameworku byla mírně zdlouhavější. Ačkoliv samotné stažení a rozbalení SDK probíhá rychle, Flutter vyžaduje i instalaci Android Studia a emulátoru, což v tomto případě nepředstavovalo problém, protože tyto komponenty již byly připraveny z předchozího vývoje Android aplikace. Vytvoření Flutter projektu je následně otázkou jednoho příkazu v terminálu a aplikaci lze pohodlně vyvíjet ve Visual Studio Code. Celkově lze říci, že příprava vývojového prostředí pro žádnou z platforem nebyla výrazně složitá. Nejrychlejší a nejméně zatěžující proces přípravy však nabídl Android Studio pro nativní Android vývoj.

Během vývoje bylo patrné, že aplikace vytvořená pomocí Flutteru vyžadovala největší množství externích knihoven, a to i pro základní funkce, jako je formátování čísel. Řada funkcionalit, které jsou v nativním vývoji dostupné bez nutnosti explicitního importu, bylo ve Flutteru nutné přidat formou balíčků. Instalace knihoven však byla jednoduchá, obvykle šlo pouze o spuštění jednoho příkazu v terminálu, případně ruční přidání názvu balíčku do souboru **pubspec.yaml**. Po synchronizaci byly knihovny okamžitě dostupné v rámci projektu. Také nativní Android aplikace vyžadovala import specifických knihoven, například

pro práci s databází **Room** nebo pro zobrazení **AR**. Tyto knihovny bylo nutné deklarovat v souboru **build.gradle**, a následně synchronizovat projekt. Tento proces je poměrně standardní, nicméně může být náchylný k chybám, v některých případech bylo nutné restartovat Android Studio, pokud synchronizace selhala. Pro správné fungování externích balíčků je také důležité kontrolovat a případně upravit verze nástrojů pro sestavování (build), což platí jak pro Flutter, tak pro nativní Android vývoj. V případě nativní iOS aplikace nebylo nutné do tohoto projektu importovat žádné externí knihovny, a rovněž nebylo třeba upravovat verze nástrojů pro sestavení. Vývoj v tomto ohledu probíhal nejplynuleji.

Aplikace v každé variantě využívá funkce zařízení, jako je fotoaparát nebo lokalizace, které vyžadují potvrzení přístupu od uživatele. Tato oprávnění bylo nutné nastavit v příslušných konfiguračních souborech, v případě Androidu v **AndroidManifest.xml**, v iOS v **Info.plist**. Flutter, jako hybridní řešení, vyžaduje definování těchto oprávnění v obou těchto souborech současně. Přestože se konfigurace liší podle platformy, z pohledu vývojáře nejde o složitý ani časově náročný krok.

Vývoj nativní iOS aplikace ve SwiftUI i hybridní aplikace ve Flutteru vyžaduje znalost jednotlivých widgetů a jejich deklarativního zápisu. Uživatelské rozhraní se zde tvoří přímo v kódu, přičemž každý prvek je reprezentován specifickým widgetem s vlastním chováním a vzhledem. V případě nativního Android vývoje je možné využít vizuální návrhář v Android Studiu, který umožňuje komponenty (tzv. Views) přetahovat do rozvržení pomocí metody drag-and-drop. To vývojáři usnadňuje návrh základního vzhledu aplikace ještě před samotným spuštěním. Každá z platforem nabízí také formu živého náhledu změn. Flutter poskytuje tzv. **hot reload**, který umožňuje okamžité zobrazení změn v UI bez ztráty stavu aplikace. SwiftUI obsahuje funkci **live preview**, která v reálném čase vykresluje rozhraní podle aktuálního kódu. Android Studio podporuje tzv. **live edit** nebo **layout preview**, které fungují podobným způsobem. Ve všech třech případech se jedná o užitečnou vlastnost, která urychluje vývoj a umožňuje rychlejší iteraci návrhu rozhraní.

Navigace mezi jednotlivými stránkami je ve všech typech aplikací velmi podobná. Ve všech případech se volá metoda, která inicializuje přechod na novou obrazovku, přičemž je možné předat také počáteční parametr. V nativních aplikacích lze zpětnou navigaci jednoduše implementovat prostřednictvím šipky v horním panelu nástrojů (toolbaru). Ve Flutter aplikaci funguje navigace na principu zásobníku, kde se jednotlivé obrazovky explicitně při-

dávají a odebírají pomocí příkazů **push** a **pop**.

Všechny verze aplikace rovněž obsahují globální statickou třídu (nebo její ekvivalent), která je dostupná napříč celou aplikací a slouží pro načítání a ukládání dat. Z hlediska implementační logiky se tedy v tomto ohledu platformy výrazně neliší.

Načítání dat probíhalo ve všech verzích aplikace obdobným způsobem. Vytvořilo se spojení se serverem, a po obdržení validní odpovědi bylo tělo zprávy (ve formátu JSON) deserializováno do seznamu objektů. Tento proces mohl být realizován buď pomocí ručního cyklu přes položky, nebo přímým mapováním do předem definovaného modelu. Hlavní rozdíly mezi jednotlivými platformami spočívaly především ve správě stavů obrazovky, fulltextovém vyhledávání a způsobu inicializace komponent rozhraní.

Ve Flutter aplikaci bylo při práci s prvky, které reagují na změny stavu (např. filtrování seznamu), nutné definovat obrazovku jako **StatefulWidget**. Jedná se však o standardní přístup, který není složitý a je běžnou součástí vývoje. V případě nativní Android aplikace bylo nutné vytvořit vlastní adaptér a ručně navázat hodnoty na jednotlivé prvky rozhraní (**ViewHolder** pattern). Tento postup byl z hlediska implementace časově náročnější a méně efektivní ve srovnání s ostatními variantami. Naopak vývoj v nativní iOS aplikaci působil nejjednodušeji a nejintuitivněji, zejména díky deklarativnímu přístupu SwiftUI, který umožňuje přímé propojení dat s rozhraním pomocí deklarativního označení proměnných (property wrapper) jako **@State** či **@ObservedObject**.

V případě práce s databází a lokálním úložištěm vyžadovala Flutter aplikace největší množství externích balíčků, a to jak pro samotnou práci s databází (např. SQLite), tak i pro správu a manipulaci s cestami k uloženým souborům. Každá platforma realizovala tento proces odlišným způsobem, přičemž bylo nezbytné důsledně pracovat s dokumentací použitých knihoven. Ve všech verzích aplikace bylo nutné vytvořit model s jedinečným identifikátorem (**ID**), který sloužil jako primární klíč databázové tabulky. Zároveň bylo potřeba ovládat alespoň základní **SQL** příkazy, zejména pro vytvoření tabulky a pro získání (dotazování) uložených záznamů. Všechny aplikace rovněž využívaly centralizované úložiště nebo správce dat, který zajišťoval přístup k funkcím pro čtení a zápis dat napříč celou aplikací. Ačkoliv jednotlivé kroky implementace se mezi platformami lišily, základní principy zůstaly stejné. Nelze proto jednoznačně určit, který přístup byl nejnáročnější. Ve všech případech probíhala implementace bez větších komplikací.

Pořizování fotografií probíhalo ve všech verzích aplikace obdobně. Ve všech případech bylo nutné přidat do příslušných konfiguračních souborů oprávnění pro přístup ke kameře. Nahrávání obrázku ve formuláři bylo realizováno prostřednictvím tlačítka, které aktivovalo fotoaparát zařízení, a náhled pořízené fotografie se následně zobrazil přímo v komponentě formuláře. Pro získání aktuální polohy zařízení každá platforma využívala vlastní knihovnu, která vrací zeměpisnou šířku a délku. I zde bylo nutné deklarovat odpovídající oprávnění, obdobně jako v případě přístupu ke kameře. Validace formuláře byla realizována ve všech verzích aplikace stejně, a to kontrolou na prázdné hodnoty (null). Při implementaci této funkcionality se nevyskytly žádné zásadní rozdíly ani komplikace.

Největší problémy při vývoji nastaly v oblasti rozšířené reality. Technologie AR se neustále vyvíjí a bylo obtížné najít aktuální a dobře udržované knihovny. Potíže se objevily i při testování. Některé dostupné balíčky nebyly kompatibilní s ostatními knihovnami nebo s konkrétním testovacím zařízením. Zatímco Android využívá **ARCore** a iOS pracuje s **ARKit**, při vývoji hybridní aplikace ve Flutteru bylo velmi náročné najít knihovnu, která by podporovala obě technologie. Knihovny, které tuto podporu nabízely, byly často zastaralé a dlouhodobě neudržované. Také při vývoji nativní Android aplikace se vyskytly komplikace, oficiální knihovny od Googlu nejsou pro běžné použití snadno integrovatelné a bylo nutné sáhnout po komunitních balíčcích dostupných například na GitHubu. Práce s těmito alternativními knihovnami již probíhala hladčeji, ale stále si vyžadovala pečlivé nastudování dokumentace. Nejsnazší a nejstabilnější řešení nabídla nativní implementace pro iOS, kde byl vývoj díky dobře integrovanému frameworku **ARKit** a stabilnímu vývojovému prostředí výrazně jednodušší.

Z hlediska požadavků na operační systém se jednotlivé verze aplikací mírně liší. Nativní iOS aplikace vyžaduje minimálně verzi iOS 17, a to z důvodu využití nejnovějších nativních frameworků a technologií, jako je SwiftUI ve své aktuální podobě nebo knihovny související s ARKit a správou oprávnění. Teoreticky by bylo možné cílit i na nižší verze systému, ovšem za cenu využití zastaralejších komponent nebo knihoven třetích stran, což by mohlo negativně ovlivnit stabilitu a udržitelnost kódu. Hybridní iOS aplikace vyvinutá ve Flutteru si naopak vystačí již s verzí iOS 13, čímž pokrývá širší spektrum zařízení.

U platformy Android jsou požadavky také mírně odlišné. Nativní Android aplikace vyžaduje minimální API úroveň 26 (Android 8.0), především kvůli použití moderních funkcí

systému a knihoven jako Room nebo knihoven pro rozšířenou realitu. Hybridní aplikace postavená na Flutteru vyžaduje minimálně API úroveň 24 (Android 7.0), což opět poskytuje větší kompatibilitu s různými zařízeními.

8 TESTOVACÍ SCÉNÁŘE

Tato část se zaměřuje na testovací scénáře použité při ověřování výkonu nativních verzí aplikací a hybridní varianty. Testování bylo orientováno výhradně na měření rychlosti a výkonu. Scénáře nebyly záměrně navrženy jako extrémně zatěžující, ale odrážejí běžné situace, se kterými se může setkat uživatel jednoduchých mobilních aplikací. Cílem bylo zjistit, jak se liší výkon mezi nativním a hybridním přístupem při typických uživatelských úkonech a který z nich je v těchto případech výhodnější.

Testování pro Android zařízení probíhalo na emulátoru, které emulovalo zařízení Google Pixel 6, přičemž jak nativní, tak hybridní aplikace byly testovány na stejném virtuálním zařízení. Testování pro iOS probíhalo na fyzickém zařízení iPhone 11, aby bylo možné získat přesnější výsledky a zároveň ověřit chování aplikace při využití hardwarových funkcí.

Hlavním cílem testování a následné optimalizace je zajistit co nejrychlejší načtení a vykreslení obsahu stránky pro uživatele. Dalším důležitým faktorem je udržení snímkové frekvence (FPS) ideálně nad hranicí 60 FPS. Měřená aplikace může obsahovat tzv. jank snímky, které říkají, že vykreslení snímku trvalo déle jak 16ms. Tuto hranici je nutné se snažit nepřekročit, aby bylo dodrženo stabilních 60 snímků za sekundu.

8.1 Měření rychlosti

Měření rychlosti se v této části zaměřuje především na responzivitu aplikace, tedy na to, jak rychle je zařízení schopné připravit aplikaci do stavu, kdy ji může uživatel začít aktivně používat. Testování zahrnuje především dobu prvotního spuštění aplikace, rychlost vykreslení uživatelského rozhraní, dobu potřebnou pro navázání spojení se serverem a rychlost následného zpracování obdržených dat.

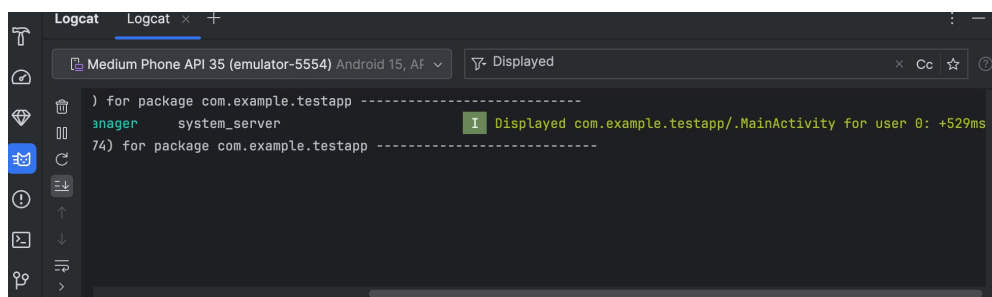
8.1.1 Návrh

Při měření rychlosti bylo cílem procházet jednotlivé obrazovky a zaznamenávat časy stanovených parametrů, které byly zmíněny v úvodu této podkapitoly. Každá obrazovka byla otevírána samostatně a nezávisle na ostatních. Data byla měřena jak pomocí výpisů v aplikačních logech, tak prostřednictvím profileru (nástroje pro analýzu výkonu) dostupného ve vývojovém prostředí. Každý test byl proveden desetkrát, přičemž výsledky byly zprůměrovány s ohledem na případné odchylky. Stejný postup byl použit i při měření rychlosti startu

aplikace, ta byla vždy před spuštěním ukončena z běžícího procesu na pozadí.

8.1.2 Nativní Android testování

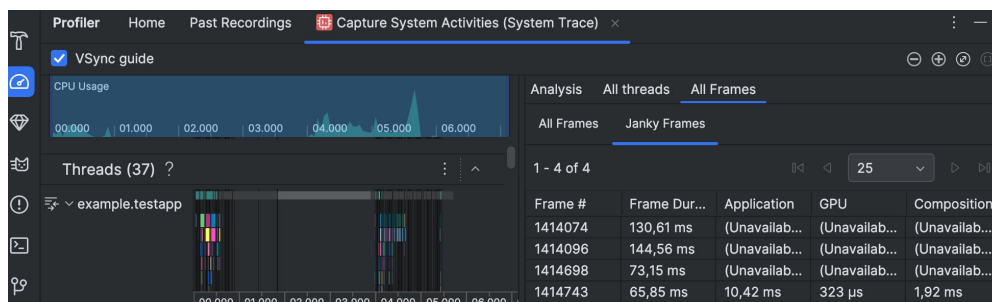
Měření rychlosti u nativní Android aplikace probíhalo ve vývojovém prostředí Android Studio. Pro zjištění doby načtení aplikace ze startovního stavu byl využit nástroj **Logcat**. Jedná se o systémový logovací nástroj, který v reálném čase vypisuje informační, varovné i chybové zprávy z jednotlivých procesů běžících na zařízení. Vzhledem k tomu, že cílem bylo zaznamenat dobu potřebnou pro spuštění aplikace (tzv. **cold start**), nebylo možné filtrovat výpisy podle ID procesu, které se při každém spuštění mění. Místo toho bylo využito filtrování podle textového řetězce „*Displayed com.example.testapp*“. Klíčové slovo „*Displayed*“ označuje okamžik, kdy byla aplikace zcela načtena a připravena k použití. Název balíčku *com.example.testapp* jednoznačně identifikuje testovanou aplikaci. Za tímto logovým výstupem se nachází čas v milisekundách, který udává dobu potřebnou k načtení. Příklad filtrování v Logcatu je znázorněn na obrázku 8.1. Průměrná doba spuštění aplikace byla vypočtena na $474 \pm 53,6$ ms.



Obrázek 8.1 Měření spuštění aplikace v Android Studiu

Měření výkonu probíhalo ve spuštěné aplikaci, přičemž každá stránka byla testována samostatně a nezávisle. Po provedení deseti měření na jedné stránce se pokračovalo na další. Profiler byl spuštěn vždy těsně před vstupem na konkrétní obrazovku. Po načtení stránky a ustálení jejího obsahu bylo měření zastaveno a získaná data byla vyhodnocena. Pro měření doby vykreslení uživatelského rozhraní byl využit nástroj **Capture System Activities**, který umožňuje sledovat čas potřebný k vykreslení jednotlivých UI prvků, počet zobrazených snímků a výskyt tzv. zpožděných (**jank**) snímků. Tento proces je znázorněn na obrázku 8.2. Během měření se rozlišovalo mezi vykreslováním UI (zajišťovaným procesorem – CPU) a rasterizací (zajišťovanou grafickým čipem – GPU). Oba procesy probíhají paralelně, avšak vykreslování na CPU zpravidla začíná dříve a bývá rozděleno do více vláken. Celková doba

vykreslení UI byla určena součtem systémových funkcí, jako jsou **Choreographer** a **Draw**. Rasterizační čas byl získán z funkcí typu **DrawFrame**. Na základě těchto dat byl také vypočten průměrný počet snímků za sekundu (**FPS**). Optimální hodnota FPS pro většinu zařízení je 60. Aby aplikace této hodnoty dosáhla, musí být každý snímek vykreslen do 16 ms. Pokud tento čas překročí, dochází k poklesu plynulosti. Výpočet FPS vycházel z celkového počtu vykreslených snímků a času potřebného pro vykreslení celé stránky.



Obrázek 8.2 Měření spuštění aplikace v Android Studiu

Na základě deseti měření byla průměrná **doba vykreslování UI** následující:

- Stránka ImageEdit: $87,6 \pm 17,87$ ms.
- Stránka Countries: $85,48 \pm 31,3$ ms.

Z těchto údajů vyplývá, že obě stránky vyžadovaly přibližně stejnou dobu pro kompletní vykreslení, ačkoliv stránka Countries vykazuje větší odchylky kvůli načítání.

Rasterizace vykazovala větší rozdíly:

- Stránka ImageEdit: $358,74 \pm 128,72$ ms.
- Stránka Countries: $309,67 \pm 58,06$ ms.

Rasterizace nevykazovala pouze vyšší časové nároky než samotné vykreslování UI, ale také větší odchylky mezi jednotlivými měřeními. Stránka ImageEdit by přitom za běžných okolností měla dosahovat nižší doby rasterizace než stránka Countries, jelikož obsahuje méně komponent a žádný seznam. Vyšší průměrná hodnota u ImageEdit však byla ovlivněna jedním extrémním měřením, kdy došlo k výraznému zpomalení a rasterizace přesáhla 800 ms, což bylo pravděpodobně zapříčiněno přetížením hlavního vlákna. Jednotlivá měření proto zahrnují i výkyvy v chování aplikace, aby co nejvěrněji odrážela její reálný provoz.

Celkový počet snímků použitých k vykreslení byl:

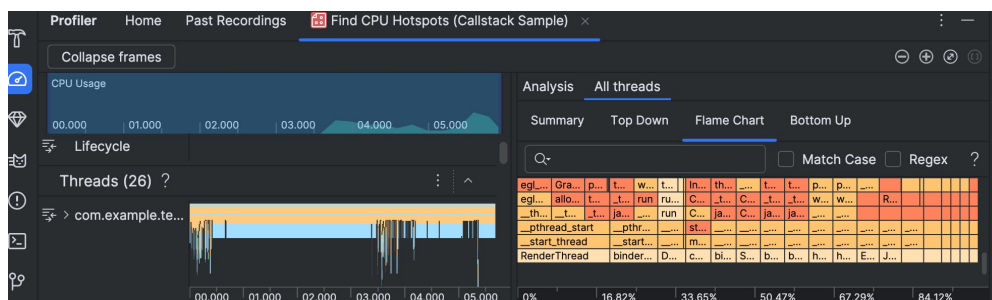
- Stránka ImageEdit: 48 ± 14 ms.
- Stránka Countries: 103 ± 13 ms.

Průměrné FPS:

- Stránka ImageEdit: $71,9 \pm 11,1$ ms.
- Stránka Countries: $85,1 \pm 5,6$ ms.

Stránka Countries obsahuje výrazně vyšší počet snímků než stránka ImageEdit, a to zejména kvůli přítomnosti komponenty ProgressBar, která se zobrazuje během načítání dat. V nativní aplikaci však nedochází k jejímu neustálému překreslování, což výrazně snižuje celkový čas rasterizace. Díky rychlému načtení i vykreslení celé stránky pak bylo dosaženo relativně vysokého průměrného FPS.

Pro analýzu výpočetně náročných funkcí byla využita funkce **Find CPU Hotspots**, která umožňuje zobrazit hierarchii volání metod a dobu jejich běhu. Průběh této analýzy je uveden na obrázku 8.3. Tato metoda byla převážně použita k měření systémových funkcí jako například JSON serializace.



Obrázek 8.3 Měření spuštění aplikace v Android Studiu

Pro měření doby načítání dat z API a výpočtu vlastních funkcí byla použita systémová metoda `System.currentTimeMillis()`, která vrací aktuální čas v milisekundách. Tato metoda byla volána před a po daném měřeném úseku kódu. Odečtením těchto hodnot se získala přesná doba trvání dané operace.

Průměrné naměřené časy pro sledované operace jsou následující:

- API Volání: $1010 \pm 196,8$ ms.
- Zpracování hodnot: $30,06 \pm 11,74$ ms.

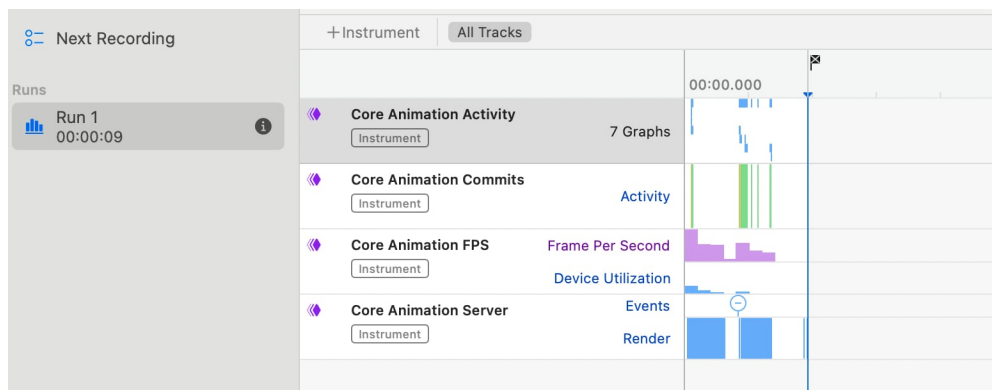
- JSON Serializace: $48,34 \pm 10,12$ ms.
- Geolokace: $42,5 \pm 10,8$ ms.
- DB volání: $1,7 \pm 0,84$ ms.

Naměřená data z jednotlivých operací vykazují převážně stabilní hodnoty s pouze drobnými odchylkami. Výraznější kolísání bylo zaznamenáno u zpracování vlastních funkcí a serializace, které mohou být ovlivněny aktuálním vytížením systému. Také u geolokace a API volání se objevily odchylky, avšak vzhledem k jejich celkovému trvání a poměrovému významu se tyto výkyvy jeví jako zanedbatelné. Podrobnější výsledky jsou uvedeny v příloženém excelu.

8.1.3 Nativní iOS testování

Měření rychlosti u nativní iOS aplikace bylo provedeno pomocí nástroje **Xcode Instruments**, který je součástí vývojového prostředí Xcode a slouží k analýze výkonu aplikací na zařízeních Apple. Pro měření takzvaného **cold startu** byl použit nástroj **Time Profiler**, jenž umožňuje detailní sledování časové náročnosti jednotlivých operací ve vývojovém prostředí. Pro získání potřebných dat bylo nejprve nutné v prostředí Xcode Instruments vybrat konkrétní testovanou aplikaci a poté spustit měření pomocí zvoleného nástroje. Po aktivaci nahrávání aplikace běží v tzv. profilovacím režimu, během kterého nástroj zaznamenává veškeré klíčové události. **Time Profiler** následně zobrazí čas potřebný pro inicializaci hlavního vlákna až po vykreslení prvotní obrazovky aplikace, což odpovídá době cold startu. Na základě deseti měření byla průměrná doba spuštění aplikace vypočtena na $161,3 \pm 4,7$ ms.

Pro měření rychlosti vykreslování a zpracování snímků byl použit nástroj **Core Animation**, viz obrázek 8.4. Pomocí **Core Animation Activity** bylo sledováno vykreslování UI, zatímco nástroj **Core Animation Server** sloužil k měření rasterizace a zaznamenání celkového počtu snímků potřebných k vykreslení jednotlivých obrazovek. Kombinací těchto dvou nástrojů bylo možné získat informace o počtu jank snímků i o délce trvání vykreslení snímků. Přestože **Core Animation** nabízí i vlastní měření FPS, jeho přesnost je omezená. FPS je zobrazováno průběžně i ve chvílích, kdy se na obrazovce nic neděje. Z tohoto důvodu bylo průměrné FPS spočítáno manuálně, obdobně jako při testování nativní Android aplikace.



Obrázek 8.4 Nástroj Core Animation pro měření rychlosti na iOS

Na základě deseti měření byla **průměrná doba vykreslování UI** následující:

- Stránka ImageEdit: $62,35 \pm 8,72$ ms.
- Stránka Countries: $48,78 \pm 5,21$ ms.

Z těchto údajů vyplývá, že složitější stránka ImageEdit vyžaduje delší dobu pro kompletní vykreslení, avšak obě stránky vykazují poměrně stabilní hodnoty.

Rasterizace vykazovala větší rozdíly:

- Stránka ImageEdit: $99,45 \pm 24,41$ ms.
- Stránka Countries: $195,27 \pm 50,85$ ms.

Vyšší odchylky u stránky Countries jsou způsobeny opakovaným překreslováním komponenty **ProgressBar** během načítání dat z API, které zvyšuje nároky na rasterizaci a počet snímků.

Celkový počet snímků použitých k vykreslení byl:

- Stránka ImageEdit: 37 ± 1 ms.
- Stránka Countries: 67 ± 17 ms.

Průměrné FPS:

- Stránka ImageEdit: 62 ± 1 ms.

- Stránka Countries: $36,77 \pm 9,65$ ms.

Zatímco stránka ImageEdit vykazovala plynulé a stabilní vykreslování, stránka Countries byla výrazně proměnlivější. Důvodem je zejména zátěž způsobená načítáním dat z API a opakovaným překreslováním animovaného prvku **ProgressBar**, který prodlužuje čas potřebný pro rasterizaci a celkově snižuje FPS.

Měření rychlosti funkcí, ať už systémových nebo vlastních, bylo prováděno přímo v aplikačním kódu. Stejným způsobem byly měřeny také síťové požadavky. K měření byla použita funkce `CFAbsoluteTimeGetCurrent`, která vrací aktuální čas. Volání této funkce bylo umístěno na začátek a konec měřených bloků kódu. Výsledná doba trvání byla určena odečtením těchto hodnot a následným vynásobením tisícem pro získání hodnoty v milisekundách.

Průměrné naměřené časy pro sledované operace jsou následující:

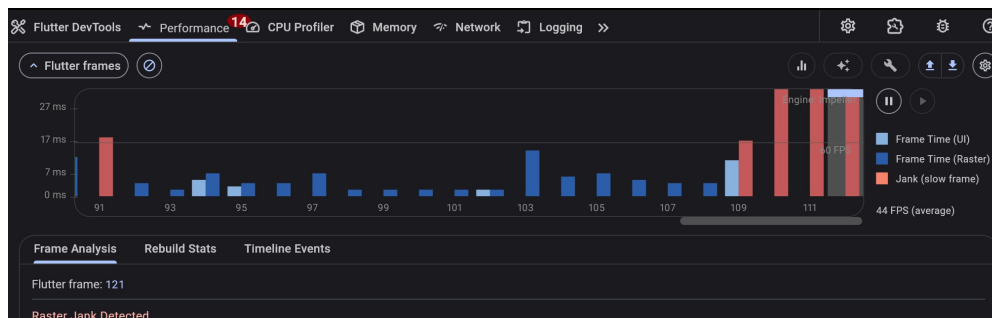
- API Volání: $1462,3 \pm 266,84$ ms.
- Zpracování hodnot: $2,6 \pm 0,48$ ms.
- JSON Serializace: $44,7 \pm 1,48$ ms.
- Geolokace: $1207,34 \pm 1050,84$ ms.
- DB volání: $22,37 \pm 2,44$ ms.

Většina operací vykazuje krátké doby trvání s nízkou odchylkou, což potvrzuje stabilní chování aplikace. Výjimku tvoří zejména API volání a získání geolokace. Geolokační data jsou zvláště proměnlivá. V některých případech se souřadnice získají během několika desítek milisekund, jindy trvá získání hodnot i přes tisíc milisekund. Podrobnější výsledky jsou uvedeny v příloženém excelu.

8.1.4 Flutter testování

Měření rychlosti hybridní aplikace pro Android i iOS probíhalo prostřednictvím nástroje **Flutter DevTools** (viz obrázek 8.5). Tento nástroj umožňuje sledovat průběh vykreslování snímků, včetně doby potřebné pro vykreslení uživatelského rozhraní (UI) a rasterizaci. Zároveň poskytuje přehled o dosažené snímkové frekvenci (FPS) pro konkrétní stránku. DevTools

Lze využít i k monitorování volání funkcí, spotřeby paměti či síťové aktivity, podobně jako nativní vývojové nástroje. Některé specifické typy měření však tento nástroj nezvládá plně pokrýt, a proto byly pro měření doby spuštění aplikace (tzv. **cold start**) využity nativní nástroje Android Studio a Xcode Instruments. Postup měření odpovídal metodám popsaným výše v příslušných podkapitolách. Výsledky měření startu hybridní aplikace pro obě platformy jsou uvedeny v tabulce 8.1.



Obrázek 8.5 Měření rychlosti ve Flutter DevTools

Tabulka 8.1 Naměřené hodnoty startu aplikace pro hybridní aplikace

	Cold Start [ms]
Android	$1035,4 \pm 67,6$
iOS	$276,9 \pm 15,9$

Pro měření doby vykreslování uživatelského rozhraní byla využita sada nástrojů **Flutter DevTools**, konkrétně záložka **Performance**. Tato část nástroje umožňuje analyzovat průběh vykreslování jednotlivých snímků a přehledně odděluje čas strávený na vykreslení UI od rasterizace. Průměrné naměřené hodnoty lze vidět v tabulkách 8.2 a 8.3.

Tabulka 8.2 Průměrné hodnoty vykreslování v hybridních aplikacích na stránce Countries

	Render UI [ms]	Rasterizace [ms]	Min. snímek [ms]	Max. snímek [ms]
Android	$113,3 \pm 8,4$	$833,6 \pm 213,3$	$112,9 \pm 10,9$	$1,9 \pm 0,4$
iOS	$122,3 \pm 12,2$	$336,0 \pm 30,6$	$19,6 \pm 3,8$	$1,4 \pm 0,1$

Jediným omezením je, že jednotlivé snímky je nutné ručně sčítat, protože nástroj neposkytuje souhrnné statistiky. Hodnota FPS je zde naopak automaticky vypočtena a není třeba ji manuálně dopočítávat, na rozdíl od počtu snímků. Tato naměřená data lze vidět v tabulkách 8.4 a 8.5.

Tabulka 8.3 Průměrné hodnoty vykreslování v hybridních aplikacích na stránce ImageEdit

	Render UI [ms]	Rasterizace [ms]	Min. snímek [ms]	Max. snímek [ms]
Android	$39,5 \pm 18,4$	$479,8 \pm 31,4$	$122,8 \pm 8,3$	$16,4 \pm 2,3$
iOS	$25,4 \pm 3,9$	$118,4 \pm 9,5$	$12,4 \pm 2,3$	$3,2 \pm 0,9$

Tabulka 8.4 Průměrné hodnoty snímků v hybridních aplikacích na stránce Countries

	FPS	Jank snímky	Snímky
Android	$42,4 \pm 1,76$	$18,8 \pm 2$	$111,1 \pm 7,9$
iOS	60 ± 0	$1 \pm 0,4$	$120 \pm 11,2$

Pro detailní analýzu volání systémových funkcí byla využita záložka **CPU Profiler**, která nabízí rozpad volání metod a jejich časovou náročnost. Měření doby načítání dat z API a zpracování vlastních funkcí bylo provedeno pomocí systémové třídy **Stopwatch**, která umožňuje měřit čas v milisekundách. Naměřené hodnoty lze vidět na tabulkách 8.6 a 8.7. Výsledky pro ostatní testované stránky společně s detailnějším měřením jsou součástí příloženého souboru ve formátu Excel.

Z naměřených hodnot vyplývá, že přestože se jedná o jednu a tutéž hybridní aplikaci, její chování se na platformách Android a iOS výrazně liší. Tyto rozdíly lze přičíst zejména odlišnostem v hardwarové konfiguraci použitých zařízení a tomu, že Flutter kompiluje kód samostatně do nativního strojového kódu pro každou platformu. Způsob, jakým jednotlivé operační systémy přistupují ke správě paměti, procesorovým vláknům a vykreslování grafiky, také výrazně ovlivňuje celkový výkon.

Naměřená data vykazují rozdíly v téměř všech sledovaných oblastech, přičemž nejvýrazněji se liší časy získávání geolokačních údajů. Tyto rozdíly mohou být způsobeny rozdílnou implementací přístupových mechanismů k lokalizačním službám – například v iOS je lokalizační API silně optimalizováno a integrované s hardwarem, zatímco na Androidu může být závislé na konkrétním výrobci zařízení. Další patrný rozdíl je v plynulosti vykreslování. Verze pro iOS je obecně méně náročná na grafické operace a vykazuje stabilnější průběh, což může být důsledkem lepší optimalizace **Swift-based** prostředí a nižšího počtu variabilních faktorů oproti široké škále Android zařízení.

Pro Android verzi aplikace z naměřených dat vyplývá, že stránka **ImageEdit** působí po-

Tabulka 8.5 Průměrné hodnoty snímků v hybridních aplikacích na stránce ImageEdit

	FPS	Jank snímky	Snímky
Android	$26,8 \pm 1,04$	$10,5 \pm 1,1$	$12,6 \pm 0,92$
iOS	60 ± 0	$0,1 \pm 0,18$	$24,8 \pm 0,68$

Tabulka 8.6 Průměrné hodnoty volání v hybridních aplikacích na stránce Countries

	API Volání [ms]	Zpracování hodnot [ms]	JSON Serializace [ms]
Android	$1181 \pm 225,2$ ms	$8,2 \pm 3,12$ ms	$212,6 \pm 57$ ms
iOS	$1738,8 \pm 186,36$ ms	$18,8 \pm 3,44$ ms	$166 \pm 18,4$ ms

měrně plynule, i přes nižší hodnoty FPS. Výraznější rozdíly se však mohou objevit při ukládání dat do databáze nebo při získávání geolokačních údajů, kde doba operací kolísá. Naproti tomu stránka **Countries** vykazuje značnou proměnlivost zejména v oblasti vykreslování. Zvýšené hodnoty rasterizace a počtu snímků jsou zde částečně způsobeny přítomností komponenty **ProgressBar**, která se během načítání dat z API animuje a tím generuje dodatečné snímky, které musí být opakovaně rasterizovány. U iOS verze aplikace byla největší proměnlivost zaznamenána při volání API, což je však vzhledem k povaze síťových operací běžné. Nejvýraznější výkyvy pak nastávaly při získávání geolokačních údajů, které mohou být ovlivněny různými faktory, například kvalitou signálu nebo dostupností služeb. Přesto se všechny testované stránky jeví jako poměrně plynulé a bez výraznějších problémů při vykreslování či interakci.

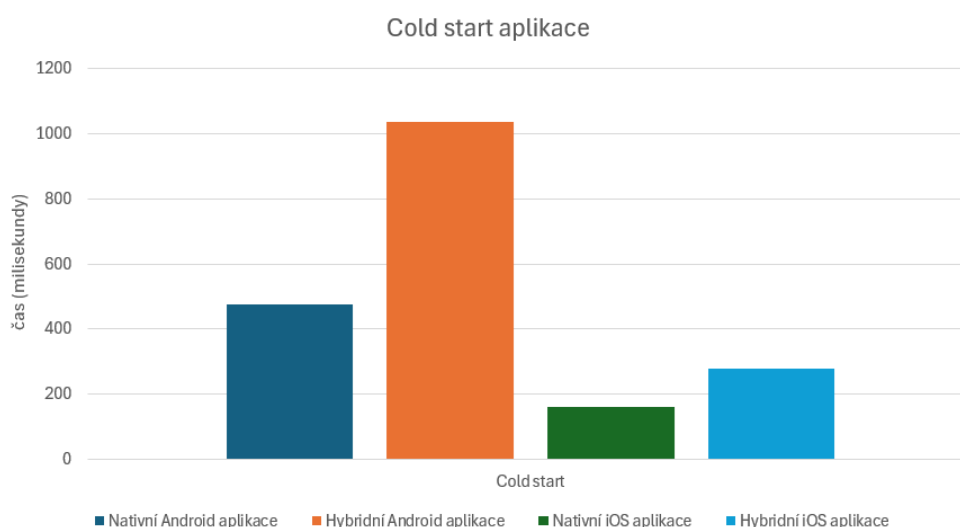
8.1.5 Závěr

První měření se zaměřovalo na tzv. **cold start**, tedy dobu potřebnou pro spuštění aplikace ze zcela neaktivního stavu. Výsledky všech čtyř variant aplikace jsou znázorněny v grafu na obrázku 8.6. Z naměřených hodnot je patrné, že nativní aplikace spouští rychleji než jejich hybridní protějšky, což potvrzuje obecně známý předpoklad o vyšší efektivitě nativního kódu při inicializaci. Zajímavým zjištěním však je, že iOS aplikace – jak nativní, tak hybridní – dosáhly kratší doby spuštění než nativní Android verze. Tento výsledek může být ovlivněn rozdíly v hardwaru zařízení, optimalizacích operačního systému nebo přístupem k inicializaci aplikace.

Druhé měření se zaměřovalo na výkon konkrétních funkcí v aplikaci, jako je volání API,

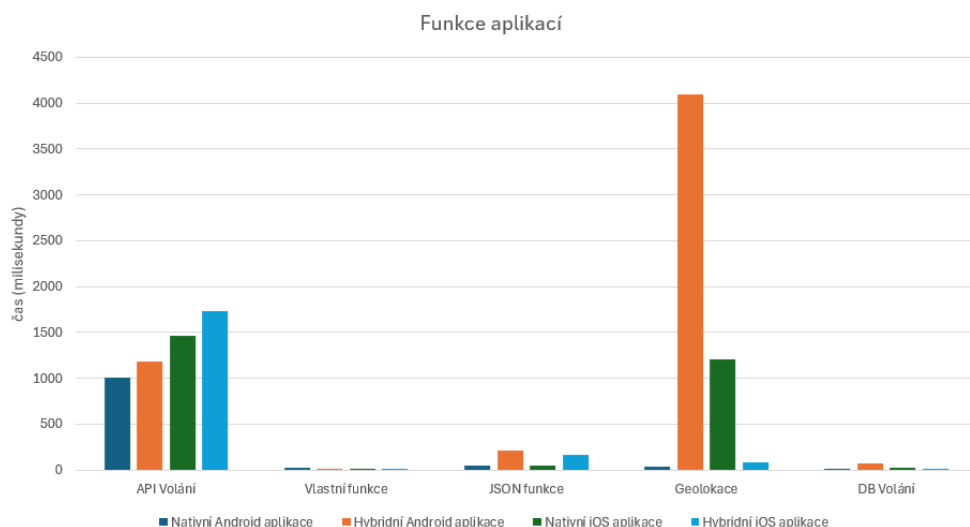
Tabulka 8.7 Průměrné hodnoty volání v hybridních aplikacích na stránce ImageEdit

	Geolokace [ms]	DB Volání [ms]
Android	4089,5 ± 280,3 ms	69,3 ± 22,24 ms
iOS	83,3 ± 29,9 ms	13,5 ± 2,9 ms



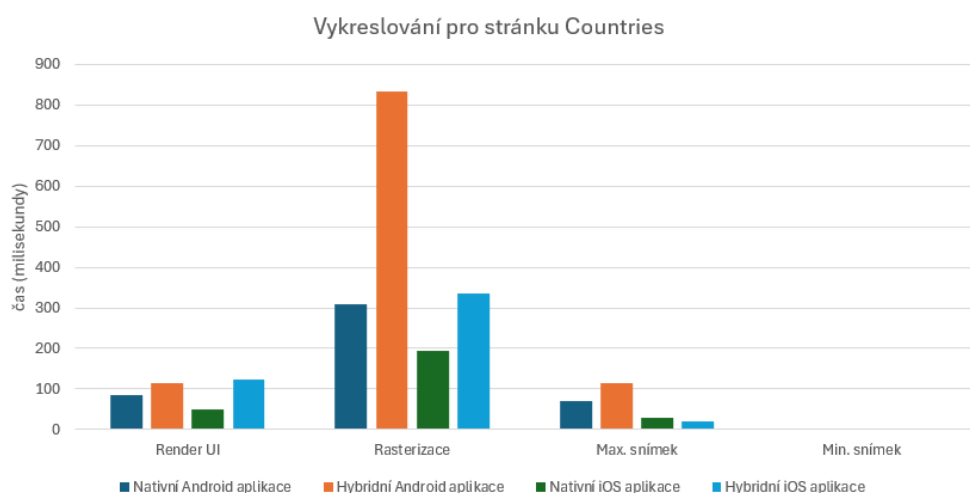
Obrázek 8.6 Výsledky měření startu aplikace

práce s databází, serializace dat nebo získávání polohy zařízení. Výsledky jsou vizualizovány v grafu na obrázku 8.7. Z dat vyplývá, že volání API bylo poměrně proměnlivé napříč platformami. Nejrychleji si vedla nativní Android aplikace, zatímco nejpomalejší odezvy byly zaznamenány u hybridní iOS aplikace. Tyto rozdíly však nelze jednoznačně přisoudit samotné platformě, jelikož mohou být ovlivněny kolísáním kvality internetového připojení během měření. Oproti tomu zpracování JSON dat, konkrétně serializace a deserializace, bylo nejpomalejší u hybridních aplikací na obou platformách. Tento výsledek je očekávatelný, jelikož hybridní řešení často využívají obecnější, méně optimalizované knihovny pro práci s daty. Volání vlastních funkcí a operace s databází vykazovaly mezi jednotlivými aplikacemi jen minimální rozdíly. Tyto rozdíly nebyly statisticky významné a výkon byl u všech variant srovnatelný. Nejvýraznější rozdíly se projevily při získávání geolokačních údajů. Hybridní Android aplikace měla v tomto směru největší zpoždění, zatímco nativní verze si vedla velmi dobře. Naopak u iOS aplikací byl výsledek opačný, hybridní verze dosáhla stabilních a rychlých výsledků, zatímco nativní iOS aplikace byla značně nestabilní. Jak již bylo zmíněno v předchozí podkapitole 8.1.3, doba načtení polohy v nativní aplikaci kolísala od desítek milisekund až po jednotky sekund, což výrazně ovlivnilo výpočet průměrné hodnoty.



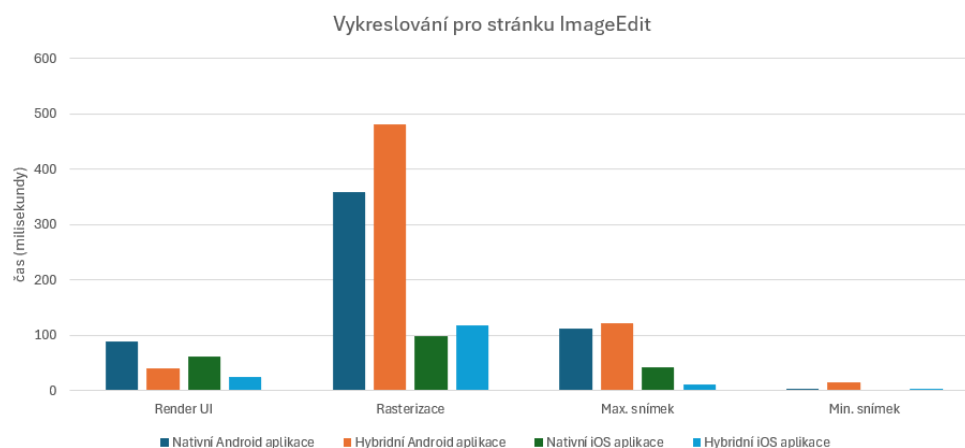
Obrázek 8.7 Výsledky měření rychlosti funkcí aplikace

Třetím testovaným scénářem byla rychlost vykreslování uživatelského rozhraní a rasterizace. Pro přesnější vyhodnocení bylo nutné analyzovat data ze dvou různých obrazovek, protože každá z nich vykresluje obsah odlišným způsobem. Stránka „Countries“ obsahuje seznam dat a před jejich načtením navíc zobrazuje komponentu ProgressBar, která zvyšuje počet vykreslovaných snímků. Naopak stránka „ImageEdit“ je jednoduchá obrazovka obsahující pouze formulář s textovým polem, komponentou pro výběr obrázku a tlačítkem pro uložení. Data pro stránku „Countries“ jsou vizualizována na obrázku 8.8, zatímco výsledky pro stránku „ImageEdit“ jsou uvedeny na obrázku 8.9.



Obrázek 8.8 Výsledky měření vykreslování aplikace pro stránku Countries

Z naměřených dat je patrné, že v oblasti rasterizace hybridní aplikace obecně zaostá-

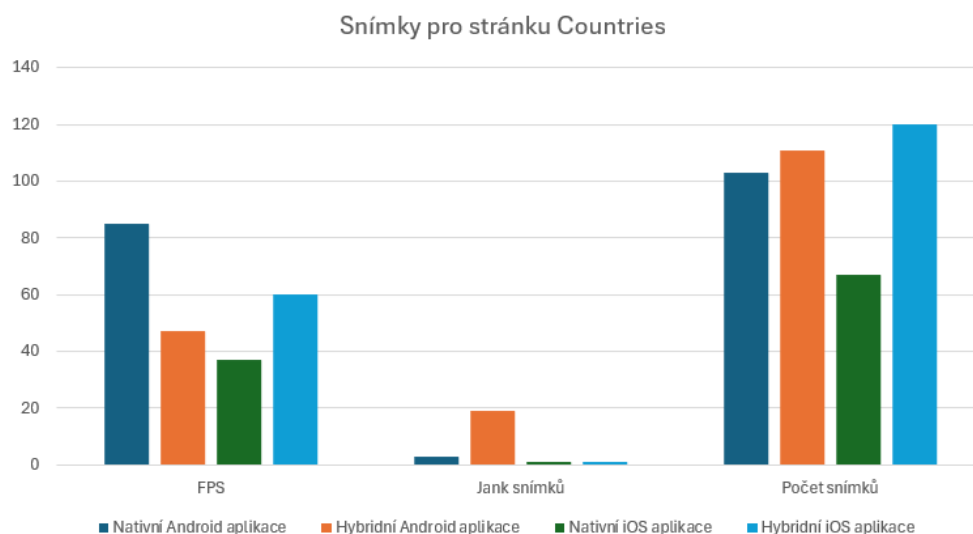


Obrázek 8.9 Výsledky měření vykreslování aplikace pro stránku ImageEdit

valy za nativními verzemi. Nejhuře si vedla hybridní aplikace pro Android, která vykazovala nejvyšší časy rasterizace. Zajímavým zjištěním však je, že na stránce ImageEdit dokázala hybridní aplikace pro iOS překonat dokonce i nativní verzi pro Android. Celkově lze říci, že rasterizace je znatelně rychlejší u nativních aplikací, přičemž nejlepší výsledky byly zaznamenány na zařízeních s iOS. Co se týče rychlosti vykreslování uživatelského rozhraní, výsledky byly proměnlivější. Na stránce Countries byly rychlejší nativní aplikace, zatímco na stránce ImageEdit se lépe vedlo hybridním verzím. Tyto rozdíly mohou být způsobeny specifiky implementace nebo optimalizací daných obrazovek. V obou případech však vedly aplikace pro iOS, bez ohledu na to, zda šlo o nativní či hybridní řešení. V případě doby trvání nejdelšího a nejkratšího snímku byly výsledky konzistentní. Android aplikace se obecně potýkaly s více „zaseknutými“ snímky, zatímco iOS aplikace měly tyto hodnoty nižší a stabilnější. Zajímavé je, že hybridní aplikace pro iOS v tomto ohledu téměř dorovnala nativní variantu, což svědčí o dobré optimalizaci Flutter engine na platformě iOS.

Závěrečné testování se zaměřovalo na měření počtu snímků za sekundu, celkového počtu snímků potřebných k vykreslení obrazovky a výskytu tzv. jank snímků. Pro srovnání byly opět vybrány dvě stránky, Countries a ImageEdit, protože, jak již bylo patrné z předchozích měření vykreslování, hodnoty se mezi nimi výrazně liší. Výsledky tohoto testu jsou zobrazeny v grafech na obrázcích 8.10 a 8.11.

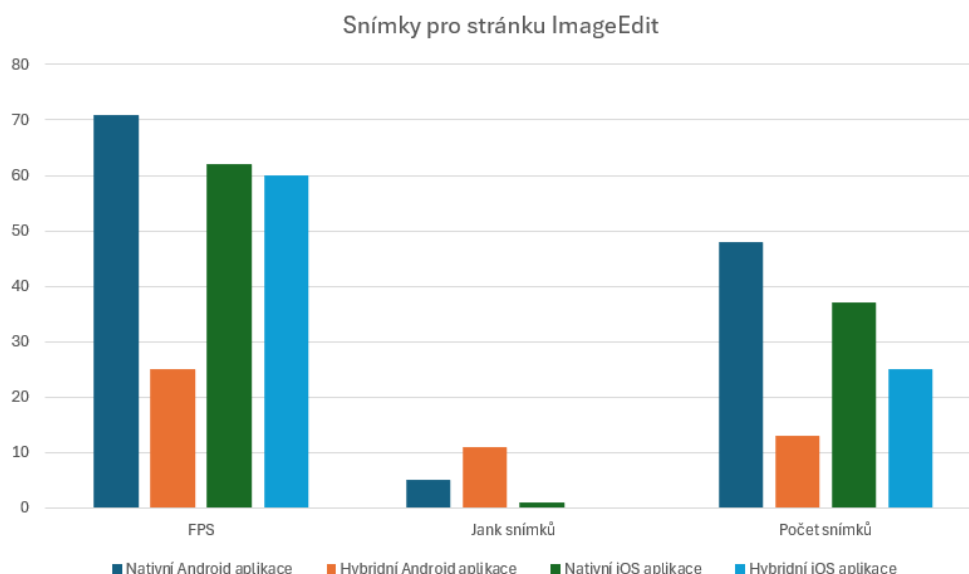
Z grafů je patrné, že se obě stránky výrazně liší. Zvláště nápadný je vztah mezi sloupci FPS a Počet snímků, jejich hodnoty jsou do jisté míry opačné. Stránka Countries vykazuje vyšší počet vykreslených snímků, ale nižší FPS, zatímco stránka ImageEdit má méně



Obrázek 8.10 Výsledky měření snímků aplikace pro stránku Countries

snímků, ale vyšší FPS. Tento rozdíl je způsoben tím, že stránka ImageEdit obsahuje méně komponent, zatímco stránka Countries během načítání zobrazuje ProgressBar, který se průběžně znovu vykresluje, což zvyšuje počet snímků potřebných pro vykreslení. Z výsledků je zřejmé, že nejhorších hodnot dosahuje hybridní Android aplikace, která má nejnižší FPS a nejvyšší počet jank snímků. Zajímavostí ale je, že nativní Android aplikace překonává jak hybridní, tak i nativní iOS aplikaci, a to i přesto, že v měření pro vykreslování a rasterizace měla horší časy a má i dokonce více jank snímků. Tento výsledek může být ovlivněn rozdílnými metodami výpočtu FPS. Ve Flutter aplikaci byly hodnoty FPS přímo zobrazovány nástrojem Flutter DevTools a pohybovaly se stabilně kolem 60 FPS, především u iOS verze, která vykazovala minimální počet jank snímků. U nativních aplikací však bylo FPS počítáno ručně na základě celkového počtu snímků a času vykreslení. Zde platí, že pokud stránka trvala déle než jednu sekundu načíst, pak byla výsledná hodnota FPS nižší než počet snímků. Naopak u rychleji načtených obrazovek byl počet snímků vyšší než vypočtené FPS. Navzdory těmto rozdílům zůstávala většina aplikací vizuálně stabilní a plynulá i při nižších hodnotách FPS.

Na závěr je možné usoudit, že nativní aplikace si ve většině případů vedly lépe než jejich hybridní protějšky. Výkonnostní výhoda nativního přístupu se projevila zejména v rychlosti odezvy, nižším zatížení systému a stabilnějším chování při náročnějších operacích. Zároveň však platí, že ve zhruba polovině případů, zejména při vykreslování uživatelského rozhraní, dosahovala nejlepší výsledky iOS aplikace. Pozoruhodné také je, že hybridní verze pro iOS



Obrázek 8.11 Výsledky měření snímků aplikace pro stránku ImageEdit

v některých testech překonala nativní Android aplikaci, což může závist na hardwaru iOS zařízení.

Data do grafů byla brána z průměrných hodnot bez odchylek, které byli vypočítány z deseti různých měření. Všechna data a to i pro všechny stránky je možné nalézt v příloze v Excelu.

8.2 Měření výkonu

Měření výkonu aplikace bylo zaměřeno především na ověření, zda aplikace běží plynule, bez zasekávání nebo zpomalení. Testování spočívalo v monitorování aktivity procesoru, zátěže operační paměti a spotřeby baterie během běžného používání aplikace. Měření výkonu pro Android aplikace muselo probíhat na fyzickém zařízení z důvodu měření spotřeby energie a testování rozšířené reality. Konkrétně na modelu Samsung A34.

8.2.1 Návrh

Měření výkonu probíhalo tak, že při spuštění aplikaci bylo aktivováno měření telemetrických dat a sledovaly se výkyvy při přechodu mezi jednotlivými obrazovkami. Při každém přechodu se vyčkalo, dokud se aplikace nedostala do klidového stavu, a následně se pokračovalo v navigaci, přičemž se zaznamenávaly další výkyvy za účelem identifikace nejvíce zatěžujících procesů. Tento postup byl několikrát zopakován, aby bylo možné ověřit, zda se

jedná o opakovaný jev, nebo o jednorázovou odchylku ve výkonu. Využití procesoru a paměti bylo sledováno průběžně a výsledky byly zobrazeny ve formě časových grafů. Spotřeba baterie byla měřena především při použití rozšířené reality, ale hodnoty jsou zaznamenány i v rámci předchozích testovacích scénářů.

Byly navrženy tři testovací scénáře, přičemž každý z nich byl měřen po dobu deseti sekund. První scénář se zaměřoval na navigaci z domovské obrazovky na stránku se seznamem států, kde probíhalo rolování a vyhledávání pomocí fulltextového pole. Druhý scénář spočíval ve vytvoření obrázku a následné zpětné navigaci na seznam. Třetí scénář zahrnoval přechod na stránku s rozšířenou realitou a vložení 3D modelu do scény.

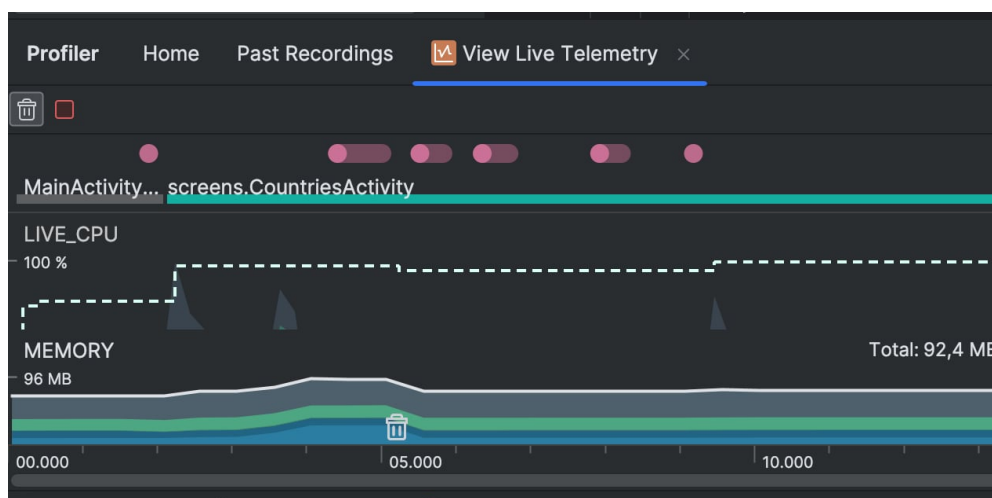
Pro zaznamenávání dat z procesoru byly definovány konkrétní momenty z testovacích scénářů jako například scrollování, ze kterých byla vybrána nejvyšší dosažená hodnota v daném čase. Při sledování využití operační paměti byly zaznamenávány výkyvy v průběhu času celého testovacího scénáře, přičemž za celý testovací průběh byla zaznamenána jak nejnížší, tak nejvyšší naměřená hodnota.

8.2.2 Nativní Android testování

Měření výkonu pro Android aplikaci bylo prováděno ve vývojovém prostředí Android Studio v nástroji Profiler. Pro měření využití procesoru byl využit nástroj **Find CPU Hotspots**, který zobrazuje, jaká vlákna a funkce zatěžují procesor v čase, včetně doby jejich vykonávání. Pro sledování využití paměti RAM slouží nástroj **Track Memory Consumption**, jenž umožňuje sledovat alokaci paměti, uvolňování objektů a práci **Garbage Collectoru**. Oba tyto nástroje umožňují vývojářům získat přesný a časově závislý přehled o tom, jak aplikace zatěžuje zařízení při jednotlivých operacích. Pro zjednodušený a vizuálně přehledný pohled na výkon v reálném čase byl použit nástroj **View Live Telemetry**. Ten zobrazuje souběžné grafy využití CPU a paměti, což usnadňuje srovnání mezi různými běhy aplikace nebo mezi různými verzemi (např. nativní vs. hybridní). Tento nástroj byl hlavním prostředkem pro vizuální srovnání výkonu během testovacích scénářů. Pro závěrečné měření a srovnání byly použity již zmíněné nástroje, které tyto data zobrazují přesně. Tyto hodnoty jsou zobrazeny v závěru podkapitoly.

Na prvním obrázku 8.12 je zachycen průběh prvního testovacího scénáře, který zahrnoval přechod na stránku se seznamem států. V grafu CPU lze pozorovat náhlý, avšak mírný ná-

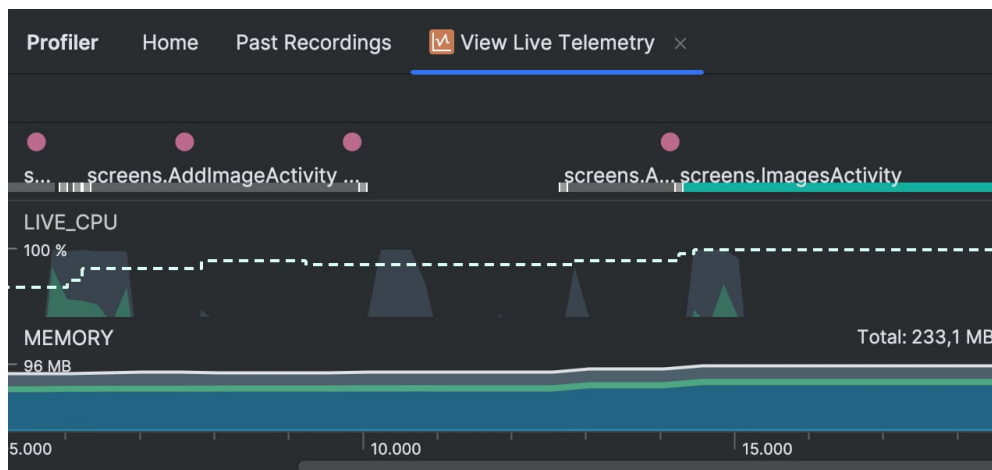
růst výkonu při vykreslování seznamu, což je zcela běžné chování. Přerušovaná čára v grafu zobrazuje změnu počtu použitých procesorových vláken. V grafu využití paměti lze identifikovat momenty, kdy **Garbage Collector** uvolnil objekty z paměti, což se projevilo poklesem spotřeby. Po krátkém období nečinnosti následuje fáze scrollování seznamu, která je v grafu označena červenými tečkami, ty představují doteky uživatele na displej. S každou interakcí dochází ke zvýšení aktivity CPU, i když ve stále akceptovatelné míře. Poslední znatelný výkyv výkonu nastal při zadávání textu do fulltextového pole, kdy se aktivovalo filtrování dat. Přechod na jinou stránku je z grafu také možné pozorovat změnou aktivity.



Obrázek 8.12 Měření výkonu nativní android aplikace pro scénář 1

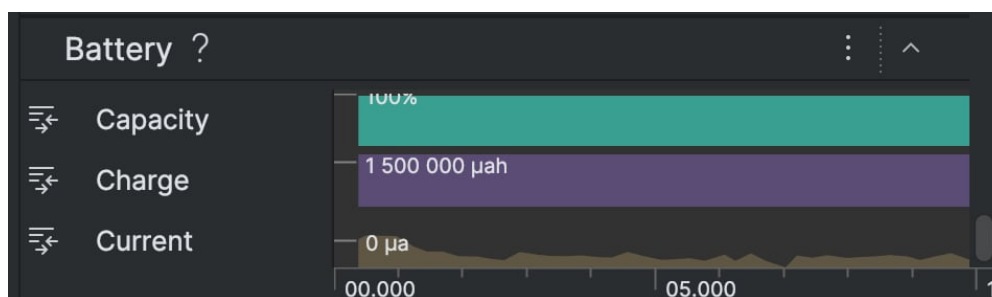
Na obrázku 8.13 je zachycen průběh druhého testovacího scénáře. V jeho rámci lze pozorovat výrazné zvýšení využití CPU a paměti během inicializace formulářových prvků. Spotřeba paměti je v tomto případě podstatně vyšší než u prvního scénáře. V grafu aktivity lze rovněž identifikovat časovou mezeru, která naznačuje otevření fotoaparátu zařízení. Tato událost je následována dvěma výraznými špičkami v grafu CPU, odpovídajícími pořízení fotografie a jejím zpracování. Závěrečný výkyv pak značí uložení obrázku do databáze a následnou zpětnou navigaci na předchozí obrazovku.

Pro třetí testovací scénář, zaměřený na sledování spotřeby baterie při použití rozšířené reality, byl využit nástroj **Capture System Activities**, který umožňuje zobrazit aktuální spotřebu energie. Výsledky tohoto měření jsou zachyceny na obrázku 8.14. Grafy zobrazují kapacitu baterie, úroveň nabití zařízení a aktuální odběr či přírůstek energie v reálném čase. Spotřeba energie je vyjádřena jako proud baterie v mikroampérech. Konkrétně graf s názvem **Current** ukazuje, kolik mikroampér bylo v daném okamžiku odebráno nebo přidáno. Tento



Obrázek 8.13 Měření výkonu nativní android aplikace pro scénář 2

graf tedy neudává souhrnnou spotřebu (např. v ampérhodinách), ale okamžité hodnoty odběru proudu. Do výsledného srovnání byla zahrnuta maximální naměřená hodnota odběru. Z grafu lze vyčíst, že v nulté sekundě začíná hodnota nad nulou, což je způsobeno tím, že zařízení bylo během testování zároveň nabíjeno. Po přechodu na stránku s rozšířenou realitou a přidání modelu na scénu je možné pozorovat poklesy pod nulovou hodnotu, které indikují začátek vybíjení baterie.



Obrázek 8.14 Měření výkonu nativní android aplikace pro scénář 3

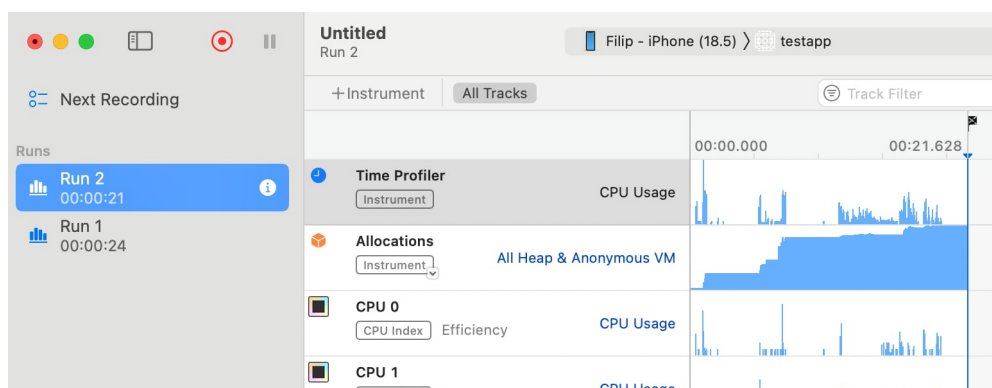
Z testovacích scénářů je možné vidět, že práce s objekty a obrázky je nejvíce náročná na paměť zařízení, zatímco práce se seznamem je nejvíce náročná na CPU.

8.2.3 Nativní iOS testování

Měření výkonu iOS aplikace probíhalo pomocí nástroje **Xcode Instruments**. Pro sledování vytížení procesoru byl použit nástroj **Time Profiler**, zatímco nástroj **Allocations** sloužil ke sledování využití operační paměti RAM. Tyto nástroje umožňují velmi přesně zaznamenat aktuální zátěž systému a spotřebu prostředků v konkrétních momentech běhu aplikace. Díky

možnosti grafického zobrazení dat v reálném čase lze výsledky snadno analyzovat a vizualizovat pomocí přehledných grafů.

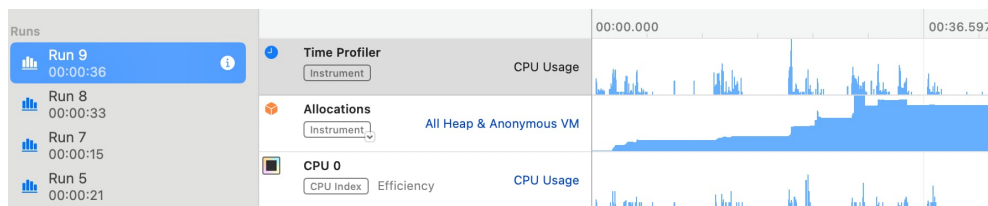
První testovací scénář je vizualizován na obrázku 8.15. Graf CPU aktivity je rozdělen na jednotlivé úseky, které odrážejí fáze provádění operací a období nečinnosti. Úvodní nárůst signalizuje spuštění aplikace. Následující nárůst pak označuje začátek samotného testovacího scénáře, konkrétně přechod na stránku se seznamem států. Po krátké odmlce začíná fáze scrollování seznamem, přičemž závěrečný prudší výkyv značí spuštění fulltextového vyhledávání. V grafu využití operační paměti RAM je patrné, že v každé z těchto fází dochází k nárůstu spotřeby paměti.



Obrázek 8.15 Měření výkonu nativní iOS aplikace pro scénář 1

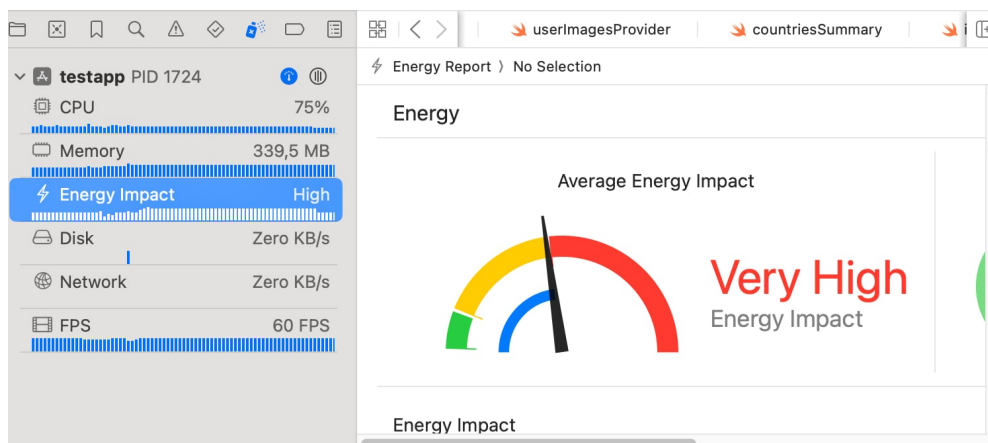
Na obrázku 8.16 znázorňujícím druhý testovací scénář jsou patrné rozdíly oproti scénáři prvnímu. Využití CPU opět začíná úvodním úsekem při startu aplikace. Druhý úsek zachycuje práci s formulářovými prvky. Největší nárůst je patrný ve třetím úseku, který odpovídá otevření fotoaparátu. Po krátké odmlce následuje fáze pořízení fotografie, což se rovněž projevuje na aktivitě procesoru. Závěrečný nárůst značí uložení snímku do databáze. Graf využití operační paměti RAM má podobný průběh jako v předchozím scénáři. Paměť postupně narůstá, přičemž největší nárůst nastává při práci s fotoaparátem. Následně dochází k částečnému uvolnění paměti a krátkodobému poklesu jejího využití.

Pro třetí testovací scénář byl využit nástroj **Energy Report**, který není součástí sady Instruments, ale je přímo integrován ve vývojovém prostředí Xcode. Tento nástroj slouží ke sledování spotřeby energie aplikace. Neudává aktuální spotřebu v jednotkách jako mWh, ale poskytuje přehled o tom, jaké části aplikace zatěžují baterii a v jaké míře. Místo konkrétních čísel tedy ukazuje relativní spotřebu podle komponent, například paměti, CPU nebo senzory



Obrázek 8.16 Měření výkonu nativní iOS aplikace pro scénář 2

zařízení. Výsledky tohoto měření jsou zachyceny na obrázku 8.17. Z něj je patrné, že během provozu rozšířené reality je výrazně zatěžována paměť a celková spotřeba energie je klasifikována jako velmi vysoká. Při přidání 3D modelu na scénu došlo k výraznému nárůstu spotřeby, která se následně při prohlížení modelu ustálila na hranici vysoké až velmi vysoké zátěže. Rozšířená realita zatěžovala CPU i GPU. Z celkové vytíženosti CPU bylo vytíženo přibližně na 57 % a GPU na 43 %.



Obrázek 8.17 Měření výkonu nativní iOS aplikace pro scénář 3

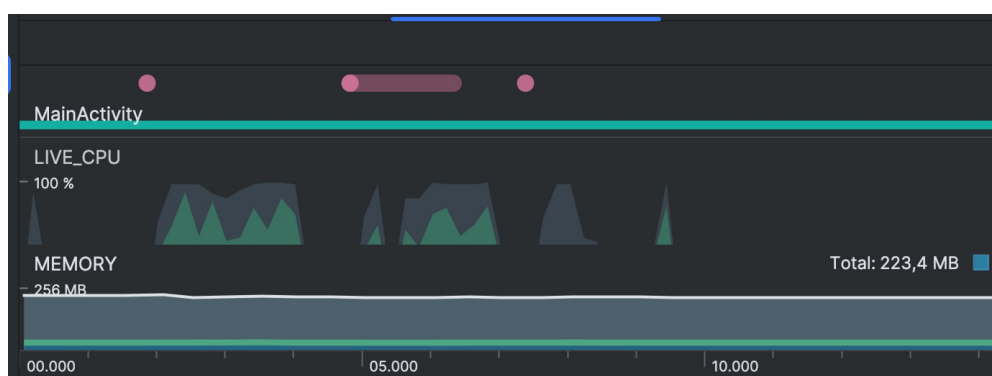
Stejně jako v případě testování na platformě Android se ukázalo, že práce s rozšířenou realitou je nejvíce náročná na operační paměť. Využití procesoru je naopak nejvyšší při interakci s formulářovými prvky, zejména během pořizování fotografie. Samotné ukládání fotografie představuje větší zátěž pro paměť, než pro CPU. Obecně lze říci, že akce spojené s přechody mezi obrazovkami zatěžují procesor více než běžné operace na jedné stránce. Tato skutečnost i platí pro jakékoliv jednorázové akce.

8.2.4 Flutter testování

Měření pro Flutter aplikaci se lišilo v závislosti na cílové platformě, tedy podle toho, zda šlo o verzi pro Android nebo iOS. Samotné testování však probíhalo obdobným způsobem jako

u nativních aplikací, a byly při něm využity totožné nástroje a metodiky odpovídající dané platformě.

U Android aplikace lze v prvním testovacím scénáři, zachyceném na obrázku 8.18, pozorovat, že počáteční kliknutí signalizuje přechod na stránku se seznamem států. V tomto okamžiku došlo k výraznému nárůstu využití CPU, který přetrvával po celou dobu načítání dat. Následné scrollování seznamem vykazuje v grafu podobný průběh se zvýšenou aktivitou. Závěrečná špička v aktivitě a související kliknutí odpovídají spuštění fulltextového vyhledávání.



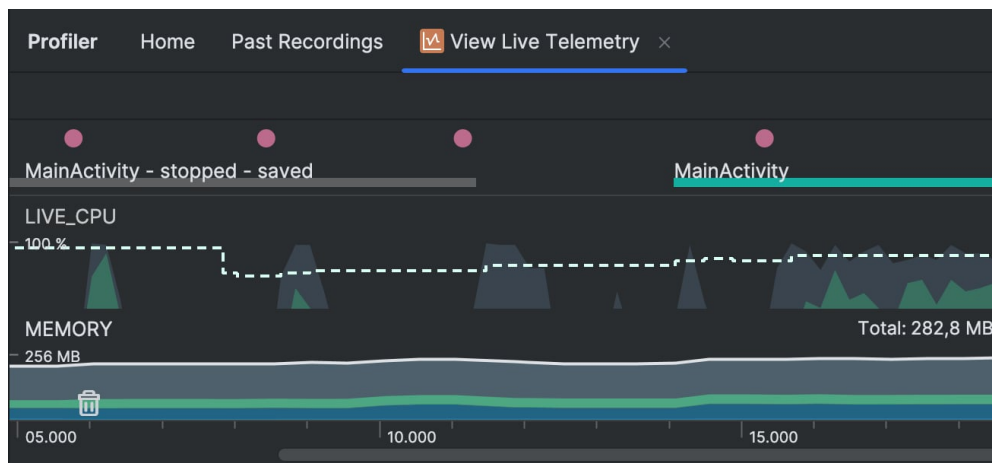
Obrázek 8.18 Měření výkonu hybridní Android aplikace pro scénář 1

Ve druhém scénáři lze z obrázku 8.19 pozorovat, že k největšímu výkyvu dochází při načtení stránky a inicializaci formulářových prvků. Následující zvýšení aktivity odpovídá zadávání textu do vstupního pole. Výraznější změny při přepínání mezi aktivitami signalizují otevření a použití fotoaparátu. Závěrečný nárůst pak souvisí s ukládáním dat do databáze a současným získáváním geolokačních údajů zařízení.

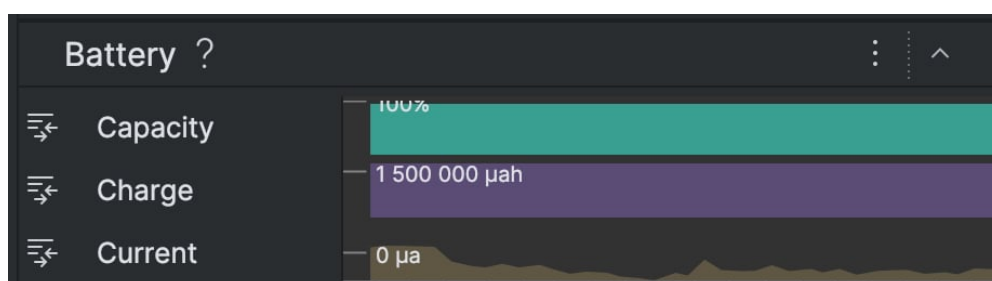
Pro třetí scénář lze z grafu na obrázku 8.20 pozorovat spotřebu energie. Průběh je téměř totožný s nativní aplikací, avšak v tomto případě je patrný o něco výraznější pokles v odběru energie.

U Flutter aplikace pro Android lze při práci se seznamem a vykreslování jednotlivých prvků pozorovat zvýšenou aktivitu procesoru. Spotřeba operační paměti RAM zůstává ve všech sledovaných scénářích relativně vysoká.

Pro první scénář hybridní iOS aplikace lze na obrázku 8.21 pozorovat, že procesor je aktivní po celou dobu měření. Podobně jako při testování nativní iOS aplikace jsou patrné tři hlavní úseky aktivity, přičemž mírná zátěž je viditelná i mezi nimi. Využití operační paměti

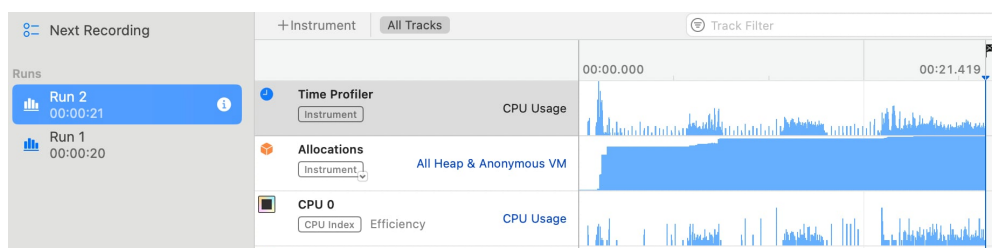


Obrázek 8.19 Měření výkonu hybridní Android aplikace pro scénář 2



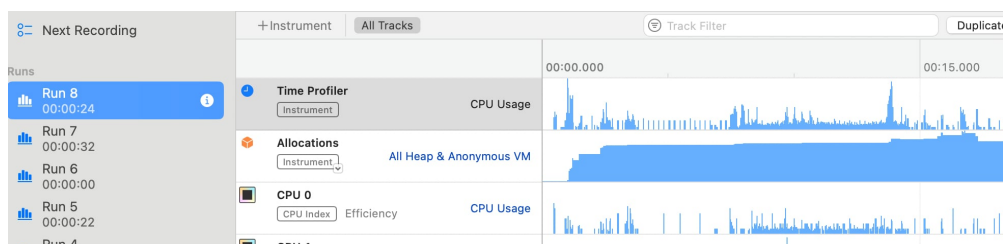
Obrázek 8.20 Měření výkonu hybridní Android aplikace pro scénář 3

RAM je vysoké již od samotného začátku testu a téměř se po celou dobu běhu nezvětšuje.



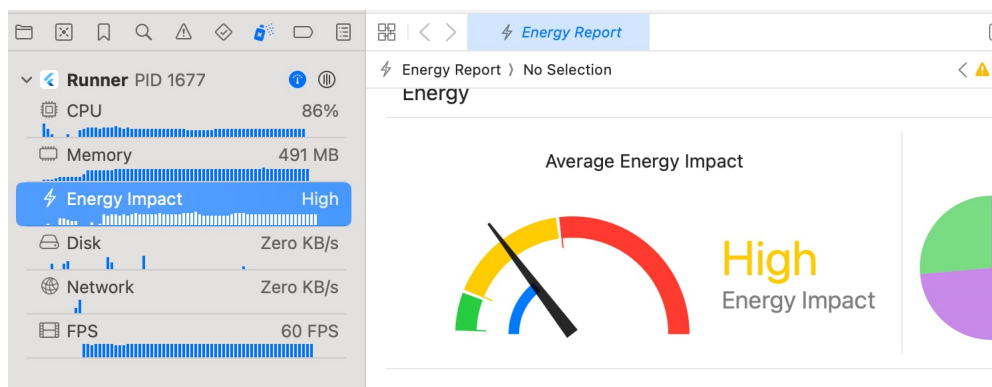
Obrázek 8.21 Měření výkonu hybridní iOS aplikace pro scénář 1

Pro druhý scénář hybridní iOS aplikace na obrázku 8.22 lze pozorovat stejně jako v prvním scénáři neustálou aktivitu na procesoru. Výraznější nárůst aktivity nastává po úvodní mezeře, která značí načtení stránky pro práci s obrázkem. Tento vzrůst je následován kontinuální zátěží, zakončenou prudkým výkyvem při otevření fotoaparátu. Operační paměť (RAM) je výrazně využita již na začátku testu a po většinu průběhu roste jen mírně. K výraznějšímu nárůstu dochází až při aktivaci fotoaparátu a pořízení snímku, po čemž následuje opětovný pokles.



Obrázek 8.22 Měření výkonu hybridní iOS aplikace pro scénář 2

Pro třetí scénář hybridní iOS aplikace lze na obrázku 8.23 pozorovat výrazné využití operační paměti. Spotřeba energie je rovněž velmi vysoká. Po přidání 3D modelu na scénu došlo k prudkému nárůstu spotřeby, která se ustálila přibližně v polovině oblasti označené jako vysoká. Na rozdíl od ostatních sledovaných parametrů však zatížení procesoru v tomto scénáři nebylo příliš výrazné. Z celkové vytíženosti CPU bylo vytíženo přibližně na 77 % a GPU na 23 %.



Obrázek 8.23 Měření výkonu hybridní iOS aplikace pro scénář 3

U Flutter aplikace pro iOS lze v obou scénářích pozorovat vysoké a téměř konstantní využití paměti, přičemž rozdíly mezi nimi jsou minimální. Pokud jde o zatížení procesoru, práce se seznamem vykazuje pravidelné výkyvy, zatímco práce s formulářem a fotoaparátem způsobuje trvalou zátěž procesoru bez výrazných útlumů. Nejvyšší zatížení paměti lze pozorovat ve třetím scénáři při práci s rozšířenou realitou. V tomto scénáři je také velká spotřeba baterie zařízení.

8.2.5 Závěr

Pro účely měření výkonu není vhodné přímo porovnávat Android a iOS verze aplikací mezi sebou. Smysluplnější je srovnání nativní a hybridní varianty v rámci jedné platformy. Dů-

vodem není pouze rozdílný hardware obou zařízení, ale také odlišný způsob, jakým se měří zatížení procesoru. Android Studio zobrazuje vytížení procesoru jako zátěž jednoho jádra, zatímco nástroj Xcode Instruments ukazuje souhrnné vytížení všech jader. V případě testovaného iOS zařízení (model iPhone 11 se šesti jádry) tak může maximální zobrazená zátěž dosahovat až 600 %. Rozdíly ve způsobu měření spotřeby energie spočívaly v tom, že u Android aplikace byla spotřeba vyjádřena konkrétní hodnotou vybitého proudu v mikroampérech za časový úsek. Naproti tomu u iOS aplikace nástroj Xcode Energy Report neposkytuje přesné číselné hodnoty, ale místo toho zobrazuje spotřebu energie vizuálně jako relativní zatížení jednotlivých komponent (například CPU, GPU, paměti) ve škálách typu nízká, střední nebo vysoká.

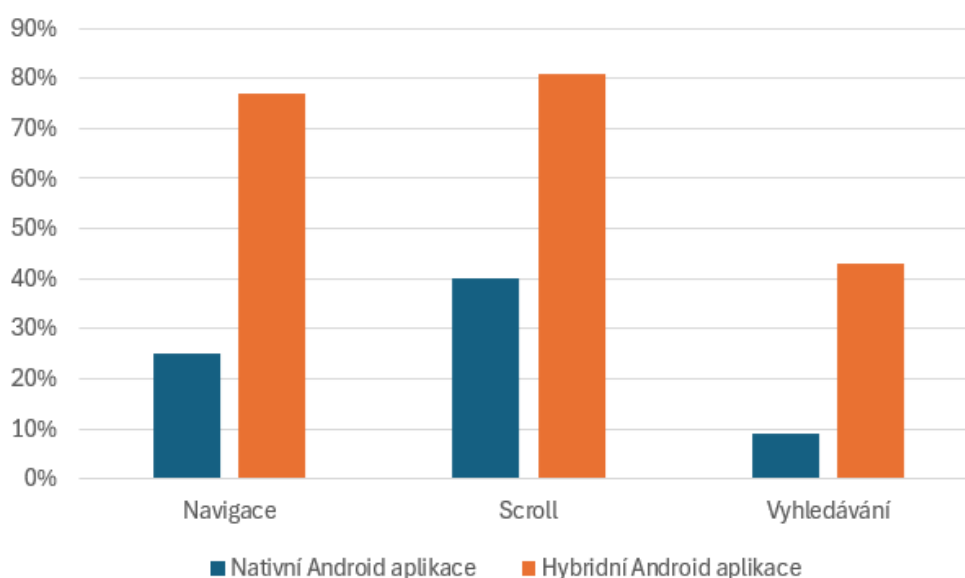
Z výsledků pro Android aplikace je patrné, že nativní verze dosahuje celkově lepších výsledků. Největší rozdíly jsou patrné zejména v oblasti využití operační paměti, neboť Flutter aplikace spotřebovává výrazně více paměti již při samotné inicializaci. Tento nárůst je způsoben přítomností vlastního vykreslovacího nástroje, který Flutter používá pro zajištění jednotného vzhledu a chování na různých platformách. Flutter nástroj zajišťuje samostatné vykreslování uživatelského rozhraní bez využití nativních komponent systému, což přináší vyšší nároky na paměť. Přehled využití paměti v jednotlivých testovacích scénářích je uveden v tabulce 8.8. Údaje o využití paměti RAM byly získány zaznamenáním nejnižší a nejvyšší hodnoty během celého průběhu daného scénáře.

Tabulka 8.8 Použití paměti RAM v Android aplikaci

	1. scénář [MB]		2. scénář [MB]		3. scénář [MB]	
	Nativní	Hybridní	Nativní	Hybridní	Nativní	Hybridní
Min RAM	85	226	201	241	330	550
Max RAM	114	240	232	286	444	653

V CPU zátěži pro Android aplikaci také byly značné rozdíly. CPU ve flutter aplikaci bylo znatelně více vytíženo. Ovšem jedním společným prvkem napříč aplikacemi bylo to, že finální křivka použití byla více méně stejná a klesala a vzrostla ve stejných místech. Pro první scénář je toto možné pozorovat na obrázku 8.24 a pro scénář dva na obrázku 8.25. Data v grafu vycházejí z průměrných hodnot. Pro každý test v rámci scénáře bylo provedeno pět samostatných měření, z nichž byl následně vypočten průměr.

Při měření spotřeby energie nebyly zaznamenány výrazné rozdíly. Nativní i hybridní apli-



Obrázek 8.24 Zátěž procesoru pro Android aplikaci pro scénář 1

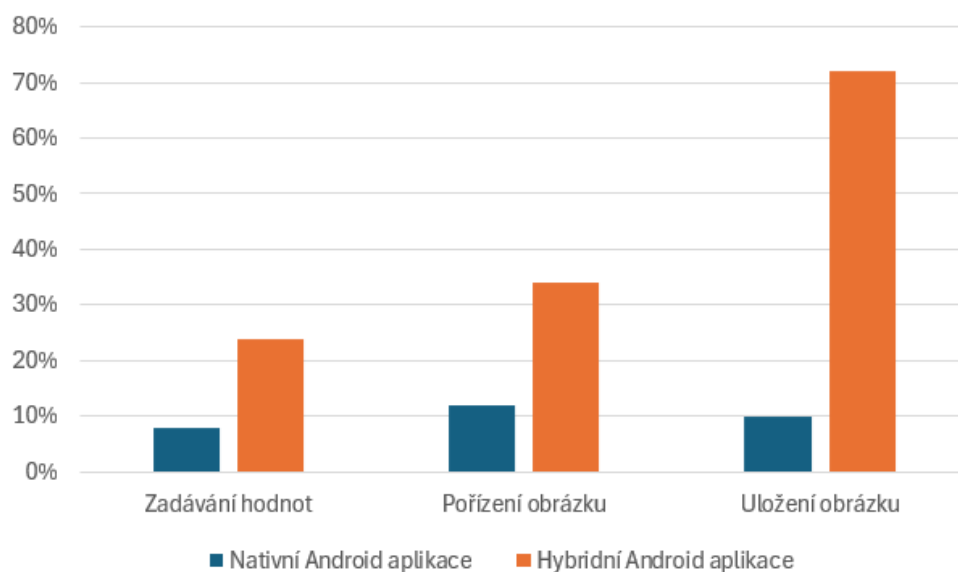
kace vykazovaly v průměru srovnatelnou hodnotu. Ačkoliv je spotřeba energie při třetím scénáři, tedy při práci s rozšířenou realitou, přibližně stejná, vyšší zátěž procesoru byla zaznamenána u hybridní aplikace. Tyto údaje jsou uvedeny v tabulce 12. Každé měření bylo provedeno pětkrát a výsledný průměr byl vypočten z těchto pěti hodnot.

Tabulka 8.9 Porovnání Android aplikací pro scénář 3

	Využití CPU [%]	Spotřeba energie [μA]
Nativní Android	26	1 060
Hybridní Android	42	1 060

Z celkových výsledků vyplývá, že nativní Android aplikace dosahovala lepšího výkonu než její hybridní varianta, a to ve všech testovaných případech. Nejvýraznější rozdíly byly patrné zejména ve využití procesoru, kde hybridní aplikace vykazovala několikanásobně vyšší zátěž. Vyšší spotřeba operační paměti u hybridní aplikace je dána přítomností Flutter enginu, i když v některých scénářích se hodnoty částečně vyrovnaly. Překvapivě však spotřeba energie při práci s rozšířenou realitou byla v obou aplikacích srovnatelná.

Využití operační paměti u iOS aplikací ve všech scénářích lze přehledně sledovat v tabulce 8.10. Stejně jako v případě Android aplikací, i zde nativní aplikace vykazuje výrazně nižší spotřebu paměti ve srovnání s hybridní variantou. V rámci třetího testovacího scénáře, který zahrnuje práci s rozšířenou realitou, se spotřeba u obou verzí pohybovala ve stovkách



Obrázek 8.25 Zátěž procesoru pro Android aplikace pro scénář 2

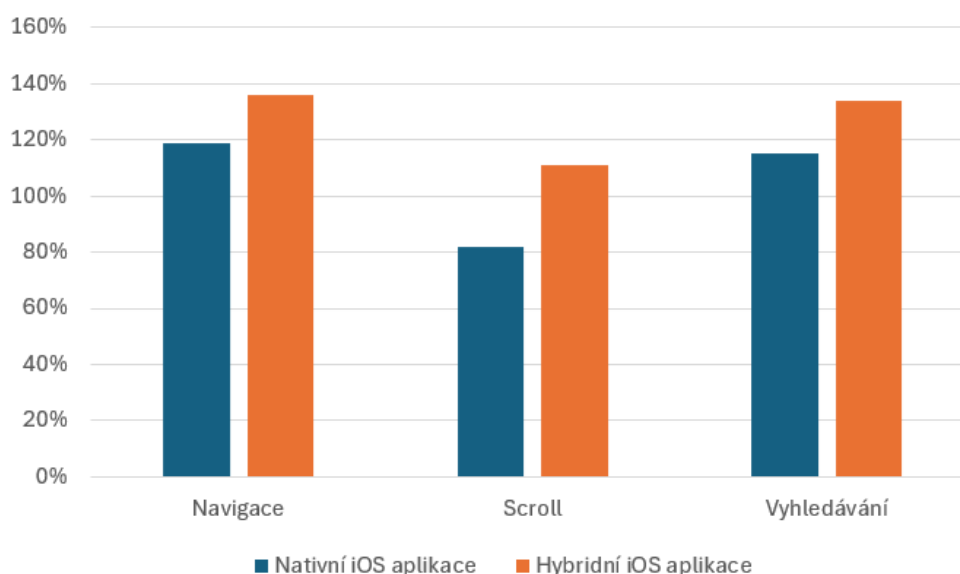
MB. V prvním a druhém scénáři však spotřeba paměti u nativní aplikace nepřesáhla ani 10 MB.

Tabulka 8.10 Použití paměti RAM v iOS aplikaci

	1. scénář [MB]		2. scénář [MB]		3. scénář [MB]	
	Nativní	Hybridní	Nativní	Hybridní	Nativní	Hybridní
Min RAM	8	62	9	68	321	422
Max RAM	29	79	45	97	340	496

Testování využití procesoru u iOS aplikací přineslo překvapivé výsledky. Zatímco u Android aplikací byly rozdíly výrazné a nativní verze vykazovala podstatně nižší zátěž CPU než hybridní, tak u iOS aplikací se situace obrátila. Zatížení procesoru je zde u obou variant téměř totožné. Hybridní aplikace je sice o něco náročnější, ale rozdíl je minimální a její hodnoty jsou velmi blízké těm, které dosahuje nativní verze. Využití CPU pro první scénář je znázorněno na obrázku 8.26 a pro scénář druhý na obrázku 8.27.

Ve třetím scénáři s rozšířenou realitou nebyly podobně jako u nativní aplikace zaznamenány výrazné rozdíly ve vytížení procesoru. Hybridní aplikace vykazuje o něco vyšší zátěž, ale rozdíl není zásadní. Největší rozdíl lze opět pozorovat ve využití operační paměti. Zajímavým zjištěním je spotřeba energie. Jak ukazuje tabulka 8.11, nativní aplikace spotřebovala podstatně více energie než hybridní varianta. Tento rozdíl je však pravděpodobně způsoben odlišným způsobem vykreslení 3D modelu. Zatímco nativní aplikace zobrazila model v co



Obrázek 8.26 Zátěž procesoru pro iOS aplikaci pro scénář 1

největší možné velikosti, hybridní aplikace pracovala s jeho výrazně menším měřítkem. To vedlo u nativní verze nejen k vyšší energetické náročnosti, ale i k viditelnému zahřívání zařízení. Spotřeba energie byla v nástroji Xcode zobrazena pouze pomocí slovních ukazatelů (např. „vysoká“ nebo „velmi vysoká“), a proto byly hodnoty pro účely porovnání přibližně převedeny na procentuální škálu. I podle této aproximace byla energetická náročnost u nativní aplikace téměř dvojnásobná oproti hybridní variantě.

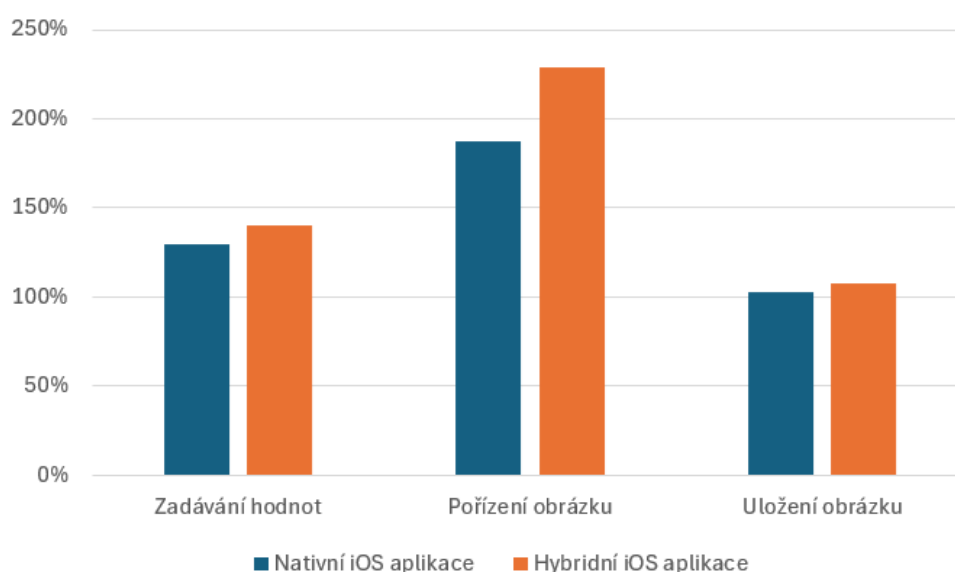
Tabulka 8.11 Porovnání iOS aplikací pro scénář 3

	Využití CPU [%]	Spotřeba energie [%]
Nativní iOS	110	Velmi vysoká = 48
Hybridní iOS	122	Vysoká = 25

Z celkových výsledků vyplývá, že největší rozdíly mezi nativní a hybridní aplikací na platformě iOS se projevily ve využití operační paměti. Ve vytížení procesoru byly obě varianty ve všech testovacích scénářích srovnatelné. Ačkoli byla spotřeba energie při použití rozšířené reality téměř dvojnásobná, tento rozdíl byl pravděpodobně způsoben odlišným zpracováním 3D modelu a implementací rozšířené reality v jednotlivých aplikacích.

8.3 Shrnutí

Měření rychlosti mobilních aplikací bylo prováděno pomocí specializovaných nástrojů dostupných pro jednotlivé platformy. Pro Android aplikace bylo využito prostředí Android



Obrázek 8.27 Zátěž procesoru pro iOS aplikace pro scénář

2

Studio, zatímco pro iOS aplikace sloužil nástroj Xcode Instruments. Hybridní aplikace byla testována prostřednictvím Flutter DevTools, které nabízí základní přehled o výkonnosti. Pro podrobnější měření výkonu (např. využití CPU, paměti nebo energie) však bylo nutné použít nativní nástroje dané platformy, protože Flutter DevTools tyto možnosti neposkytuje s dostatečnou přesností.

Měření rychlosti pro Android aplikace bylo provedeno na emulátoru, který simuloval zařízení Google Pixel 6 s verzí API 35. Pro měření výkonu, zejména v souvislosti s rozšířenou realitou a spotřebou baterie, bylo však nutné využít fyzické zařízení. K tomuto účelu posloužil telefon Samsung A34 s nejnovější dostupnou verzí systému. Testování iOS aplikace bylo ve všech případech prováděno na fyzickém zařízení, jelikož simulátor není schopen plně replikovat hardwarové vlastnosti, jako je práce s fotoaparátem či senzory. Konkrétně se jednalo o zařízení iPhone 11 s operačním systémem iOS 18.

Při měření rychlosti dosahovaly nativní aplikace výrazně lepších výsledků než jejich hybridní protějšky. V některých případech však hybridní verze nabídlý srovnatelný výkon, a výjimečně dokonce překonaly nativní řešení. Nejhorší výsledky vykazovala hybridní Android aplikace, zatímco nejlépe si vedla nativní iOS aplikace. Zajímavostí je, že hybridní iOS aplikace v několika scénářích překonala i nativní Android verzi, což lze přičíst především lepší hardwarové optimalizaci zařízení se systémem iOS.

Měření výkonu bylo rozděleno zvlášť pro platformy iOS a Android. Ve všech testovacích scénářích dosahovaly nativní aplikace lepších výsledků. Hybridní aplikace vykazovaly výrazně vyšší spotřebu operační paměti, což je způsobeno přítomností Flutter enginu. Nativní Android aplikace zatěžovala procesor znatelně méně než její hybridní varianta, avšak spotřeba energie byla v obou případech téměř totožná. Na iOS zařízení byla zátěž procesoru u nativní i hybridní aplikace téměř identická, stejně jako spotřeba energie během vykreslování 3D modelu. Pokud bychom měli porovnat výkon mezi platformami, lze říci, že iOS aplikace obecně spotřebovávají méně paměti. V ostatních sledovaných parametrech však nelze jednoznačně určit jasného vítěze.

Z prováděného měření lze tedy usoudit, že nativní aplikace obecně dosahují lepších výsledků než jejich hybridní protějšky. Rozdíly však nejsou vždy zásadní a v případech, kdy je hybridní aplikace pečlivě optimalizována, může být její vývoj výrazně výhodnější, zejména s ohledem na nižší náklady a možnost cílit na více platforem z jediného kódu.

ZÁVĚR

Tato diplomová práce se zabývala srovnáním hybridního a nativního přístupu k vývoji mobilních aplikací, a to jak z hlediska implementace, tak i výkonnosti. V rámci praktické části byly vyvinuty tři identické aplikace. Dvě nativní pro Android (v Kotlinu) a iOS (ve Swiftu), a jedna hybridní pomocí Flutteru. Byly navrženy scénáře testování zaměřené na typické funkcionality, jako je načítání dat, práce s databází, pořizování fotografií, práce s geolokací a rozšířenou realitou. Aplikace byly následně podrobeny měření rychlosti, výkonu (CPU, paměť), spotřeby energie a plynulosti UI. Výsledky byly získávány pomocí nástrojů jako Android Studio Profiler, Xcode Instruments a Flutter DevTools.

Z výsledků testování vyplynulo, že nativní aplikace dosahují v průměru lepší výkonnosti než hybridní řešení, zejména v oblasti využití paměti a procesoru. Výrazný rozdíl byl patrný především u Android zařízení, kde hybridní aplikace vykazovala vyšší zátěž na systémové prostředky. Naopak iOS aplikace jak nativní, tak hybridní si vedly výkonnostně podobně, což lze přičíst vyšší míře optimalizace v ekosystému Apple.

Při testování rychlosti dosahovaly nativní aplikace výrazně lepších výsledků než jejich hybridní protějšky. Největší rozdíly byly patrné při vykreslování UI prvků a rasterizaci, kde si Flutter engine vedl nejhůře. Z celkového srovnání vyplývá, že hybridní Android aplikace vykazovala nejnižší výkon, zatímco nativní iOS aplikace byla nejrychlejší a nejstabilnější.

Vývoj aplikací se značně lišil také z implementačního hlediska. U základních prvků, jako je práce se seznamem nebo formulářem, byla implementace v nativním i hybridním prostředí poměrně jednoduchá a srovnatelná. Největší rozdíly se projevily při práci s úložištěm, kamerou a zejména při integraci rozšířené reality. Práce s AR byla v hybridním prostředí nejkomplexnější a bylo nutné najít vhodné knihovny, které jsou nejen moderní a dobře zdokumentované, ale zároveň podporují více platforem. Ačkoliv hybridní aplikace nakonec podporovala i zařízení se starší verzí systému, její funkčnost byla výrazně závislá na dostupnosti a kvalitě knihoven. Ty často nejsou tak rychle aktualizovány jako oficiální nástroje pro nativní vývoj, který navíc v některých případech nevyžaduje vůbec žádné externí knihovny.

Přestože hybridní přístup přináší výhody v podobě jednodušší správy kódu a nižších nákladů, u složitějších aplikací, které vyžadují přístup k hardwaru nebo vyšší výkonnost, je vhodnější volbou vývoj nativní. Hybridní řešení však i nadále představuje vhodný kompro-

mis, pokud je správně optimalizováno a cílí na méně náročné scénáře.

Tato práce však také ukazuje, že volba mezi nativním a hybridním vývojem není univerzální a závisí na konkrétním typu aplikace, cílové platformě a požadavcích na výkon. U složitějších scénářů, jako je práce s kamerou nebo AR, zůstává nativní přístup stále jistější volbou.

V praxi tak nelze jednoznačně říct, že jeden přístup je „lepší“. Záleží na prioritách konkrétního projektu, zda je cílem maximální výkon, nebo efektivita vývoje. S ohledem na neustálý vývoj nástrojů a frameworků lze očekávat, že se rozdíly budou nadále stírat a hybridní nástroje budou čím dál schopnější i v náročnějších scénářích.

SEZNAM POUŽITÉ LITERATURY

- [1] SHAH, Disha, 2024. *Android vs iOS Development: Which Platform is the Best for Mobile App?* Online. Dostupné z: <https://radixweb.com/blog/android-vs-ios>. [cit. 2025-02-16].
- [2] KHAN, Amir Ahmed, 2024. *How To Publish Your App On Apple App Store And Google Play Store In 2024?* Online. Dostupné z: <https://magenative.com/blog/how-to-publish-your-app-on-apple-app-store-and-google-play-store/>. [cit. 2025-02-16].
- [3] IBM, 2025. *What is Android development?* Online. Dostupné z: <https://www.ibm.com/think/topics/android-development>. [cit. 2025-04-03].
- [4] TOPPR, 2020. *Mobile Operating System*. Online. Dostupné z: <https://www.toppr.com/guides/computer-science/computer-fundamentals/operating-system/mobile-operating-system/>. [cit. 2025-04-03].
- [5] BASUMALLICK, Chiradeep, 2024. *What Is Android OS? History, Features, Versions, and Benefits*. Online. Dostupné z: <https://www.spiceworks.com/tech/tech-general/articles/android-os/>. [cit. 2025-04-03].
- [6] IBM, 2025. *What is iOS app development?* Online. Dostupné z: <https://www.ibm.com/think/topics/ios-app-development>. [cit. 2025-04-03].
- [7] NETGURU, 2025. *iOS Operating System: Mobile Development Explained*. Online. Dostupné z: <https://www.netguru.com/glossary/ios-development>. [cit. 2025-04-03].
- [8] HONOR, 2024. *How Many Operating Systems Are There for Smartphones?* Online. Dostupné z: <https://www.honor.com/uk/blog/how-many-operating-systems-are-there-for-smartphones/?srsltid=AfmBOopSwmNtd54g1DRABcC7c4oRwgAeAD5TR9hqmbzwm1PXciNhFV3D>. [cit. 2025-04-04].
- [9] MONUS, Anna, 2023. *The 2023 guide to native app development*. Online. Dostupné z: <https://raygun.com/blog/native-app-development/>. [cit. 2025-02-02].

- [10] CLOUDINARY, 2025. *Native Mobile App Development*. Online. Dostupné z: <https://cloudinary.com/guides/front-end-development/native-mobile-app-development>. [cit. 2025-02-02].
- [11] CODEEPSILON, 2022. *Difference between Hybrid and Cross-Platform apps*. Online. Dostupné z: <https://www.codeepsilon.com/difference-between-hybrid-and-cross-platform-apps/>. [cit. 2025-02-15].
- [12] AWATI, ARahul, 2023. *cross-platform mobile development*. Online. Dostupné z: <https://www.techtarget.com/searchmobilecomputing/definition/cross-platform-mobile-development>. [cit. 2025-02-15].
- [13] APPSFLYER, 2023. *Hybrid app*. Online. Dostupné z: <https://www.appsflyer.com/glossary/hybrid-app/>. [cit. 2025-02-15].
- [14] GRIFFITH, Chris, 2024. *What is Hybrid Mobile App Development?* Online. Dostupné z: <https://ionic.io/resources/articles/what-is-hybrid-app-development>. [cit. 2025-02-15].
- [15] SOFTUDE, 2024. *Xamarin The Ultimate Cross Platform Mobile App Development*. Online. Dostupné z: <https://www.softude.com/blog/xamarin-the-ultimate-cross-platform-mobile-app-development>. [cit. 2025-02-15].
- [16] JSCRAMBLER, 2022. *12 Frameworks for Hybrid Mobile Apps*. Online. Dostupné z: <https://jscrambler.com/blog/12-frameworks-for-mobile-hybrid-apps>. [cit. 2025-02-15].
- [17] MAYANI, Paresh, 2025. *Flutter for Hybrid Apps: Why Flutter is the Best Platform to Make Hybrid Apps?* Online. Dostupné z: <https://solguruz.com/blog/flutter-for-hybrid-apps/>. [cit. 2025-02-15].
- [18] FLUTTER, 2018. *Flutter*. Online. Dostupné z: <https://flutter.dev/>. [cit. 2025-02-15].
- [19] AHMED, Asad Jibran, 2024. *Hybrid Power: Flutter Advantages and Benefits*. Online. Dostupné z: <https://www.toptal.com/flutter/hybrid-power-flutter-advantages>. [cit. 2025-02-15].

- [20] JETBRAINS, 2025. *The Six Most Popular Cross-Platform App Development Frameworks*. Online. Dostupné z: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html#popular-cross-platform-app-development-frameworks>. [cit. 2025-04-12].
- [21] ADJUST, 2025. *What is augmented reality (AR)?* Online. Dostupné z: <https://www.adjust.com/glossary/augmented-reality/>. [cit. 2025-03-15].
- [22] SCANDIT, 2023. *What is Mobile Augmented Reality?* Online. Dostupné z: <https://www.scandit.com/resources/guides/what-is-mobile-augmented-reality/>. [cit. 2025-03-15].
- [23] SHARMA, Piyush, 2025. *The Ultimate Guide to Mobile App Optimization (2025)*. Online. Dostupné z: <https://vwo.com/blog/mobile-app-optimization/>. [cit. 2025-04-12].
- [24] SOLUTIONS, 2023. *Mobile App Performance Optimization: Best Practices and Tips*. Online. Dostupné z: <https://medium.com/@blog.iroidsolutions/mobile-app-performance-optimization-best-practices-and-tips-01eea1d4834e>. [cit. 2025-04-12].
- [25] LONGE, Tope, 2025. *How to improve mobile app performance*. Online. Dostupné z: <https://uxcam.com/blog/how-to-improve-mobile-app-performance/>. [cit. 2025-04-12].
- [26] RAUTHAN, Mansi, 2024. *A Comprehensive Guide to Optimizing App Performance by Reducing CPU Usage Spike*. Online. Dostupné z: <https://www.headspin.io/blog/how-to-improve-app-performance-by-fixing-cpu-usage-spike>. [cit. 2025-04-12].
- [27] CRUDU, Ana, 2024. *Strategies for optimizing app performance on low-memory devices*. Online. Dostupné z: <https://moldstud.com/articles/p-strategies-for-optimizing-app-performance-on-low-memory-devices>. [cit. 2025-04-12].
- [28] INTERACTIVE, Sidekick, 2025. *Best Practices for Reducing App Battery Drain*. Online. Dostupné z: <https://www.sidekickinteractive.com/uncategorized/best-practices-for-reducing-app-battery-drain/>. [cit. 2025-04-12].

- [29] QALIFIED, 2024. *Step by step how to test mobile applications.* Online. Dostupné z: https://qalified.com/blog/how-to-test-mobile-applications/?fbclid=IwY2xjawJ0kGFleHRuA2FlbQIxMAABHRrWA-4UhZcc1NcC_9qLqxXrvntvfBsjWwCD8WLrSWgDu6nKEUubEKoOTQ_aem_VQgw4smlpDApUoFAYCFt6A. [cit. 2025-03-24].
- [30] PARKER, Sandra, 2024. *A Complete Guide to Mobile App Testing: Terms, Phases, Costs, & More.* Online. Dostupné z: <https://sandra-parker.medium.com/a-complete-guide-to-mobile-app-testing-terms-phases-costs-more-3c1bc787a26b>. [cit. 2025-03-25].
- [31] TESTLIO, 2024. *Mobile Application Testing Process Explained.* Online. Dostupné z: https://testlio.com/blog/step-step-mobile-application-testing-process/?fbclid=IwY2xjawJ0kJhleHRuA2FlbQIxMAABHTPZzpXXsoD1t-KedixFAZrtAgfQN8oljJSKPABbJ1baOUizN3NojHstKw_aem_aw2ZftP7l9fBVpD71lmdvA. [cit. 2025-03-24].
- [32] IYENGAR, Prathitha, 2024. *A Guide on Automated Mobile App Performance Testing (iOS and Android).* Online. Dostupné z: https://www.headspin.io/blog/mobile-application-performance-testing?fbclid=IwY2xjawJRMJtleHRuA2FlbQIxMAABHY9_F4o5z99MGoLuBdbrFAxLa59ORk_ncR0PGJ90h9S2X0p1HKrlNywBbA_aem_ZZ564P5DxTAmv6qc6PH-bQ. [cit. 2025-03-26].
- [33] TARASOV, Gleb, 2024. *Measuring mobile apps performance in production.* Online. Dostupné z: <https://medium.com/booking-com-development/measuring-mobile-apps-performance-in-production-726e7e84072f>. [cit. 2025-03-27].
- [34] COLLINS, Tom, 2024. *Compare Android vs iOS Mobile App Testing.* Online. Dostupné z: https://www.browserstack.com/guide/difference-between-android-and-ios-testing?fbclid=IwY2xjawJRJ9xleHRuA2FlbQIxMAABHaHXbPau-8f9wHh-iiGjHMNIeolBM1Y14CcCMypWh2ltjELDsyDkcAXquw_aem_D_GU6ygFjEzPMqEJraTrGg#:~:text=Difference%20between%20Android%20and%20iOS%20Application%20testing,-Testing%20for%20Android&text=Android%20testing%20is%20challenging%20due,for%20app%20approval%20and%20security. [cit. 2025-03-26].

- [35] DEVELOPERS, Android, 2025. *Overview of measuring app performance*. Online. Dostupné z: <https://developer.android.com/topic/performance/measuring-performance>. [cit. 2025-03-27].
- [36] DEVELOPERS, Android, 2025. *Profile your app performance*. Online. Dostupné z: <https://developer.android.com/studio/profile>. [cit. 2025-03-27].
- [37] PFLB, 2022. *Android Apps Performance and Load Testing: Tips and Tricks*. Online. Dostupné z: <https://pflb.us/blog/android-apps-performance-and-load-testing/>. [cit. 2025-03-27].
- [38] LEE, Antoine, 2019. *Xcode Instruments usage to improve app performance*. Online. Dostupné z: <https://www.avanderlee.com/debugging/xcode-instruments-time-profiler/>. [cit. 2025-03-27].
- [39] DOCUMENTATION, Apple Developer, 2025. *Writing and running performance tests*. Online. Dostupné z: <https://developer.apple.com/documentation/xcode/writing-and-running-performance-tests>. [cit. 2025-03-27].
- [40] KATALON, 2025. *Top 10 Automated Mobile Testing Tools*. Online. Dostupné z: <https://katalon.com/resources-center/blog/mobile-testing-tools>. [cit. 2025-04-13].
- [41] COUNTRIES, REST, 2025. *REST Countries API*. Online. Dostupné z: <https://restcountries.com/>. [cit. 2025-04-20].

SEZNAM OBRÁZKŮ

Obrázek 1.1	Použití mobilních OS v čase	18
Obrázek 2.1	Cíl hybridní aplikace	21
Obrázek 2.2	Nárůst popularity frameworku Flutter	22
Obrázek 3.1	Typy rozšířené reality	27
Obrázek 3.2	Použití rozšířené reality pro skenování zásilek	28
Obrázek 4.1	Monitorování návštěvnosti a pádu aplikace	31
Obrázek 4.2	Monitorování UI aplikace	36
Obrázek 5.1	Překážky v testování	38
Obrázek 5.2	Cold start aplikace	43
Obrázek 5.3	TTI a TTFR	43
Obrázek 6.1	Android emulátor	45
Obrázek 6.2	iOS simulátor	46
Obrázek 6.3	Android studio profiler	47
Obrázek 6.4	XCode Instruments	48
Obrázek 7.1	Domovská stránka aplikace (iOS, Android, Flutter)	52
Obrázek 7.2	Chybová hláška aplikace (iOS, Android, Flutter)	53
Obrázek 7.3	Seznam států v aplikaci (iOS, Android, Flutter)	54
Obrázek 7.4	Detail stránky státu (iOS, Android, Flutter)	54
Obrázek 7.5	Sumarizační obrazovka (iOS, Android, Flutter)	55
Obrázek 7.6	Seznam obrázků v aplikaci (iOS, Android, Flutter)	56
Obrázek 7.7	Vytváření obrázků v aplikaci (iOS, Android, Flutter)	56
Obrázek 7.8	Použití rozšířené reality v aplikaci	57
Obrázek 8.1	Měření spuštění aplikace v Android Studiu	78
Obrázek 8.2	Měření spuštění aplikace v Android Studiu	79
Obrázek 8.3	Měření spuštění aplikace v Android Studiu	80
Obrázek 8.4	Nástroj Core Animation pro měření rychlosti na iOS	82
Obrázek 8.5	Měření rychlosti ve Flutter DevTools	84
Obrázek 8.6	Výsledky měření startu aplikace	87
Obrázek 8.7	Výsledky měření rychlosti funkcí aplikace	88
Obrázek 8.8	Výsledky měření vykreslování aplikace pro stránku Countries	88

Obrázek 8.9	Výsledky měření vykreslování aplikace pro stránku ImageEdit	89
Obrázek 8.10	Výsledky měření snímků aplikace pro stránku Countries	90
Obrázek 8.11	Výsledky měření snímků aplikace pro stránku ImageEdit	91
Obrázek 8.12	Měření výkonu nativní android aplikace pro scénář 1	93
Obrázek 8.13	Měření výkonu nativní android aplikace pro scénář 2	94
Obrázek 8.14	Měření výkonu nativní android aplikace pro scénář 3	94
Obrázek 8.15	Měření výkonu nativní iOS aplikace pro scénář 1	95
Obrázek 8.16	Měření výkonu nativní iOS aplikace pro scénář 2	96
Obrázek 8.17	Měření výkonu nativní iOS aplikace pro scénář 3	96
Obrázek 8.18	Měření výkonu hybridní Android aplikace pro scénář 1	97
Obrázek 8.19	Měření výkonu hybridní Android aplikace pro scénář 2	98
Obrázek 8.20	Měření výkonu hybridní Android aplikace pro scénář 3	98
Obrázek 8.21	Měření výkonu hybridní iOS aplikace pro scénář 1	98
Obrázek 8.22	Měření výkonu hybridní iOS aplikace pro scénář 2	99
Obrázek 8.23	Měření výkonu hybridní iOS aplikace pro scénář 3	99
Obrázek 8.24	Zátěž procesoru pro Android aplikaci pro scénář 1	101
Obrázek 8.25	Zátěž procesoru pro Android aplikace pro scénář 2	102
Obrázek 8.26	Zátěž procesoru pro iOS aplikaci pro scénář 1	103
Obrázek 8.27	Zátěž procesoru pro iOS aplikace pro scénář 2	104

SEZNAM TABULEK

Tabulka 8.1	Naměřené hodnoty startu aplikace pro hybridní aplikace.....	84
Tabulka 8.2	Průměrné hodnoty vykreslování v hybridních aplikacích na stránce Countries	84
Tabulka 8.3	Průměrné hodnoty vykreslování v hybridních aplikacích na stránce ImageEdit.....	85
Tabulka 8.4	Průměrné hodnoty snímků v hybridních aplikacích na stránce Countries	85
Tabulka 8.5	Průměrné hodnoty snímků v hybridních aplikacích na stránce ImageEdit	86
Tabulka 8.6	Průměrné hodnoty volání v hybridních aplikacích na stránce Countries .	86
Tabulka 8.7	Průměrné hodnoty volání v hybridních aplikacích na stránce ImageEdit	87
Tabulka 8.8	Použití paměti RAM v Android aplikaci	100
Tabulka 8.9	Porovnání Android aplikací pro scénář 3.....	101
Tabulka 8.10	Použití paměti RAM v iOS aplikaci	102
Tabulka 8.11	Porovnání iOS aplikací pro scénář 3	103

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

UI	User Interface
UX	User Experience
XAML	Extensible Application Markup Language
MVC	Model–View–Controller
API	Application Programming Interface
FPS	Frames Per Second
JSON	JavaScript Object Notation
WiFi	Wireless Local Area Network
AR	Augmented Reality
VR	Virtual Reality
MX	Mixed Reality
EX	Extended Reality
GC	Garbage Collector
GPU	Graphics processing unit
HTTPS	Hypertext Transfer Protocol Secure)
QA	Quality Assurance
KPI	Key Performance Indicators
DNS	Domain Name System
TTFB	Time to first byte
TTI	Time to interactivity
TTFR	Time to first render
REST	Representational State Transfer

SEZNAM PŘÍLOH

PŘÍLOHA P I: Obsah elektronických příloh

PŘÍLOHA P I: OBSAH ELEKTRONICKÝCH PŘÍLOH

Přiložená data v elektronické formě obsahují následující soubory a adresáře. Přiložená data byla nahrána službou filesender.cesnet.cz.

- Soubor **reportApp.xlsx** obsahuje naměřené hodnoty aplikací z testování
- Složka **test_appFlutter.tar.gz** obsahuje zdrojový kód ve Flutteru, musí být v tar gzpu, kvůli jeho velikosti
- Složka **TestAppAndroid.zip** obsahuje zdrojový kód nativní aplikace pro Android
- Složka **testappiOS.zip** obsahuje zdrojový kód nativní aplikace pro iOS