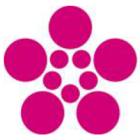


Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Ekonomická fakulta

Katedra aplikované matematiky a informatiky

Bakalářská práce

Vývoj Multiplatformní Cashback Aplikace ve Flutteru

Vypracoval: Tomáš Čeloud

Vedoucí práce: Mgr. Radim Remeš, Ph.D.

České Budějovice 2023

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Ekonomická fakulta

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Tomáš ČELOUD**
Osobní číslo:	**E20051**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Téma práce:	**Vývoj multiplatformní cashback aplikace ve Flutteru**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování

Cílem bakalářské práce je vytvoření aplikace pro podporu nakupování se službou cashback. Výsledná aplikace bude vyvíjena pomocí aplikačního frameworku Flutter a bude použitelná na mobilních zařízeních se systémy iOS a Android, ale též jako webová stránka. Aplikace bude umožňovat registraci uživatele, zobrazit hlavní informace o účtu uživatele formou dashboardu (např. zůstatek, poslední transakce, apod.), možnost platby pomocí QR kódu, základní nastavení uživatelského prostředí (např. volba jazyka, notifikace, apod.) a případně další funkce.

Metodický postup:

1. Studium odborné literatury.
2. Popis použitých technologií pro vývoj aplikace.
3. Návrh, popis vývoje a implementace aplikace.
4. Zhodnocení použitelnosti aplikace pro nasazení v reálném prostředí.
5. Závěr a doporučení.

Rozsah pracovní zprávy:	**40 – 50 stran**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

1. Alessandria, S. (2020). *Flutter Projects: A practical, project-based guide to building real-world cross-platform mobile applications and games.* Packt.
2. Bailey, T., & Biessek, A. (2021). *Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter 2.5 and Dart.* Packt.
3. Cheng, F. (2019). *Flutter Recipes: Mobile Development Solutions for iOS and Android.* Apress.
4. Payne, R. (2019). *Beginning App Development with Flutter: Create Cross-Platform Mobile Apps.* Apress.
5. Windmill, E. (2020). *Flutter in Action.* Manning.

Vedoucí bakalářské práce: **Mgr. Radim Remeš, Ph.D.**
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: **11. ledna 2022**
Termín odevzdání bakalářské práce: **14. dubna 2023**

L.S.

doc. Dr. Ing. Dagmar Škodová Parmová
děkanka

doc. RNDr. Tomáš Mrkvička, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 16. března 2022

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to – v nezkrácené podobě/v úpravě vzniklé vypuštěním vyznačených částí archivovaných Ekonomickou fakultou – elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 7.4.2023 .....

.....

Podpis

Abstrakt

Bakalářská práce popisuje problematiku vývoje mobilních aplikací. Jejím cílem je popsat proces vývoje mobilní aplikace pro podporu nakupování se službou cashback ve frameworku Flutter, která je dostupná na mobilních zařízeních se systémy Android a iOS. V rešeršní části se práce zabývá typy mobilních aplikací a jejich rozdělením, popisem alternativních vývojových možností, vývojovými prostředími a následným seznámením s vybraným frameworkem Flutter a programovacím jazykem Dart. Praktická část představuje popis vývoje samotné aplikace. Jejím cílem je vyvinout uživatelsky příjemnou aplikaci umožňující zobrazit hlavní informace o účtu formou nástěnky, zobrazovat poslední transakce, registraci partnerského účtu a obchodů, ale také možnost platby pomocí QR kódu. Představuje využité knihovny a jejich případné alternativy a popisuje registraci prostřednictvím služby BankId zabezpečenou standardem OpenId.

Klíčová slova: Flutter, Dart, Cashback, Cross-platform

Abstract

The bachelor thesis describes the issue of mobile application development. Its aim is to describe the process of developing a mobile application to support shopping with cashback service in Flutter framework, which is available on Android and iOS mobile devices. The literary research part of the thesis discusses the types of mobile applications and how they are divided, a description of alternative development options, development environments and then an introduction to the selected Flutter framework and the Dart programming language. This work also points to a description of the development of the application itself. Its goal is to develop a user-friendly application that allows to display main account information in the form of a dashboard, display recent transactions, partner account and store registration as well as the possibility to pay using a QR code. It presents the libraries used as well as their possible alternatives and describes registration via the BankId service secured by the OpenId standard.

Keywords: Flutter, Dart, Cashback, Cross-platform

Poděkování

Rád bych touto cestou poděkoval Mgr. Radimu Remešovi, Ph.D., za vedení mé bakalářské práce, konzultace, připomínky a cenné rady, které mi poskytl v průběhu psaní bakalářské práce. Dále bych chtěl poděkovat své rodině a blízkým za podporu v průběhu celého studia.

Obsah

Úvod

Za téma bakalářské práce jsem zvolil vývoj multiplatformní aplikace ve frameworku Flutter. Tento nástroj na vývoj multiplatformních aplikací mě zaujal, a proto jsem chtěl prohloubit své znalosti jeho využitím při vývoji mobilní aplikace. Flutter patří mezi novější vývojové nástroje, ale jeho popularita v IT sektoru se prudce zvyšuje, a to hlavně díky jeho jednoduchosti a rychlosti.

Hlavním cílem bakalářské práce je vytvoření frontendu mobilní cashback aplikace Pointly, která by měla po ověření uživatele prostřednictvím služby BankId umožnit uživateli platit v partnerských obchodech s procentuálním cashbackem. Pointly by měla umožňovat registraci partnerských podúčtů po uvedení kontaktních údajů a identifikačního čísla osoby, registraci obchodů na základě vyplnění adresy a procentuální odměny pro zákazníka, platbu pomocí QR kódu, zobrazení posledních účtenek a v neposlední řadě biometrické ověření uživatele pomocí otisku prstu, pokud je na zařízení dostupné. Aplikace umožní přepnutí režimu mezi světlým a tmavým a výběr jazyka mezi českým a anglickým, a to na základě zařízení nebo dle preferencí uživatele. Pokud by jazyk zařízení nebyl v aplikaci dostupný, výchozím jazykem aplikace bude angličtina.

Práce je rozdělena na teoretickou a praktickou část. Teoretická část se věnuje obecné problematice vývoje a návrhu mobilních aplikací a dostupným alternativám v tomto odvětví. Pozornost bude také věnována samotnému Flutteru a programovacímu jazyku Dart, ve kterém je celá aplikace programována.

Praktická část je poté zaměřena na samotný vývoj aplikace a zdokumentování jeho postupu, včetně všech nově nabytých poznatků ohledně vývoje samotného a věnuje se také struktuře aplikace a využitým knihovnám z oficiálního úložiště balíčků pub.dev.

1 Rozdělení aplikací

1.1 Mobilní aplikace

Mobilní aplikace je software určený pro spuštění na mobilním zařízení jako je chytrý telefon, tablet nebo chytré hodinky. Trend mobilních aplikací se za poslední dekádu velmi posunul a velká část aplikací se z desktopových zařízení a webů přesunula do kapes uživatelů. Nejedná se pouze o mobilní hry, aplikace určené ke vzdělávání a zvýšení produktivity, nakupování, poslechu hudby nebo správě financí ve formě bankovnictví, akciových brokerů nebo peněženek určených pro placení mobilním zařízením či chytrými hodinkami. Mezi nejpoužívanější operační systémy na mobilních zařízeních můžeme zařadit open source Android od společnosti Google a iOS od společnosti Apple.

Rostoucí popularita mobilních aplikací motivuje společnosti k masivní migraci webových a desktopových aplikací do aplikací nativních, multiplatformních nebo hybridních. Některé responzivní webové aplikace jsou sice snadno použitelné na mobilních zařízeních, ale minimálně z uživatelského pohledu jsou mobilní aplikace preferovaná varianta, zejména pokud jsou navrženy tak, aby nevyžadovaly internetové připojení, což od webové aplikace nelze očekávat.

1.2 Nativní aplikace

Nativní aplikace je program vyvinutý pro použití na konkrétní platformě nebo zařízení s konkrétní verzí operačního systému. Aplikace tak má možnost používat hardware a software specifický pro dané zařízení. Nativní aplikace proto mohou poskytovat optimalizovaný výkon a využívat nejnovějších dostupných technologií. Oproti webovým aplikacím nebo mobilním cloudovým aplikacím, které jsou určené pro více systémů.

Tyto aplikace jsou programovány v jazyce používaném na daném zařízení a jeho operačním systému. Nativní aplikace pro iOS jsou většinou vyvíjeny v programovacím jazyce Swift nebo Objective-C, zatímco nativní aplikace pro Android jsou vytvářeny v jazyce Java nebo Kotlin.

Komunikace nativních aplikací s operačním systémem zařízení umožňuje plynulejší chod samotné aplikace a její zvýšenou flexibilitu oproti alternativním typům aplikací. Pokud je aplikace vyvíjena pro více typů zařízení, vývojáři přistupují k tvorbě samostatných verzí pro každé zařízení. (Gillis, 2020)

1.3 Webové aplikace

Webové aplikace jsou opakem nativní aplikace, jedná se o program, který využívá webového prohlížeče a webové technologie k provádění úkolů prostřednictvím internetu. Webové aplikace využívají kombinaci skriptů na straně serveru, které se starají o ukládání a načítání informací a poté skriptů na straně klienta, které informace prezentují koncovému uživateli. Pomocí těchto aplikací je uživatel schopen komunikovat prostřednictvím online formulářů, systémů správy obsahu, nákupních košíků atd.

Tvorba webových aplikací je mnohem jednodušší než tvorba nativních aplikací, protože není nutné tvořit kód speciálně pro každé zařízení. Webová aplikace bude fungovat na všech operačních systémech s přístupem k internetu.

Tyto aplikace jsou obvykle programovány v jazyce podporovaném prohlížečem. Jako je například HTML a Javascript/TypeScript. Tyto jazyky spoléhají na to, že prohlížeč vykreslí spustitelný program. Webové aplikace vyžadují webový server pro zpracování požadavků od uživatele, aplikační server pro provádění požadovaných operací a někdy také databázi pro ukládání informací. Zároveň webové aplikace nemají přístup ke všem hardwarovým funkcím zařízení, na kterém jsou spuštěny, a proto občas nelze využívat fotoaparátu, GPS, biometrického ověření anebo třeba notifikací. (stack-path, 2022)

1.4 Hybridní aplikace

Hybridní aplikace jsou kombinací nativních a webových aplikací. Vnitřní fungování hybridní aplikace je podobné aplikaci webové, ale instaluje se jako nativní aplikace. Tyto aplikace mají přístup k hardwarovým funkcím zařízení, tzn. mohou využívat prostředky jako úložiště, notifikace, biometrické ověření, GPS lokalizaci nebo fotoaparát.

Hybridní aplikace k vývoji využívají webových technologií, mezi které patří HTML, CSS nebo JavaScript/TypeScript. Tyto nástroje však nedokážou samy od sebe pracovat s mobilním operačním systémem, a proto musí být kompilační mezivrstvou převedeny do nativního jazyka Androidu nebo iOS. Aplikace mnohdy spouštějí webovou aplikaci prostřednictvím WebView uvnitř jednotlivých stránek aplikace.

Vývojářům hybridních aplikací stačí vytvořit jeden základ kódu a poté uzpůsobit maličkosti každé platformě. Vývoji hybridních aplikací se může věnovat podstatně méně vývojářů než tvorbě plně nativních aplikací. (Koďousková, 2021)

1.5 Multiplatformní aplikace

Jak už název napovídá, multiplatformní aplikace jsou kombinací aplikací hybridních a nativních. Jedná se o aplikace, při jejichž návrhu není třeba brát ohled na operační systém a jeho verzi. Platformy Android a iOS, které se řadí mezi nejpoužívanější při vývoji mobilních aplikací využívají svých vlastních osobitých prostředí s vlastním jazykem a API.

Při návrhu multiplatformních aplikací stejně jako u hybridních, musejí vývojáři vytvořit pouze jeden kód, poté se ovšem musejí přizpůsobit odlišnostem jednotlivých operačních systému, tzn. že zhruba 80 % celého kódu lze využít pro obě platformy a zbylých 20 % se musí pro jednotlivé operační systémy přizpůsobit. (Koďousková, 2021)

1.6 Rozdíl multiplatformní a hybridní aplikace

Multiplatformní a hybridní aplikace mohou působit na první pohled velmi podobně. Obě se kompilují do nativní aplikace, mohou využívat hardwarových funkcí zařízení a stačí vyvinout jeden kód, který se dá využívat napříč systémy a platformami. Rád bych proto uvedl několik rozdílů, které tyto aplikace odlišují.

Hybridní aplikace kombinují mobilní a webové prvky tzn. kódová základna se buduje pomocí standardních webových technologií jako je již zmíněné HTML, CSS, JavaScript/TypeScript. Po vytvoření této základny se aplikace tzv. zabalí do nativního kontejneru, který využije mobilního WebView. V rámci WebView je poté obsah aplikace vykreslen jako prostý web. Tento kontejner je poté zodpovědný za UX a veškerý přístup k hardwarovým funkcím jako jsou fotoaparát, GPS atd. Přístup k hardwarovým funkcím je však u hybridních aplikací poměrně omezený oproti nativním aplikacím.

Multiplatformní aplikace využívají nativní vykreslovací jádro. Kódová základna napsaná v JavaScriptu se připojuje k nativním komponentám prostřednictvím tzv. můstků. Tímto spojením je zajištěno rozhraní UX blízké nativnímu. Multiplatformní aplikace však nemají vazbu na platformu, jak vypovídá z názvu. Nabízejí bezproblémovou funkčnost, snadnou implementaci a nákladově efektivní vývoj. Přesto nelze očekávat výkon naprosto srovnatelný s nativní aplikací. (Basu, 2020)

2 Přehled vybraných frameworků

V dnešní době existuje nespočet nástrojů používaných pro vývoj mobilních aplikací. Podívejme se na některé alternativy Flutter frameworku, který si samostatně popíšeme v kapitole 5.

2.1 Ionic

Ionic je pravděpodobně jeden z nejznámějších open-source nástrojů v poli vývoje mobilních aplikací. Tento nástroj je postaven na platformě Apache Cordova a Angular, která umožnuje přístup k různým systémovým a hardwarovým funkcím, jako je fotoaparát, geolokace atd. Vývojářům umožňuje vytvářet aplikace pro Android a iOS pomocí webových technologií, jako jsou HTML, CSS, JavaScript/TypeScript. Nejnovější verze Ionic využívají framework Angular 2+. Ekosystém Ionic vývojářům nabízí různé cloudové nástroje pro správu, nasazení a škálování aplikací. (Kukic, 2017)

2.2 React Native

React Native od společnosti Meta Platforms nabízí nástroj pro tvorbu mobilních aplikací pomocí Reactu. Aplikace vytvořené prostřednictvím React Native jsou k nerozeznání od nativních aplikací pro Android a iOS napsaných v Javě nebo Objective-C, což je jeden z důvodů, proč je tolik oblíbeným nástrojem. React Native kombinuje syntaxi Reactu s využitím JavaScriptu, ale umožňuje vývojářům psát i v jazycích Objective-C, Swift nebo Java, pokud je to potřeba ke zkvalitnění výkonu, tzn. vývojáři mohou využívat nativní knihovny. (Kukic, 2017)

2.3 Xamarin

Xamarin vyvinutý společností Microsoft těží z toho, že aplikace vytvořené pomocí tohoto nástroje se kompilují do nativních distribucí pro Android a iOS. Výhodou aplikací vytvářených pomocí Xamarinu je to, že ve srovnání s frameworky založenými na Apache Cordova umožnují využití nativních rozhraní API konkrétní platformy. Aplikace jsou tedy kompilované do nativních aplikací a podle toho se i chovají. Xamarin je psán výhradně prostřednictvím jazyka C#. Nejedná se o řešení, u kterého by byla využitelnost kódu napříč platformami úplná, ale je možné dosáhnout vysoké úrovně sdíleného kódu. Pro platformy Android a iOS je však více než pravděpodobné, že se specificky orientovanému kódu nevyhnete.

Dále vývojáři mohou využívat mnoho knihoven .NET, ale není umožněno využití nativních open-source knihoven dostupných pro Android a iOS. Získání přístupu k nejnovějším nativním rozhraním API může chvíli trvat, protože jejich využití může Xamarin nabídnou až poté, co vývojáři Xamarinu implementují možnost jejich využití do frameworku samotného. (Kukic, 2017)

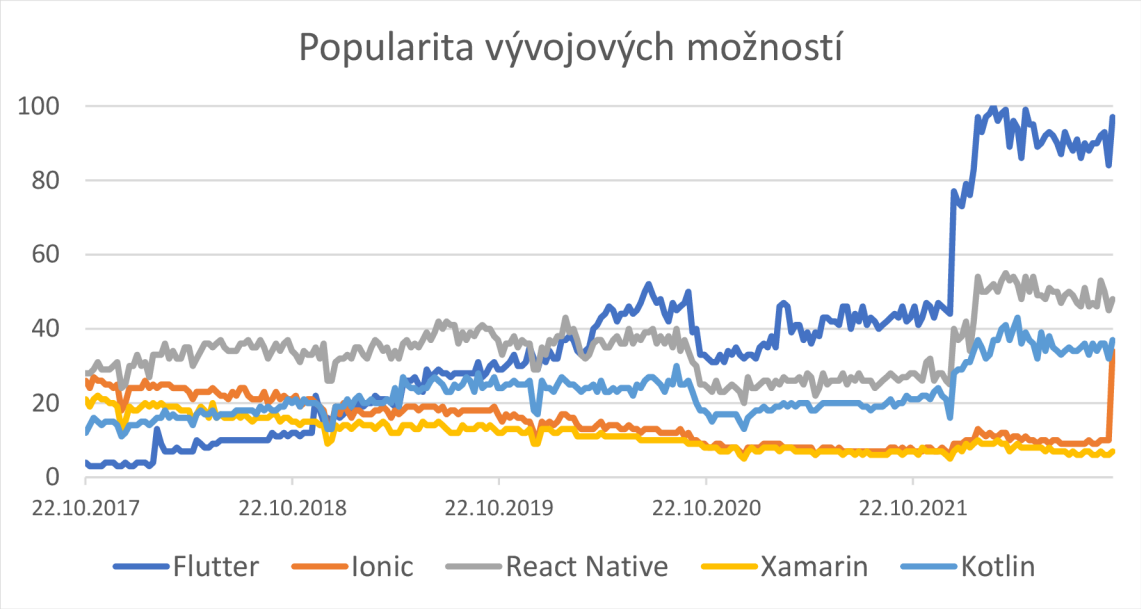
2.4 Kotlin Multiplatform Mobile

Jedná se o sadu nástrojů, která usnadňuje vývoj mobilních aplikací pro více platforem. Tato multiplatformní technologie je založena na jediném kódu a umožňuje vytvářet aplikace, které fungují na zařízeních se systémem Android a iOS. (Miszewski, 2022)

Multiplatformní portabilita je vyřešena tím, že sdílený kód psaný v Kotlinu, je kompilován do bajt kódu Java Virtual Machine pro zařízení Android a prostřednictvím nativních binárních souborů pro iOS. Sestavením aplikace vzniká sdílený modul, který nese koncovku .framework pro systém iOS a .jar pro Android. (What is Kotlin, 2022)

2.5 Popularita vývojových možností

Z dat dostupných na Google Trends je vytvořen graf, který by měl odrážet trendovou křivku jednotlivých programovacích nástrojů. Číslice 0 až 100 na ose Y na grafu znázorňují relativní zájem ve vyhledávání. Hodnota 100 představuje nejvyšší popularitu hledaného nástroje. Hodnota 50 pak tedy znamená, že měl výraz poloviční popularitu.



Obrázek 1 - Popularita vývojových možností (Google Trends, 2022)

3 Vývojová prostředí

3.1 Co je vývojové prostředí

Integrované vývojové prostředí, tzv. IDE, je nástroj pro vývojáře, který sdružuje většinu nebo všechny základní vývojářské nástroje pod jeden framework. Jedná se o softwarový program nebo spojení nástrojů, které je potřeba k psaní a testování vyvíjeného softwaru. IDE se musí skládat přinejmenším z textového editoru, nástroje pro sestavování, nástroje pro automatizaci a ladicího programu. Některá z vývojových prostředí nabízejí možnost instalace pluginů k rozšíření jejich funkcionality, ať už se jedná o komunitní pluginy, anebo pluginy přímo od vývojářů daného prostředí. Rozšíření IDE o různé pluginy může ve velkém usnadnit vývoj v daném prostředí a posunout ho tak na vyšší úroveň.

Před příchodem vývojových prostředí musel vývojář k programování využívat jednoduché textové editory. Napsaný kód potom zkompilovat, zkontrolovat výskyt chyb, vrátit se zpět do editoru, upravit kód a tento proces provádět znovu a znovu. Celý tento proces zabíral mnoho úsilí a času, protože bylo nezbytné přeskakovat z jedné aplikace na druhou.

Nyní stačí použít IDE, které sjednocuje všechny nezbytné vývojářské nástroje pod jedno grafické rozhraní, což zvyšuje produktivitu a výkonnost vývojářů. Vývojová prostředí běžně nabízejí editor zdrojového kódu, ladicí program, kompilátor, našeptávač pro konkrétní jazyk, podporu jazyků, pluginy. Kromě těchto funkcí většina prostředí pro vývoj mobilní aplikací nabízí také emulátory.

Emulátory simulují model funkčního mobilního zařízení uvnitř IDE a umožňují zobrazit vyvíjenou aplikaci a otestovat její chování přímo na obrazovce mobilního zařízení, aniž by bylo potřeba mít přístup k zařízení fyzickému. (Medewar, 2022)

3.2 Android Studio

Android Studio je oficiální integrované vývojové prostředí pro Android. Pro vývojáře nabízí výkonný editor zdrojového kódu a nástroje IntelliJ. Mezi hlavní funkce, které zvyšují produktivitu, patří rychlé emulátory, které mají velmi bohatou funkcionalitu. Nabízejí například možnost testování s využitím fotoaparátu, geolokace nebo biometrického ověření, pokud je to uvnitř aplikace vyžadováno.

Jedná se o jednotné prostředí, na kterém je možné vyvíjet pro všechna dostupná zařízení Android. Samozřejmostí jsou také rozsáhlé ladicí a testovací nástroje. Vývojáři oceňují podporu nástrojů Lint, pro analýzu výkonu, použitelnosti a kompatibility verzí. Android Studio také nabízí rozsáhlou knihovnu pro instalaci rozšíření, která umožnují ještě více optimalizovat vývoj. (Meet Android Studio, 2022)

3.3 Xcode

Xcode se skládá ze sady nástrojů. Nástroje využívají vývojáři k tvorbě aplikací pro platformy iOS. Prostřednictvím Xcode můžete spravovat celý proces pracovního vývoje aplikací, od vytvoření, otestování, optimalizaci až po publikaci do App Store. Samozřejmostí jsou opět multifunkční emulátory sloužící k testování aplikace v simulovaném prostředí, když není k dispozici fyzické zařízení. Simulátory existují pro téměř všechny produkty z dílny Apple od iPhone, iPad až po Apple Watch a zařízení Apple TV. (Xcode, 2022)

3.4 Ostatní vývojová prostředí

Vývojová prostředí většinově nabízejí srovnatelnou funkcionalitu a je víceméně subjektivní, pro které IDE se rozhodnete. Samozřejmě ekosystém Apple je více uzavřený, a proto vývoj musí alespoň z hlediska kompilace a závěrečného testování probíhat na produktech od Applu. V předchozích bodech jsem zmínil dvě hlavní vývojová prostředí pro konkrétní platformy. Níže uvádím další prostředí, která stojí za zmínku.

- Visual Studio
- Eclipse
- JetBrain Rider
- IntelliJ IDEA
- NetBeans
- Cordova

Každé zmíněné vývojové prostředí nabízí něco specifického, ale všechna obsahují stejnou základní funkcionalitu, která se občas dá do určité míry nahradit ve vámi oblíbeném prostředí pomocí dostupných pluginů. Většina z těchto prostředí je bezplatná a volně k dispozici.

4 Dart

4.1 Úvod

Programovací jazyk Dart je vyvíjen společností Google. Jedná se o klientsky optimalizovaný jazyk pro vývoj aplikací. Cílem Dartu je nabídnout co nejproduktivnější programovací jazyk pro cross-platform, umožňující flexibilní spuštění programů bez ohledu na platformu a framework. Dart poskytuje jazyk a prostředí, které pohání všechny Flutter aplikace, ale zároveň podporuje mnoho základních vývojářských úloh, jako je formátování, testování a analýza. (The Language, 2021)

4.2 Bezpečnost datových typů

Jedna z vlastností Dartu je, že se jedná o typově bezpečný jazyk. Používá statickou typovou kontrolu, která zajišťuje, že hodnota proměnné vždy odpovídá jejímu statickému typu. V programování se tento přístup dá nazvat jako „zdravé typování“. Navzdory tomu, že jsou typy povinné, jejich typové anotace jsou kvůli typové inferenci nepovinné.

Systém datových typů v Dartu je flexibilní a umožňuje použití typů v kombinaci s kontrolou za běhu programu. To může být přínosné pro programy, ve kterých zdrojový kód z jakéhokoli důvodu musí být typově dynamický. (The Platforms, 2021)

4.3 Null safety

Další z žádaných vlastností, kterou Dart přinesl, byla mimo jiné kontrola nulových hodnot nebo tzv. null safety, která ve velkém ovlivnila celý Flutter framework a objevila se konkrétně při uvedení Flutteru 2.0. Jedná se o specifikaci, která přináší funkci nazvanou *compile-time null safety*. Cílem této funkce je vynutit inicializaci všech proměnných nějakou konkrétní hodnotou. V případě inicializace a ošetření všech proměnných nedojde k žádné chybě v průběhu programu, která by způsobila výjimku nulových hodnot. (Silas K., 2021)

Tato kontrola nulových hodnot ovšem zároveň umožňuje jejich využití. Pokud však hodláte umožnit datovému typu nést nulovou hodnotu musíte to při zápisu patřičně specifikovat. V jazyce Dart je syntaxe pro tuto specifikaci značena pomocí otazníku v rámci suffixu datového typu.

4.4 Knihovny

Dart nabízí bohatou sadu dostupných knihoven, které poskytují pevnou základnu pro spoustu základních programovacích operací. Mezi tyto knihovny patří například.

- (dart:core) pro integrované typy a základní funkce programu.
- (dart:convert) obsahuje kodéry a dekodéry pro převod mezi různými reprezentacemi dat, jako jsou JSON nebo UTF-8.
- (dart:math) pro práci s matematickými konstantami, pseudonáhodnými čísly a podobně.
- (dart:io) obsahuje podporu soketů, souborů, vstupně-výstupních operací, pro jiné než webové aplikace.
- (dart:async) umožňuje využití asynchronního programování a Futures.

Kromě základních knihoven je dostupné také mnoho API rozhraní. Mezi tato rozhraní patří například škálovatelná knihovna **http**, která slouží k vytváření HTTP požadavků. Za zmínku stojí dále **intl**, který umožňuje jednoduše pracovat s formáty datumů, čísel, ale také třeba s výchozím globálním prostředím atd.

Tisíce balíčků a knihoven s podporou a rozšířením funkcionality přináší komunita a vydavatelé třetích stran, kteří je většinou aktivně udržují. Veškeré sdílené balíčky jsou publikovány s licencí open-source, a proto se na jejich tvorbě může podílet celá komunita. Umožněno je samozřejmě vytvářet požadavky na změny a vylepšení. Mezi tyto balíčky můžeme zařadit například Dart XML, což je odlehčená knihovna pro analýzu, procházení, transformaci a vytváření XML dokumentů. Populární jsou také balíčky umožňující snazší integraci a komunikaci s různými druhy databází, ať už se jedná o SQLite, PostgreSQL, MongoDB nebo jiné.

Pokud je potřeba nějaká rozšířená funkcionalita, kterou Dart v základu nenabízí, existuje reálná šance, že někdo z komunity řešil stejný problém a již na to bude vytvořen balíček, který pomůže k řešení daného problému.

4.5 Kompilace

K dispozici jsou dva druhy kompilace, ve Flutteru se Dart používá jak při vývoji, tak i při finální kompilaci, ale typ kompilace je odlišný.

V režimu vývoje a ladění se jedná o kompilátor nesoucí název just-in-time, zkráceně JIT. Kompilace kódu tímto způsobem se hodí v průběhu vývoje, neboť dochází k průběžné kompilaci, což umožňuje například funkci hot-reaload, kterou naleznete v kapitole 5.6. Aplikace sestavená pomocí kompilátoru JIT je náročnější na úložiště, protože obsahuje veškeré ladicí a profilovací služby, což je pro produkční verzi aplikace nežádoucí.

Druhým typem kompilace, kterou Dart nabízí, je kompilátor ahead-of-time nebo zkráceně AOT. Tento kompilátor funguje tak, že celý kód je zkompilován před samotným spuštěním aplikace, což umožňuje vyšší rychlost provedení a také umožňuje lepší předvídatelnost výkonu oproti jazykům využívajícím just-in-time kompilátor pro distribuční verzi aplikace. V režimu AOT, jsou odstraněny ladicí a profilovací služby a zůstává samotné běhové prostředí (runtime), což znatelně snižuje základní velikost produkční verze aplikace. (Sullivan, 2019)

4.6 Garbage collector

Programovací jazyk Dart je primárně určen pro reaktivní programování, to je umožněno díky jeho schopnosti rychlé alokace paměti pro nové objekty a zároveň rychlé dealokace, kdy se zbavuje nepotřebných objektů pomocí integrovaného garbage collectoru.

Garbage collector je nezbytnou součástí Flutteru, protože tam dochází k vytváření nepřeberného množství objektů. Všechny widgety, které neobsahují stav, tzv. Stateless Widgety, se totiž při změně stavu aplikace celé zničí a dojde k jejich obnovení. Netřeba se však obávat poklesů na rychlosti, garbage collector je uzpůsoben k vytváření a ničení objektů s velmi vysokou frekvencí. Samotný garbage collector je také propojen s Flutter enginem, který collector upozorňuje v moment, kdy uživatel neprovádí žádnou interakci v aplikaci, což je nejlepší čas pro dealokaci paměti, protože uživatel neočekává žádné okamžité výstupy. Během tohoto časového intervalu dochází také k defragmentaci paměti, a to zlepšuje celkový výkon a optimalizaci aplikace. (Sullivan, 2019)

5 Flutter

5.1 Úvod

Flutter je sada nástrojů vytvořená společností Google, pro tvorbu nativně kompilovaných aplikací pro mobilní zařízení, web a desktop na základě jednoho sdíleného zdrojového kódu. Jedná se o open-source projekt, který je dostupný všem, a na kterém se stále aktivně podílí společnost Google a mnoho jiných společností a jednotlivců.

Framework Flutter je navržen pro podporu mobilních aplikací pro systémy Android a iOS, ale zároveň pro interaktivní aplikace, které mohou fungovat na webových stránkách nebo počítači.

Ekosystém balíčků Flutter podporuje širokou škálu hardwaru a umožňuje tak jeho využití. Lze například naplno využívat fotoaparát, GPS, síť nebo interní úložiště. Dále samozřejmě nabízí široké využití služeb, zde se jedná o různé cloudové služby, ověřování identity, reklamy nebo například platby prostřednictvím GooglePay a ApplePay.

5.2 Unikátnost Flutteru

Flutter se odlišuje od většiny ostatních alternativ pro vývoj mobilních aplikací, protože nespoléhá na technologii webového prohlížeče a ani na sadu widgetů dodávaných s každým zařízením. Místo toho Flutter využívá vlastní vysoce výkonný vykreslovací engine.

Další odlišností tohoto frameworku je, že Flutter obsahuje pouze tenkou vrstvu kódu v jazyce C/C++, většina implementovaného systému je vlastní. Jedná se například o animace, kompozice, gesta, widgety atd. Tyto implementované součásti systému jsou vytvořeny prostřednictvím moderního objektově orientovaného jazyka Dart, který je taktéž produktem společnosti Google. To, že většina součástí systému není převzata, ale vytvořena ve vlastním jazyce, dává vývojářům obrovskou kontrolu nad celým systémem. (Introduction, 2017)

5.3 Co Flutter nabízí

- Vysoce optimalizovaný 2D vykreslovací engine s vynikající podporou textu.
- Moderní framework ve stylu React.
- Bohatou sadu widgetů implementujících Android Material Design a styl iOS.
- API pro jednotkové a integrační testy.
- Interop a API pluginy pro připojení k systému a SDK třetích stran.

- Headless test runner pro spouštění testů v systémech Windows, Linux a Mac.
- Dart DevTools pro testování, ladění a profilování aplikace.
- Nástroje příkazového řádku pro tvorbu, sestavování, testování a kompilaci.

5.4 Vývojová prostředí

Flutter poskytuje pluginy pro Android Studio, IntelliJ IDEA a VS Code. V dokumentaci na internetu jsou dostupné veškeré informace potřebné ke konfiguraci editoru a tipy na využití pluginu. (Introduction, 2017)

5.5 Proč Dart

Během počáteční fáze vývoje Flutteru bylo zkoumáno mnoho programovacích jazyků a Dart získal vysoké hodnocení ve všech relevantních kategoriích. Při hodnocení se zvažovaly potřeby autorů frameworku, vývojářů a koncových uživatelů.

Dart podporuje kombinaci dvou pro Flutter důležitých vlastností. Těmi jsou rychlý vývojový cyklus založený na JIT, který umožnuje tzv. hot reload a kompilátor AOT, který emituje efektivní kód ARM pro rychlé spuštění a předvídatelný výkon produkčních nasazení.

Velkou výhodou je také skutečnost, že Dart je produktem společnosti Google, takže jakákoli potřebná vylepšení pro použití ve Flutteru jsou umožněna díky úzké spolupráci s týmem a komunitou Dart. Například v Dartu scházel řetězec nástrojů pro tvorbu nativních binárních souborů, což je klíčové pro dosažení předvídatelného a vysokého výkonu, ale nyní ho jazyk má, protože ho tým Dart vytvořil speciálně pro Flutter podle potřebných požadavků. (Technology, 2017)

5.6 Hot reload

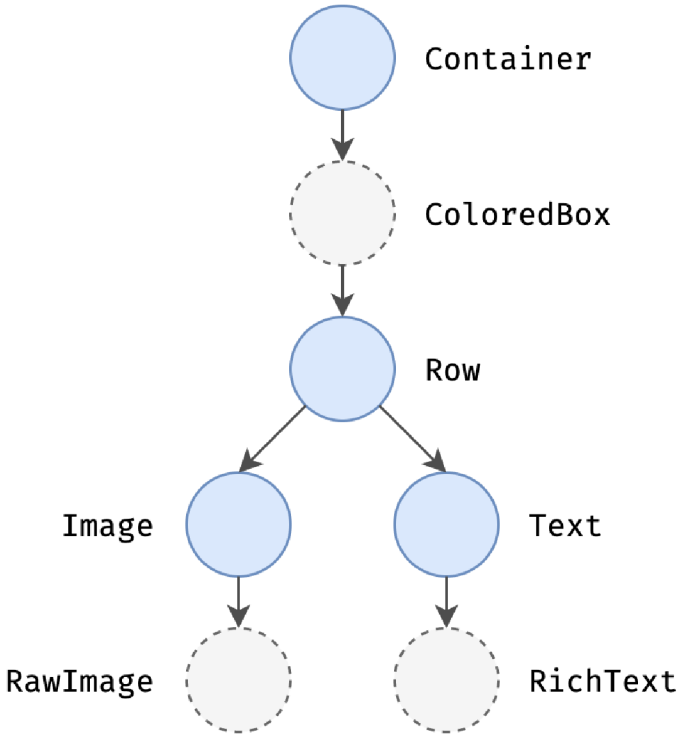
Velkou výhodu při vývoji aplikací přináší funkce rychlého načtení nebo tzv. hot-reload. Tato funkce pomáhá snadnému a rychlému testování aplikací a návrhu uživatelského rozhraní.

Hot reload funguje tak, že se aktualizovaný zdrojový kód vloží do běžícího Dart Virtual Machine. Poté virtuální počítač aktualizuje třídy novými poli a funkcemi a Flutter framework automaticky přestaví strom widgetů, což umožní okamžitě zaznamenat provedené změny. (Hot Reload, 2017)

5.7 Widgety

Flutter klade důraz na widgety jako základní kompoziční jednotku. Widgety jsou základním stavebním kamenem uživatelského rozhraní. Každý widget je neměnnou deklarací části uživatelského rozhraní. Tyto widgety tvoří hierarchii založenou na kompozici. Widgety se do sebe vnořují, takže každý widget má svého rodiče a může přijímat jeho kontext. Tato struktura se nese až ke kořenovému widgetu.

Widgety se obvykle skládají z mnoha dalších jednoúčelových widgetů, které se kombinují a vytvářejí tak celkový dojem uživatelského rozhraní. Vzniká tak rozsáhla kompozice, která by se dala znázornit diagramem. (Flutter Architectural Overview, 2017)



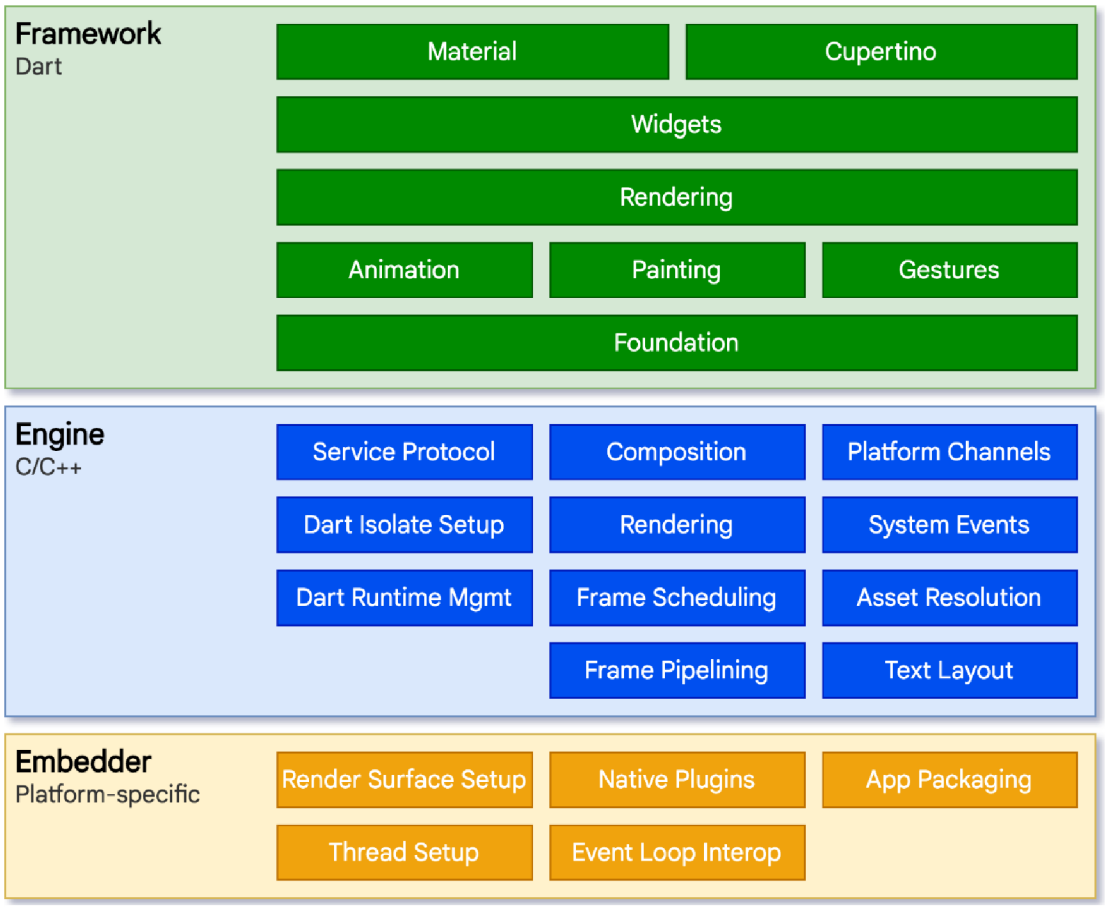
Obrázek 2 - Widget tree (Flutter docs, 2022)

5.7.1 Druhy widgetů

Widgety lze rozdělit na základě jejich stavů na dva typy. Prvním typem je tzv. *StatelessWidge*t neboli bezstavový widget, a druhým typem je *StatefullWidget* neboli stavový widget. Stav widgetu vyjadřuje tzv. logický stav, který ovlivňuje jeho sestavování. Pokud widget stav obsahuje, očekává se, že v průběhu jeho životního cyklu dojde k jeho změně. Flutter nabízí několik přístupů, jak řídit stavový managment aplikace, tomuto tématu se budeme podrobněji věnovat v praktické části.

5.8 Přehled architektury Flutter

Flutter je navržen jako rozšiřitelný vrstvený systém. Existuje jako řada nezávislých knihoven, z nichž každá závisí na základní vrstvě. Žádná z vrstev nemá privilegovaný přístup k vrstvě nižší a každá část frameworku je navržena tak, aby byla volitelná a nahraditelná. (Architectural Layers, 2017)



Obrázek 3 - Architectonic layers (Flutter docs, 2022)

Do základního operačního systému jsou aplikace zabaleny stejně jako aplikace nativní. Embedder je specifický pro platformu a poskytuje tak vstupní bod. Napsán je v jazyce specifickém pro danou platformu. V současné době se jedná o jazyky Java, C++, Objective-C/Objective C++. Jádrem je poté engine, který je z větší části napsán v C++ a podporuje primitiva nezbytná pro podporu všech aplikací Flutter. Poskytuje nízko úrovňovou implementaci základního rozhraní API Flutteru, včetně grafiky, rozvržení textu, souborových a síťových vstupů a výstupů atd. (Architectural Layers, 2017)

6 Tvorba projektu

Předpokladem je, aby byl Flutter nainstalovaný na zařízení, kde se chystá vývoj aplikace. Pro samostatné vytvoření projektu je potřeba využít konzolového rozhraní, tzv. CLI.

6.1 Flutter CLI

Konzolové rozhraní umožňuje vývojáři nebo vývojovému prostředí komunikovat s Flutterem. Pro umožnění komunikace například z Windows shell je nezbytné přidat Flutter mezi proměnné prostředí ve vlastnostech systému a nastavit k němu cestu.

Pokud máme Flutter přidaný a windows shell nám umožňuje využívat jeho příkazů, můžeme se přesunout do složky, kde chceme náš projekt založit a zadáme příkaz **flutter create <název-projektu>**. Po úspěšném vytvoření se zobrazí hláška viz. (Obrázek 4 – Založení projektu)

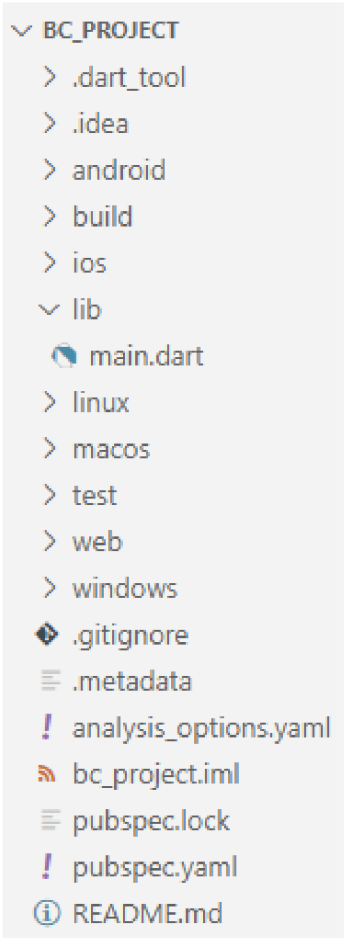
```
D:\Users\Bob\Documents\GitHub>flutter create bc_project
Creating project bc_project...
Running "flutter pub get" in bc_project...
Wrote 127 files.

All done!
In order to run your application, type:

  $ cd bc_project
  $ flutter run

Your application code is in bc_project\lib\main.dart.
```

Obrázek 4 - Založení projektu (vlastní)

Po založení projektu je výhodné otevřít projekt pomocí některého z vývojových prostředí dle vlastní preference. Já jsem si pro vývoj zvolil Visual Studio Code, které mi vyhovuje díky jeho rozšíření, široké komunitě a možnostem pro debugovaní projektu.

6.2 Struktura složek

Nově vygenerovaný projekt obsahuje výchozí strukturu složek (Obrázek 5 – Defaultní struktura). Popíšeme si, co jednotlivé složky obsahují a které z nich budou nezbytné při tvorbě projektu z hlediska programování.



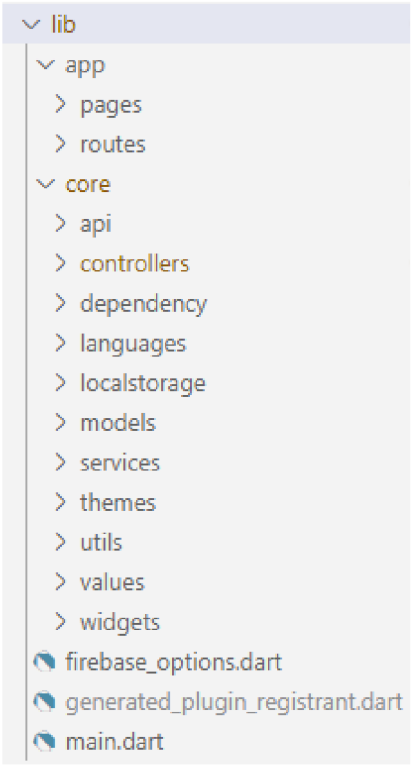
Obrázek 5 - Výchozí struktura složek (vlastní)

- **.dart_tool** – Obsahem této složky jsou soubory používané nástroji jazyku Dart.
- **.idea** – Specifická nastavení související s editorem zdrojového kódu, který je používán k sestavení aktuálního projektu.
- **android** – Uvnitř této složky můžeme najít soubory potřebné pro spuštění aplikace v operačním systému Android.
- **ios** – Stejně jako složka android i zde nalezneme soubory nezbytné pro spuštění aplikace na konkrétní platformě a to iOS.

- **lib** – Složka lib je pro vývojáře tím nejzajímavějším místem z celé struktury. Zde je totiž napsána většina zdrojového kódu aplikace. Tato složka se mnohdy dělí na podsložky, dle uvážení vývojáře. Platí však nějaká pravidla, která je vhodné dodržovat. Mezi tato pravidla patří například rozdělení widgetů, obrazovek, modelů, služeb a utilit do samostatných složek uvnitř adresáře lib.
- **test** – Tento adresář slouží pro vkládání veškerých testovacích souborů.
- **linux, macos, web, windows** – Tyto složky obsahují opět specifický kód pro konkrétní platformy. Nás se tyto složky vůbec netýkají, protože aplikace bude provozuschopná pro zařízení s operačním systémem iOS a Android.
- **.gitignore** – Tento soubor obsahuje veškeré názvy souborů a složek, které z nějakého důvodu nechceme aktualizovat při nahrávání nové verze projektu přes správu verzí Git, o které si řekneme více v následující kapitole.
- **.metadata** – Obsahem tohoto souboru jsou metadata vyžadovaná nástrojem Flutter ke sledování projektu.
- **analysis_options.yaml –** Zde jsou uchovávána pravidla pro Linter, kterému se budeme věnovat v kapitole osmé.
- **pubspec.lock** – V tomto souboru můžeme najít používanou verzi každé součásti Dartu a všech ostatních balíčků používaných Flutterem.
- **pubspec.yaml** – Toto je soubor obsahující konfigurace pro naši aplikaci. Můžeme zde přidávat balíčky, knihovny a ostatní.
- **README.md** – V tomto souboru bývá popis aplikace, může se zde zahrnout vše, co aplikace dělá atd.

6.3 Volba struktury

Pro tvorbu mé aplikace jsem si zvolil následující strukturu *lib* složky (Obrázek 6 – Vlastní struktura). Uvnitř složky *lib*, jsem vytvořil dvě základní podsložky.



Obrázek 6 - Vlastní struktura složek (vlastní)

- **app –** Tato složka obsahuje hlavní prezentační vrstvu aplikace a navigaci.
    - **pages** – Název složky zde jednoznačně vypovídá o jejím obsahu, nalezneme zde totiž veškeré stránky existující uvnitř aplikace.
    - **routes** – Tato složka obsahuje soubor, který definuje veškeré navigační cesty napříč aplikací.
- **core** – Jednoduše řečeno obsahuje součásti aplikace, které většinou tzv. nejsou vidět, ale je zde také pár výjimek.
    - **api –** Zde jsou uchovány soubory komunikující s API mobilních platforem.
    - **controllers** – Tato složka obsahuje soubor definující kontroléry pro textová pole a vstupy.
    - **dependency** – Stejnojmenný soubor uvnitř složky zajišťuje kontrolu internetového připojení pro mobilní zařízení.

- **languages** – Definice překladů a třída umožňující jejich globální změnu.
- **localstorage** – Místní úložiště uchovává veškerá data pro aktuálně přihlášeného uživatele a také jeho preference jako například používaný motiv, jazyk atd.
- **models** – Datové modely a jejich parsery. Jedná se o třídy, které definují konkrétní modely a umožňují je vytvořit z formátu JSON, ve kterém se vracejí z backendu po úspěšném provedení dotazu.
- **services** – Služby komunikující s backendem. Zde jsou definovány veškeré HTTP požadavky, které získávají, nebo upravují data.
- **themes** – Definice všech barev a motivů, ale také třída umožňující přepínaní a upřesňující celkové chování motivů v aplikaci.
- **utils** – V této složce jsou uchovány veškeré formátovače textových vstupů, validátory, formátovač finanční částky a podobně.
- **values** – API tokeny, konstanty a jiné statické hodnoty používané uvnitř aplikace.
- **widgets** – Veškeré widgety, které mají potenciál být znovu využity na vícero místech v aplikaci, jsou uchovány zde. Patří sem například tlačítka, dlaždice, horní lišta aplikace, textový vstup atd.

7 Správa verzí

Při vývoji softwaru je vhodné kontrolovat a držet si přehled o aktuálních verzích, produkčních a vývojových větvích a provedených změnách napříč časem. V případě systému s otevřeným kódem nebo spolupráce při vývoji, je také vidět, kdo dané změny provedl. Díky tomu je poté možné zdrojový kód a veškeré změny před uvolněním do produkční verze softwaru ověřovat a kontrolovat. Systémů pro správu verzí je mnoho, ale nejoblíbenějším z nich je dle mého názoru Git.

7.1 Git

Git je systém pro správu verzí softwaru nebo obecněji pro sledování změn v libovolné sadě soborů. Primární obsluha Gitu probíhá z příkazového řádku. V průběhu let vzniklo mnoho grafických rozhraní, tzv. GUI, pro usnadnění práce při verzování.

7.1.1 Git GUI

Grafická rozhraní pracující s Git obvykle fungují jako desktopové nebo webové aplikace. Umožnují veškeré nezbytné operace pro verzování a velice usnadňují správu verzí pro nové vývojáře, kteří s Gitem nemají zkušenosti a zároveň se ho nechtějí ihned učit.

7.1.2 GitHub

Jako software pro verzování projektu jsem zvolil GitHub Desktop. Jedná se o desktopovou aplikaci, která je napříč komunitou velmi oblíbená. U většiny balíčků vyvíjených pro Flutter a Dart navíc verzování probíhá právě na GitHubu, což umožňuje okamžitý přehled o změnách. Dále GitHub nabízí vytváření issues, tzv. problémů nebo nedostatků. Pokud komunitě něco chybí nebo v balíčku či softwaru něco nefunguje správně, tak je možné evidovat právě issue a komunita nebo samotný vývojář se poté mohou pustit do jeho řešení. Tento přístup zajišťuje rychlé řešení a opravu těchto chyb, nelze však tuto zásluhu připsat GitHubu, tento proces vývoje přinesl open-source neboli software s otevřeným zdrojovým kódem.

8 Linter

Program se může zastavit z mnoha důvodů, ale většině chyb se dá předejít ještě před prvním sestavením. K tomu, aby bylo možné chybám předejít, využívám právě Linter. Jedná se o sadu předdefinovaných pravidel, která se integrují do používaného vývojového prostředí.

Linter dělí pravidla na tři typy. Prvním typem jsou pravidla pro chybová hlášení, tato pravidla předcházejí možným chybám nebo omylům, kterých se může autor dopustit při psaní zdrojového kódu. Tyto chyby bývají funkční a kritické z hlediska kompilace.

Druhým typem jsou pravidla předcházející stylistickým chybám, které jsou odvozeny z příručky stylu jazyku Dart. Pomáhají správnému formátování kódu a v jeho následné čitelnosti. Tyto chyby nejsou pro běh programu kritické, ale je vhodné tato pravidla dodržovat. Při spolupráci v týmu při vývoji softwaru je díky těmto pravidlům dodržen jednotný styl tvorby kódu, což dělá spolupráci o mnoho příjemnější.

Třetím a zároveň posledním typem jsou pravidla předcházející chybám způsobených knihovnami a balíčky z webového úložiště pub.dev. Tato pravidla ošetřují problémy s nastavením různých balíčků a pomáhají tak vyhnout se rozsáhlým chybovým hláškám při kompilaci.

```
always_specify_types: true
prefer_single_quotes: true
always_declare_return_types: true
cancel_subscriptions: true
close_sinks: true
comment_references: true
one_member_abstracts: true
only_throw_errors: true
package_api_docs: true
prefer_final_in_for_each: true
```

Výpis 1 - Linter pravidla (vlastní)

Všechna pravidla jsou definována uvnitř souboru **analysis_options.yaml**, který se nachází v kořenovém adresáři projektu. Tato pravidla (Výpis 1 – Linter pravidla) používám v projektu při vývoji cashback aplikace. Nejdůležitějším pravidlem je však pravidlo na první pozici. Jedná se o pravidlo **always_specify_types**. Právě toto pravidlo předchází největšímu množství kompilačních chyb, protože si vynucuje specifikování datových typů. Jejich definování usnadňuje následnou revizi kódu a zabraňuje

kompilačním chybám kvůli nulovým hodnotám, protože Linter vždy aktivně napoví, že proměnná nemůže obsahovat hodnotu null.

Dalším pravidlem, které můžeme vidět je **prefer_single_quotes**. Toto pravidlo nemá žádný efekt z hlediska funkčnosti, protože se jedná spíše o kosmetický doplněk. Záleží čistě na preferencích autora zdrojového kódu. Toto pravidlo se vždy projeví v moment, kdy v zápisu použijeme špatný tvar uvozovek. Jak bylo popsáno výše, i kdyby pravidlo oznamovalo chybu, funkčně to nevadí a žádné kompilační chyby se neobjeví.

Pravidlo **always_declare_return_types** je žádoucí použít, pokud na zdrojovém kódu spolupracuje více vývojářů, ale je vhodné i pokud je vývojář pouze jeden. Tímto pravidlem si vývojové prostředí všude vynucuje specifikaci návratových typů. Jak jsem již zmínil, toto pravidlo pomáhá při týmovém vývoji, a to hlavně z toho důvodu, že to velmi zlepšuje čitelnost kódu. Specifikací návratových typů stačí jeden pohled na funkci či metodu a hned lze jednoduše poznat, co máme očekávat na výstupu.

Ostatní pravidla není úplně důvod rozebírat. Chtěl jsem však poukázat na to, že využití nástrojů jako je například právě Linter, může ve velkém ovlivnit úroveň zdrojového kódu, který vyvíjíte. Zároveň to může podpořit efektivitu vývoje uvnitř týmů a spoustu chyb při kompilaci, či běhu programu samotného.

9 Knihovny a balíčky

Flutter a Dart nabízí velké množství doplňků, které lze najít na stránkách pub.dev, jak už bylo v práci mnohokrát zmíněno. Tyto doplňky slouží k tomu, aby vývojářům usnadnily práci a nemuseli v mnoha případech znovu vymýšlet něco, co již existuje. Většina knihoven a balíčků je udržována a upravovaná autory nebo komunitou, která je využívá. Nicméně je více než užitečné si před použitím některého z balíčků udělat jeho analýzu, a to obzvlášť, pokud se jedná o balíček využívaný ve větší míře napříč aplikací.

9.1 Hodnocení

Při výběru knihovny pro state management, která se bude používat napříč celou aplikací, jsem se rozhodoval mezi knihovnami GetX, Bloc a Riverpod. Před výběrem knihovny jsem však musel analyzovat možnosti které mám, abych mohl vybrat správně.

	Dokumentovaných API	Prvků API	Dokumentovaných [%]
GetX	744	2454	30,32
Bloc	70	**80**	87,50
Riverpod	557	617	**90,28**

Tabulka 1 - Hodnocení dokumentace (vlastní)

Hlavním ukazatelem toho, zda knihovnu zahrnu ve svém projektu je míra dokumentovaných API. Pokud knihovna obsahuje dokumentaci pouze v takové míře, aby dostala maximální počet bodů na stránkách pub.dev, je to pro mě informace, že bych se dané knihovně měl pravděpodobně vyhnout a zkusit najít alternativu.

Druhá část tohoto ukazatele je rozsah dané knihovny. Jedná se o množství prvků API, které obsahuje. Ideální je, pokud má knihovna málo prvků a vysoké procento dokumentace.

Nejlépe v hodnocení tohoto ukazatele vyšla knihovna Riverpod a hned za ním Bloc, který sice má nižší počet prvků, ale má jich procentuálně méně zdokumentováno. Nicméně pokud bych se musel rozhodovat na základě tohoto faktoru, tak jak knihovna Bloc, tak i Riverpod by byla dobrá volba. GetX zde absolutně zaostává s jeho 2454 prvky API a dokumentací pouze na 30,32 % z nich. Tento výsledek by mi stačil, abych GetX mohl vyřadit.

	Otevřených issues	Uzavřených issues	Vyřešených [%]
GetX	665	1221	45,54
Bloc	88	2187	**95,98**
Riverpod	89	**964**	90,77

Tabulka 2 - Hodnocení issues (vlastní)

Jako druhý ukazatel používám procentuální množství vyřešených problémů na GitHubu. Tento ukazatel mi dává přehled o tom, jak autor nebo komunita přispívá k řešení problémů, na které někdo z uživatelů knihovny narazí. Zároveň jsou tam zahrnuty části, které uživatelé chtějí do knihovny doplnit a chybějí tam.

V hodnocení tohoto kritéria opět prohrál GetX, kde sice nevzniká tolik problémů, ale nikdo již vzniklé problémy neopravuje anebo velmi pomalu. Na prvním a druhém místě zde máme opět Bloc a Riverpod. Bloc má nejlepší procentuální úspěšnost v řešení problémů, avšak nejvíce jich za dobu jeho životnosti vzniklo. To nemusí být nezbytně záporná informace, ale je potřeba dát si na to také pozor. Ovšem vzhledem k rychlosti řešení a úspěšnosti, je tento problém zanedbatelný.

Riverpod má úspěšnost řešení o něco nižší než Bloc, avšak vzniklých issues za dobu jeho životnosti nebylo zdaleka tolik. Nicméně v tomto kritériu vychází lépe Bloc se svými 95,98 % vyřešených issues, což je výborná hodnota.

	Otevřených požadavků	Uzavřených požadavků	Vyřešených [%]
GetX	67	440	84,77
Bloc	4	1132	**99,65**
Riverpod	8	**425**	98,12

Tabulka 3 - Hodnocení požadavků (vlastní)

Třetím ukazatelem je procentuální množství vyřešených požadavků na GitHubu. Tento ukazatel je víceméně obdobou ukazatele číslo dva, proto o něm nebude uvedeno tolik informací. Jak lze vidět, GetX opět zůstává opodál v porovnání s konkurencí, a to vzhledem k hodnocení v předchozích kritérii znamená, že tato knihovna pro mě nemá žádný význam a mohu ji s čistým svědomím vyloučit. Hodnocení Blocku a Riverpodu dopadlo obdobně jako v ukazateli zaměřujícím se na issues.

Závěrečné hodnocení tedy je takové, že GetX je pro výběr nevhodný. Co se týče Blocku a Riverpodu, dle mého názoru, jsou obě knihovny vhodné. Já jsem se rozhodl pro Riverpod a to jen proto, že jsem měl předešlé zkušenosti s knihovnou Provider, která je od stejného vývojáře jako právě Riverpod.

9.2 Instalace a využité balíčky

Pokud máme vybrány a náležitě ohodnoceny balíčky, tak dalším logickým krokem je integrace funkcionalit do našeho projektu. K tomu využijeme vestavěného terminálu uvnitř vývojového prostředí a zadáme tam příkaz **flutter pub add <název knihovny/balíčku>**. Pokud instalace proběhne úspěšně, můžeme uvnitř projektu importovat jednotlivé součásti a využívat všeho, co balíček či knihovna nabízejí.

Uvnitř projektu cashback aplikace využívám mnoho knihoven a balíčků. Rád bych některé z nich popsal a nastínil k čemu jsou uvnitř aplikace využívány. Knihovny jsou definovány v souboru **pubspec.yaml** pod **dependecies**. Notace pro definování knihovny je vždy její název a poté za dvojtečkou verze, kterou chceme využívat. Nebudu popisovat veškeré knihovny, ale poukážu na pár zásadních. (Výpis 2 – Použité knihovny)

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter

  cupertino_icons: ^1.0.2
  google_fonts: ^2.3.1
  animations: ^2.0.2
  shared_preferences: ^2.0.13
  webview_flutter: ^3.0.4
  http: ^0.13.4
  flutter_spinkit: ^5.1.0
  google_place: ^0.4.7
  flip_card: ^0.6.0
  intl: ^0.17.0
  flutter_local_notifications: ^9.8.0+1
  smooth_page_indicator: ^1.0.0+2
  local_auth: ^2.1.2
  flutter_launcher_icons: ^0.10.0
  flutter_native_splash: ^2.2.8
  qr_code_scanner: ^1.0.1
  intl_phone_number_input: ^0.7.0+2
  google_maps_flutter: ^2.2.1
  logger: ^1.1.0
  flutter_riverpod: ^1.0.4
  connectivity_plus: ^2.3.7
  pull_to_refresh: ^2.0.0
  firebase_core: ^2.2.0
  firebase_messaging: ^14.1.0
```

Výpis 2 - Použité knihovny (vlastní)

- **shared_preferences** – O této knihovně si povíme více v kapitole zaměřené na local storage. Do aplikace však přináší možnost lokálního uložení jednoduchých dat.
- **webview_flutter** – Modul, který nám umožňuje využití widgetu *WebView* na systémech Android a iOS.
- **http** – Tento balíček je jedním z nejjednodušších způsobů, jak vytvářet HTTP požadavky.
- **flutter_spinkit** – Jedná se o sadu mnoha animovaných indikátorů načítání, které jsou konfigurovatelné a jednoduché na používání.
- **google_place** – Tento balíček nám po zadání platného API klíče z google console umožní využívat našeptávače adres.
- **futter_local_notification** – Plugin, který poskytuje multiplatformní řešení pro zobrazování lokálních notifikací v moment, kdy je aplikace aktivní.
- **local_auth** – Umožňuje přístup k biometrickým ověřovacím API mechanismům na aktuálním zařízení.
- **google_maps_flutter** – Obdobně jako balíček google_place, tak i zde po zadání API klíče máme zpřístupněny specifické funkce poskytované společností Google. Zde se jedná o možnost zobrazit mapu uvnitř aplikace.
- **flutter_riverpod** – Jedná se o knihovnu, která nám umožňuje vylepšenou správu stavů widgetů uvnitř aplikace.
- **connectivity_plus** – Balíček, který rozpoznává, zda má zařízení aktivní připojení a případně jaké.
- **pull_to_refresh** – Velmi užitečná knihovna, která implementuje možnost načtení stránky popotažením od stavové lišty směrem dolů.
- **firebase_core a messaging** – Tyto knihovny umožňují propojení aplikace se službami Firebase a nabízejí tak možnost push notifikací ze serveru. Tomuto tématu se budeme věnovat dále v práci.

10 State management

Uživatelské rozhraní v aplikaci se musí na základě různých akcí měnit. Nezáleží úplně na tom, zda se jedná o změnu vyvolanou vstupem od uživatele, nebo o změnu vyvolanou například ze serveru. V moment, kdy se tato změna provede, aplikace musí správně zareagovat a upravit komponenty, které jsou touto změnou ovlivněny.

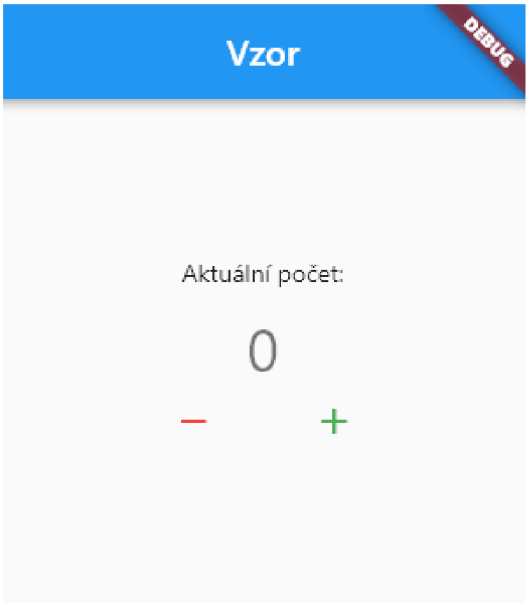
Komponent jako takový nic nedělá, kromě toho, že popisuje součást uživatelského rozhraní aplikace. Pokud uživatel očekává změnu komponenty na základě nějaké akce, změna musí nastat. K tomu, aby se změna vizuálně projevila slouží tzv. stav.

Něco málo o stavu jsme si pověděli v teoretické části této práce. Pod pojmem stav můžeme vidět jakákoliv data, která utvářejí uživatelské rozhraní v konkrétním čase. Pokud se tuto myšlenku pokusíme stručně rozvinout, jedná se například o hodnotu obsaženou uvnitř zaškrtávacího políčka. Od takového políčka očekáváme, že hodnotu budeme moci změnit pouhým kliknutím. To uvnitř zdrojového kódu znamená, že v jeden moment hodnota bude definována jako *false* čili políčko bude prázdné. Druhý stav, který toto políčko má, je jeho protiklad *true* a v tento moment očekáváme, že políčko bude zaškrtnuto.

Jak je již zmíněno v teoretické části této práce, pokud chceme využívat stavu jednotlivých widgetů, musí být widget definován jako tzv. *StatefullWidget*. Knihovna, kterou jsem vybral v předcházející kapitole, tuto skutečnost trochu upravuje. Riverpod nám umožňuje definovat si stav widgetu jako proměnou a poté využívat jeho hodnoty kdekoli v aplikaci, stačí si proměnou pouze zavolat. K tomu účelu se definuje tzv. provider. Ovšem definicí providera to nekončí, dalším krokem je úprava samotného widgetu.

Riverpod přináší upravené widgety, jedná se o spotřebitele neboli consumer. Toto rozšíření znamená, že již nemusíme využívat Statefull a Stateless widgety, ale právě upravené *ConsumerWidget* a *ConsumerStatefullWidget*. V těchto widgetech poté můžeme využívat právě našich definovaných providerů, kteří obsahují stav aplikace. Kód se díky tomu stává o hodně udržitelnějším, neboť můžeme logiku widgetu držet odděleně od jeho vizuálu.

Implementaci variant state managementu si demonstrujme na následujícím příkladě. Máme jednoduchou aplikaci obsahující dvě tlačítka a aktuální počet. První tlačítko je označeno mínus a druhé plus. Nad tlačítky máme vyobrazen počet s výchozí hodnotou nastavenou na 0. Stisknutím jednoho z tlačítek dojde k přičtení, nebo odečtení čísla 1. Tuto změnu aplikace po stisknutí okamžitě reprezentuje uživateli jako aktuální stav.



Obrázek 7 - Vzorová aplikace (vlastní)

10.1 StatefullWidget

První varianta řešení této aplikace je s použitím základního *StatefullWidgetu*, který Flutter nabízí bez dodatečných knihoven. Začněme definováním proměnné, která uchovává aktuální hodnotu, pod kterou nalezneme dvě tlačítka. Po vytvoření vizuálu stránky založme dvě metody, které voláme při kliknutí na tlačítka.

```
void incrementCounter() {
  setState(() {
    counter++;
  });
}

void decrementCounter() {
  setState(() {
    counter--;
  });
}
```

Výpis 3 - StatefullWidget metody (vlastní)

Metody nazvěme ideálně podle operací, které vykonávají. Můžeme tedy vidět metody *incrementCounter* a *dekrementCounter*. (Výpis 3 – StatefullWidget metody) Uvnitř metod je pomocí zkrácené notace zapsáno buď přičtení, nebo odečtení čísla jedna. Tato změna je poté uvnitř těla funkce nesoucí název *setState*. Po aktivaci tlačítka se změna provede a funkce *setState* informuje builder widgetu, že je potřeba přestavět některou z částí aplikace, což umožní uživateli vidět změnu prostřednictvím uživatelského rozhraní. Operace přestavby widgetů je popsána v kapitole 4.6.

10.2 ConsumerWidget

Druhá varianta demonstruje, jak bude zdrojový kód vypadat po použití Riverpodu. Definujme si provider a dvě funkce, opět jednu na přidání jedničky a druhou na její odebrání. (Výpis 4 – Riverpod provider)

```
final counterProvider = StateNotifierProvider<Counter, int>((ref) => Counter());

class Counter extends StateNotifier<int> {
  Counter() : super(0);
  void incrementCounter() => state++;
  void decrementCounter() => state--;
}
```

Výpis 4 - Riverpod provider (vlastní)

Logika funkcí se téměř nezměnila, avšak velkým rozdílem je to, že nyní je proměnná uchovávající hodnotu počítadla uvnitř provideru, a nikoliv na stránce jako takové. Globální definice provideru umožnuje zpřístupnění metod uvnitř *ConsumerWidgetu*.

Takto vypadá zpřístupnění providera a volání funkce v závislosti na vstupu od uživatele způsobené kliknutím na tlačítko plus nebo mínus (Výpis 5 – Provider volání).

```
IconButton(
  onPressed: () => ref.read(counterProvider.notifier).decrementCounter(),
  icon: const Icon(Icons.remove),
  color: Colors.red,
), // IconButton
const SizedBox(width: 40),
IconButton(
  onPressed: () => ref.read(counterProvider.notifier).incrementCounter(),
  icon: const Icon(Icons.add),
  color: Colors.green
), // IconButton
```

Výpis 5 - Provider volání (vlastní)

11 Routing

Schopnost navigace mezi různými stránkami aplikace je označována za tzv. routing. Tato funkcionalita je nezbytná pro každou vícestránkovou aplikaci. Routing umožňuje uživateli pohybovat se mezi různými sekcemi aplikace. V ideálním případě by veškeré stránky, na které se může uživatel prokliknout, měly být předem určeny a označeny.

11.1 Navigation stack

Z hlediska vývoje aplikace si musíme vysvětlit, co je to tzv. navigation stack. Při navrhování cest routingu musíme mít vždy přehled o tom, na jaké stránce se uživatel právě nachází a na jakých stránkách se nacházel předtím. Povědomí o těchto stavech nám dává dohromady právě navigation stack neboli navigační zásobník.

Pokud bychom jednoduše chtěli vysvětlit funkci tohoto zásobníku, jedná se o řadu obrazovek, které uživatel uvnitř aplikace navštívil. V moment, kdy je do zásobníku vložena nová obrazovka, jedná se o právě zobrazenou stránku, kterou může uživatel obsluhovat. Jestliže uživatel iniciuje ukončení stránky například tlačítkem zpět, tak by měla být obrazovka ze zásobníku vyjmuta a uživatel by měl být vrácen na předchozí, anebo námi vybranou stránku.

Uvnitř Flutteru je tento zásobník obsluhován pomocí widgetu *Navigator*, ten je zodpovědný za jeho udržování a poskytuje vývojáři metody pro vkládání a ukončování jednotlivých obrazovek. Existují dva přístupy, které lze využívat. Prvním přístupem je využívat navigátor tak, že při navigaci voláme stránky jejich přesným názvem jako widgety. Druhým, dle mého názoru vhodnějším přístupem, je definování si cest, které poté máme přidružené k jednotlivým stránkám a voláme je pod tímto značením.

Navigační zásobník je kritickou součástí každé aplikace. Pokud tato součást není správně navržena, může koncovému uživateli velmi znepříjemnit práci s aplikací. Pokud uvedu příklad, tak po přihlášení do aplikace je vhodné zásobník vyprázdnit a jako první obrazovku uvnitř mít například nástěnku. Jestliže zásobník nebude vyprázdněn, uživatel se pomocí systémových navigačních tlačítek může navzdory úspěšnému přihlášení vrátit zpět na přihlašovací obrazovku bez toho, aniž by se plnohodnotně odhlásil. Cesta zpět na nástěnku by poté vedla opět přes tlačítko přihlášení. Aplikace, která nebude mít kontrolu nad zásobníkem, nebude zahlcovat pouze zásobník, ale i paměť telefonu.

11.2 Řešení

Implementace routingu uvnitř cashback aplikace začíná definicí počáteční routy uvnitř *MaterialApp* widgetu, na které aplikace začne po jejím spuštění. Tato obrazovka se v průběhu životnosti aplikace mění.

Po prvním spuštění chceme uživateli zobrazit stránku umožňující přihlášení pomocí bankovní identity. Pokud po přihlášení uživatel aplikaci z nějakého důvodu opustí a nenastaví si bezpečnostní pin, tak chceme zobrazit stránku pro nastavení bezpečnostního pinu. Po jeho nastavení prezentujeme uživateli obrazovku pro potvrzení pinu.

Jestliže uživatel nastaví bezpečností pin úspěšně, zobrazíme mu nástěnku. Zobrazením nástěnky zároveň vyprázdníme navigační zásobník a jako první obrazovku v něm nastavíme právě nástěnku. Tímto předejdeme nevyžádanému chování, při kterém by se uživatel gestem zpět mohl vrátit na stránku vyžadující zadání pinu.

```
class MyApp extends ConsumerWidget {
  String decideMainPage() {
    if (localStorage.isValidUser() && localStorage.isPasscodeSet()) {
      return RoutePath.passcode.value;
    } else if (localStorage.isValidUser() && !localStorage.isPasscodeSet()) {
      return RoutePath.newPasscode.value;
    } else {
      return RoutePath.welcome.value;
    }
  }

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final ThemeProvider themeProvider = ref.watch(themeNotifier);
    final LocaleProvider localeProvider = ref.watch(localeNotifier);

    return MaterialApp(
      title: 'Cashback App',
      debugShowCheckedModeBanner: false,
      themeMode: themeProvider.themeMode,
      theme: Themes.themeData(localStorage.isDarkTheme, context),
      scrollBehavior: CustomScrollBehavior(),
      locale: localeProvider.locale,
      supportedLocales: AppLocalizations.supportedLocales,
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      initialRoute: decideMainPage(),
      routes: Routes.getRoutes(),
    ); // MaterialApp
  }
}
```

Výpis 6 - Počáteční routa (vlastní)

Widget *MaterialApp* voláme s definovaným parametrem *initialRoute*, který máme nastaven jako funkci *decideMainPage()*. (Výpis 6 – Počáteční routa) Zmíněná funkce má návratový datový typ *String*, což je řetězcová hodnota. Uvnitř funkce validujeme uživatele na základě jednoduché podmínky a poté rozhodujeme, která stránka bude počáteční.

```
enum RoutePath {
  offline('/offline'),
  welcome('/welcome'),
  bankId('/bankid'),
  passcode('/passcode'),
  newPasscode('/new_passcode'),
  newPasscodeConfirm('/new_passcode_confirm'),
  passcodeChange('/passcode_change'),
  newPasscodeChange('/new_passcode_change'),
  newPasscodeChangeConfirm('/new_passcode_change_confirm'),
  tutorial('/tutorial'),
  qrCode('/qr_code'),
  qrCodePayment('/qr_code_payment'),
  dashboard('/dashboard'),
  storeMap('store_map'),
  settings('/settings'),
  accountSecurity('/account_security'),
  devices('/devices'),
  partnerSettings('/partner_settings'),
  partnerRegisteredStores('/partner_registered_stores'),
  partnerRegistration('/partner_registration'),
  cardDetail('/card_detail'),
  cardDelivery('/card_delivery'),
  paymentSimulation('/payment_simulation'),
  manageFunds('/manage_funds'),
  storeWithdraw('/store_withdraw'),
  userDeposit('/user_deposit'),
  userWithdraw('/user_withdraw'),
  userMove('/user_move'),
  storeEdit('/store_edit');

  const RoutePath(this.value);
  final String value;
}
```

Výpis 7 - Routes (vlastní)

Veškeré routy jsou definovány v enumu uvnitř souboru **routes.dart**. (Výpis 7 – Routes) Při změně navigačního zásobníku se volá obrazovka na základě jmenné routy. Tento přístup umožňuje škálovatelnost a může usnadnit budoucí úpravy zdrojového kódu.

Kontrola a návrh navigačního zásobníku poté probíhá tak, že dle navigačního diagramu (Příloha 1 - Navigační diagram) navrhneme pohyb uživatele uvnitř aplikace. Veškeré obrazovky, které potřebujeme pro uživatele udržovat v zásobníku zaznamenáme. Tento postup poté musíme dodržovat při vývoji aplikace, stránky ze zásobníku mazat a přidávat nové.

```
Navigator.popAndPushNamed(
  context,
  RoutePath.newPasscodeConfirm.value,
);
```

Výpis 8 - Pop and Push (vlastní)

Jednou z nejpoužívanějších metod navigátoru při kontrole zásobníku je *popAndPushNamed()*. Jak už překlad napovídá, tato metoda jednoduše nahradí aktuální obrazovku v zásobníku. (Výpis 8 – Pop and Push)

```
Navigator.of(context).pushNamedAndRemoveUntil(
  RoutePath.dashboard.value,
  (Route<dynamic> route) => false,
);
```

Výpis 9 - Push and Remove Until (vlastní)

Další z funkcí je *pushNamedAndRemoveUntil()*. Tato metoda se používá například pro přesměrování uživatele z přihlašovací obrazovky na nástěnku. Použitím této metody zajistíme, že v zásobníku první obrazovkou bude námi zmíněná nástěnka a uživatel nebude mít žádnou jinou obrazovku, ke které by se mohl po přesměrování vrátit. (Výpis 9 – Push and Remove Until)

Metod v navigátoru existuje samozřejmě mnohem více. Další metoda, která stojí za zmínku, je *popUntil()*. Tato metoda uživateli umožní vymazat veškeré obrazovky v zásobníku, dokud nenarazí na tu, kterou aktuálně hledáme.

Další metodou je samostatný *push()*, který přidá stránku na vrch zásobníku bez toho, aniž by ovlivnil obrazovky pod ní. Přesným opakem je poté metoda *pop()*, která se hodí například při implementaci tlačítka zpět, nebo obdobné mechaniky, kdy potřebujeme odstranit pouze aktuální stránku a vrátit se.

12 Local storage

Ukládání uživatelských dat mimo vrstvu databáze a schopnost aplikace k nim přistupovat bez nutnosti připojení k internetu řeší lokální úložiště neboli local storage. Uživatelská data uložena v lokálním úložišti by neměla být kritická, neexistuje zde žádná záruka toho, že zapsaná data zůstanou uložena napořád. Pro vymazání těchto dat stačí uživateli např. vymazat data o aplikaci nebo občas i samotnou aplikaci pouze aktualizovat.

12.1 Shared preferences

Řešení lokálního úložiště nabízí knihovna s názvem **shared_preferences**, o které jsme se již zmínili v kapitole 9.2. Tato knihovna umožňuje aplikaci ukládat malé množství dat na zařízení, kde je momentálně spuštěna. Data jsou na zařízení uložena ve formě klíč-hodnota. Datové typy, ve kterých můžeme pomocí knihovny **shared_preferences** ukládat, jsou například čísla, řetězce nebo booleanovské hodnoty čili pravda/nepravda.

12.2 Řešení

Lokální úložiště v cashback aplikaci používám pro uchovávání uživatelských dat týkajících se nastavení aplikace. Mezi tato data patří například uživatelem zvolený jazyk, motiv aplikace, její stav a několik dalších, která nejsou pro uložení v databázi vhodná. Implementace lokálního úložiště poté vypadá následovně.

```
static Future<void> initializeSharedPreference() async {
  prefs = await SharedPreferences.getInstance();
}
```

Výpis 10 - Inicializace úložiště (vlastní)

Pro zpřístupnění lokálního úložiště v aplikaci musíme nejprve při jejím spuštění načíst veškerá data, která jsou pro danou aplikaci uložena na disku. Načtení dat dosáhneme tím, že ve funkci *main()* před buildem celé aplikace zavoláme metodu *initializeSharedPreferences()*. Tato asynchronní metoda nám zajistí, že do proměnné *prefs* budou vložena data z úložiště. To nám zajistí metoda *getInstance()*, kterou obsahuje třída *SharedPreferences*. (Výpis 10 – Inicializace úložiště)

13 Motiv aplikace

Navržení motivu usnadňuje vývojářům přizpůsobit vzhled a celkový dojem aplikace. Motiv samotný zahrnuje barevné schéma, typografii a celkový vizuální design. Dobře navržený motiv pomáhá zlepšení celkové vizuální estetiky, díky tomu aplikace pro uživatele působí přitažlivěji a snadněji se používá.

Používání motivů v aplikacích usnadňuje vývojářům provádění změn týkajících se vizuálního designu. Pokud z nějakého důvodu dojde k rozhodnutí změny barevné paletky, kterou aplikace využívá, dá se tato změna jednoduše vyřešit. To může ušetřit mnoho času a usnadnit udržení konzistentního designu aplikace.

13.1 Řešení

Implementace motivů ve Flutteru se globálně řeší uvnitř *MaterialApp* widgetu obdobně jako routing. Uvnitř tohoto widgetu se nachází parametry *themeMode* a *theme*. (Výpis 6 – Počáteční routa) Parametr *themeMode* určuje, který z motivů bude použit, pokud jsou k dispozici jak světlý, tak tmavý motiv aplikace. (Výpis 11 – Theme provider)

Uživatelský dojem z používaní aplikace je možné zpříjemnit tím, že výchozí motiv bude nastaven dle preferencí systému. Pokud má uživatel mobilní zařízení nastavené na tmavý režim, výchozím motivem aplikace bude také tmavý a naopak.

```
class ThemeProvider extends ChangeNotifier {
  late ThemeMode themeMode;

  ThemeProvider() {
    _init();
  }

  Future<void> _init() async {
    themeMode = localStorage.isDarkTheme ? ThemeMode.dark : ThemeMode.light;
    notifyListeners();
  }

  void setTheme(bool isDarkTheme) {
    LocalStorage.setValue(PrefsKeys.isDarkTheme.value, isDarkTheme);
    localStorage.getData();

    themeMode = localStorage.isDarkTheme ? ThemeMode.dark : ThemeMode.light;
    notifyListeners();
  }

  static bool getDefaultTheme() {
    return SchedulerBinding.instance.window.platformBrightness.name == Brightness.dark.name;
  }
}
```

Výpis 11 - Theme provider (vlastní)

Implementace zmíněného chování funguje tak, že při sestavování aplikace je motiv vybrán a nastaven uvnitř třídy *ThemeProvider*, která obsahuje proměnnou s názvem *themeMode*. Výchozí hodnotou proměnné *themeMode* je právě aktuální motiv zařízení.

Hodnota této proměnné je nastavena při inicializaci třídy *ThemeProvider*, protože uvnitř jejího konstruktoru dochází k volání metody *_init()*. Uvnitř této metody dochází ke kontrole klíče *isDarkTheme*, který uchovává booleanovskou hodnotu pro uživatelem uložený motiv. Jestliže hodnota *isDarkTheme* obsahuje *true*, uživateli je zobrazen tmavý motiv aplikace.

Výchozí hodnota klíče *isDarkTheme* je zajištěna inicializací lokálního úložiště. Pokud je hodnota pro klíč *isDarkTheme* v lokálním úložišti *null*, je zavolána metoda *getDefaultTheme()*, která vybere výchozí motiv na základě preferencí zařízení, na kterém je aplikace používána.

Změna motivu uvnitř aplikace funguje tak, že dochází k zavolání metody *setTheme()*, která má jako vstupní parametr booleanovskou hodnotu, která určí, jak bude motiv nastaven. Zavolání této metody převezme hodnotu z parametru a uloží ji do lokálního úložiště pro klíč *isDarkTheme* prostřednictvím metody *setValue()*. Poté je přepsána proměnná *themeMode* a aplikace je upozorněna na provedenou změnu vestavěným *notifyListeners()*. Ten můžeme využít, protože je obsažen v třídě *ChangeNotifier*, kterým rozšiřujeme naši třídu *ThemeProvider*.

Vizuální provedení a proces změny motivu aplikace lze vidět v příloze (Příloha 2 - Proces změny motivu). Změnu motivu může iniciovat uživatel na stránce „Nastavení“. Pokud tak neučiní, bude zvoleno výchozí nastavení zařízení, jak je popsáno výše v této kapitole.

14 Lokalizace a jazyk

Přizpůsobení aplikace pro uživatele žijící v jiné zemi nebo mluvících jiným jazykem řeší lokalizace, známá také jako l10n. V kontextu aplikace je tato funkcionalita nezbytná pro oslovení širšího publika, tím, že bude aplikace k dispozici ve více jazycích.

Tato funkcionalita může být důležitá zejména pro aplikace, jejichž ambicí je působit globálně. Nicméně implementace této funkcionality je dle mého názoru nezbytná i pro lokální aplikace. Díky této funkcionalitě můžou aplikaci používat i cizinci pobývající na území dané země nebo turisti.

Lokalizace dále může zlepšit samotný dojem aplikace v očích uživatelů hovořících jiným než výchozím jazykem, který je v aplikaci nastaven. To může pomoci zvýšit zapojení a udržení uživatelů, a zlepšit její pověst a hodnocení.

14.1 Řešení

Implementace lokalizace probíhá obdobně jako implementace motivů, protože se opět jedná o součást ovlivňující celou aplikaci. Řeší se tedy uvnitř *MaterialApp* widgetu. Tento widget obsahuje dva hlavní parametry týkající se lokalizace. Prvním je *locale* a druhým *supportedLocales*. (Výpis 6 – Počáteční routa)

Parametr *locale* nastavuje výchozí jazyk pro celou aplikaci. Pokud je hodnota tohoto parametru *null*, výchozí jazyk je zvolen dle nastavení systému. Jestliže jazyk systému není obsažen v podporovaných jazycích aplikace, které jsou předány parametru *supportedLocales*, tak se jako výchozí jazyk vybere první, který se v listu podporovaných jazyků nachází.

```
static const List<Locale> supportedLocales = <Locale>[
  Locale('cs'),
  Locale('en')
];
```

Výpis 12 - Podporované jazyky (vlastní)

Takto vypadá list podporovaných jazyků aplikace, který získáváme jako statickou proměnnou ze třídy *AppLocalizations*, která je automaticky generována díky **flutter_localizations**. Uvnitř listu (Výpis 12 – Podporované jazyky) můžeme vidět definované jazyky, které nesou identifikátor jazyka „Unicode“.

```
class LocaleProvider extends ChangeNotifier {
  Locale locale = Locale(localStorage.preferredLanguage);

  void setLocale(Locale newLocale) {
    LocalStorage.setValue(PrefsKeys.preferredLanguage.value, newLocale.languageCode);
    localStorage.getData();

    locale = newLocale;
    notifyListeners();
  }
}
```

Výpis 14 - Locale provider (vlastní)

Výběr výchozího jazyka funguje obdobně jako výběr výchozího motivu. Uvnitř třídy s názvem *LocaleProvider* (Výpis 13 – Locale provider) nalezneme proměnnou *locale*. Této proměnné je předána preferovaná hodnota uživatelem, pokud byla nastavena.

```
class Localization {
  static String getLanguage(String code) {
    switch (code) {
      case 'en':
        return 'English';
      case 'cs':
      default:
        return 'Čeština';
    }
  }

  static String getDefaultLanguage() {
    String deviceLocale = window.locale.languageCode;
    if (!AppLocalizations.supportedLocales.contains(Locale(deviceLocale))) {
      deviceLocale = 'en';
    }

    return deviceLocale;
  }
}
```

Výpis 13 - Localization (vlastní)

Jestliže při inicializaci lokálního úložiště nebyla nastavena žádná hodnota klíče *prefferedLanguage*, tak v metodě *getDefaultLanguage()* dochází k výběru výchozího jazyka aplikace. (Výpis 14 – Localization) Metoda nejprve převezme kód jazyka zařízení a poté kontroluje, zda se vybraný jazyk nachází v podporovaných jazycích aplikace. Pokud list jazyk obsahuje, dojde k jeho nastavení. Jestliže jazyk v podporovaných sadách neexistuje, výchozím jazykem aplikace je angličtina. Vizuální prezentace změny jazyka je demonstrována v příloze (Příloha 3 - Proces změny jazyka).

15 HTTP požadavky

Komunikace klienta a serveru je ve Flutteru zajištěna pomocí knihovny s názvem **http**. Pomocí této knihovny můžeme vytvářet a odesílat požadavky na webový server, za účelem provedení akce nebo získání informací. Jedná se o tzv. CRUD metody.

15.1 GET

Pro získání obsahu z webového serveru, databáze nebo webové stránky můžeme využít metody GET, která nám vrátí data. Dotazem na webový server získáme odpověď ve formátu, ve kterém je navržen dotazovaný endpoint. Obvykle se jedná o formát JSON. Pokud se dotazujeme webové stránky, odpovědí bývá obsah stránky v podobě HTML kódu.

```
Future<bool> isDeviceAuthenticated() async {
  final http.Response response = await http.get(
    Uri.https(url, '/device/authenticate').replace(
      queryParameters: <String, String>{
        'uid': localStorage.uid,
        'device_token': localStorage.deviceToken,
      },
    ),
  );
  final dynamic queryResponse = jsonDecode(response.body);
  final bool success = queryResponse['success'] as bool;
  if (!success) return success;
  final bool isDeviceAuthenticated = queryResponse['isDeviceAuthenticated'] as bool;
  return isDeviceAuthenticated;
}
```

Výpis 15 - Metoda GET (vlastní)

Implementace metody poté vypadá následovně. (Výpis 15 – Metoda GET) Pro ukázku jsem vybral požadavek na kontrolu ověření zařízení. Uvnitř asynchronní metody *isDeviceAuthenticated* vytvoříme proměnnou *response*. Hodnotou této proměnné jé poté odpověď z webového serveru. Odpověď získáme právě pomocí metody *get* poskytovanou knihovnou **http**, kde máme definovanou URL adresu webového serveru a konkrétní cestu k endpointu. Odpověď následně upravíme dle našich preferencí a zobrazíme uživateli, pokud je to potřeba. S touto metodou se v aplikaci můžeme setkat například při ověřování obchodu načteného z QR kódu při platbě (Příloha 4 - Proces platby pomocí QR kódu).

15.2 POST

Metoda POST slouží k odesílání dat na server. Obvykle se jedná o odesílání dat z formulářů v těle či za použití URL query daného požadavku. Implementaci požadavku POST můžete vidět v kapitole 17.2. (Výpis 20 – Firebase notifikace)

15.3 PUT

Tato metoda se používá k aktualizaci existujícího obsahu. Využití najdeme například při aktualizaci dat uvnitř databáze. Uživatel odešle požadavek s novými daty uvnitř query nebo těla požadavku. Na jejich základě se server připojí k databázi a pokusí se provést požadované změny. Pokud se požadavek provede, obvykle server vrátí odpověď s informacemi o úspěchu, nebo chybě.

```
Future<bool> putStore(String id, String cashback, String name) async {
  final http.Response response = await http.put(
    Uri.https(url, '/partner/store/info').replace(
      queryParameters: <String, String>{
        'storeId': id,
        'cashbackPercentage': cashback,
        'name': name,
      },
    ),
  );
  final dynamic queryResponse = jsonDecode(response.body);
  final bool success = queryResponse['success'] as bool;
  return success;
}
```

Výpis 16 - Metoda PUT (vlastní)

Implementace této metody probíhá naprosto totožně jako u metod GET a POST. Můžeme vidět ukázku implementace metody PUT s daty zasílanými prostřednictvím URL query stringu. (Výpis 16 – Metoda PUT)

15.4 DELETE

Poslední z metod je DELETE. Tato metoda se používá k odstraňování obsahu v databázi.

```
final http.Response response = await http.delete(
  Uri.https(url, '/device/logout').replace(
    queryParameters: <String, String>{
      'uid': localStorage.uid,
      'device_token': localStorage.deviceToken,
    },
  ),
);
```

Výpis 17 - Metoda DELETE (vlastní)

Implementaci této metody můžeme vidět na požadavku odstraňujícím ostatní přihlášená zařízení v aplikaci. (Výpis 17 – Metoda DELETE) Postup se opět od ostatních zmíněných metod liší pouze volbou metody v knihovně **http**.

16 BankId

Přihlášení a registrace do cashback aplikace probíhá prostřednictvím bankovní identity s použitím technického standardu OpenId. Tento standard se používá pro ověřování totožnosti uživatelů na internetu. Proces přihlášení s použitím BankId zajišťuje správnost získaných údajů jednotlivých uživatelů. Rozsah neboli scope získávaných informací od uživatele se liší a je definován na backendu při tvorbě OpenId identifikátoru, který má tvar běžného URL. Rozsah v identifikátoru musí odpovídat rozsahu nastavenému na stránkách BankId, kde je nutné aplikaci zaregistrovat.

```
router.get('/authenticate', (Request request) async {
  const String redirectUrl = 'https://oidcdebugger.com/debug';
  const String clientId = 'ckyewocl-477u-dkfd-poef-nq39z4z2oi1v';
  const String openIdScopes =
    '''
      profile.name profile.email profile.phonenumber
      profile.birthdate profile.locale profile.updatedat
    ''';
  final Map<String, String> queryParameters = <String, String> {
    'redirect_uri': redirectUrl,
    'client_id': clientId,
    'scope': 'openid $openIdScopes',
    'state': 'main002',
    'response_type': 'token',
  };
  final Uri uri = Uri.https('oidc.sandbox.bankid.cz', '/auth', queryParameters);
  return Response.ok(
    jsonEncode(<String, String> {'queryString': uri.toString()}),
    headers: jsonHeaders,
  );
});
```

Výpis 18 - OpenId identifikátor (vlastní)

OpenId identifikátor v aplikaci získáme zavoláním požadavku HTTP GET, který vygeneruje URL a vrátí ho zpět uživateli. (Výpis 18 – OpenId identifikátor) Tento proces by se obešel i bez generování identifikátoru na backendu, ale nebylo by to bezpečné.

Po získání identifikátoru je tato adresa v aplikaci otevřena pomocí WebView, kde po úspěšném přihlášení prostřednictvím BankId získáme *access_token*. Token poté odešleme pomocí požadavku HTTP GET na backend a pomocí OpenId získáme informace o uživateli, které máme definované rozsahem. Vizuální provedení registrace lze vidět v příloze (Příloha 5 - Proces registrace pomocí BankId). Po registraci je dále nutné zabezpečit aplikaci vstupním pinem (Příloha 6 - Proces nastavení pinu) a umožnit uživateli biometrické ověření totožnosti (Příloha 7 - Proces nastavení biometrického zámku).

17 Notifikace

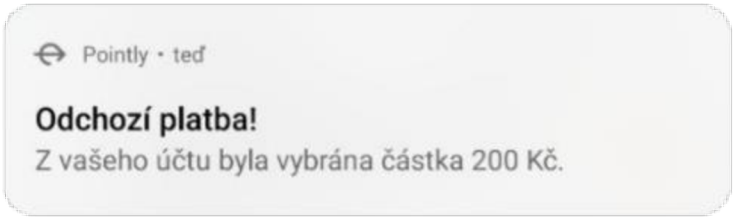
Upozornění v mobilních aplikacích se nazývá notifikace. Jedná se o formu upozornění zobrazovanou ve stavovém řádku zařízení. Tato upozornění mohou být generována přímo v aplikaci nebo je lze zasílat prostřednictvím serveru přes cloudové služby.

Notifikace nemusejí přenášet pouze titulek a krátký popis, lze s jejích pomocí zobrazit uživateli také obrázek nebo třeba přehrát zvuk. Mohou být nastaveny tak, aby se uživateli zobrazovaly opakovaně v ten samý čas nebo na základě časovače.

Jedná se o nepostradatelnou funkci u většiny moderních aplikací. Největší využití však nalezneme u aplikací chatovacích, bankovních nebo kalendářů, kde je potřeba informovat uživatele.

17.1 Lokální notifikace

Ve Flutteru lze využít knihovnu **futter_local_notification**, která umožňuje zobrazovat uživateli notifikace generované přímo uvnitř aplikace. (Obrázek 8 – Notifikace na zařízení) Toto řešení neumožňuje zasílat uživateli tzv. push notifikace ze serveru. Lze ho využít pouze pokud je aplikace aktivní, ačkoli vizuál notifikací nelze rozeznat.



Obrázek 8 - Notifikace na zařízení (vlastní)

Vyvolání takové notifikace uvnitř zdrojového kódu probíhá zavoláním funkce *showNotification*. (Výpis 19 – Lokální notifikace) Do této funkce zašleme pomocí vstupních parametrů id notifikace, titulek, a nakonec její obsah.

```
Future<void> showNotification({
  required int id,
  required String title,
  required String body,
}) async {
  final NotificationDetails details = await _notificationDetails();
  await _localNotificationService.show(id, title, body, details);
}
```

Výpis 19 - Lokální notifikace (vlastní)

17.2 Firebase Cloud Messaging

Pokud uživateli potřebujeme vyvolat notifikaci i v moment, kdy aplikace není aktivně spuštěna, musíme využít knihoven **firebase_core** a **firebase_messaging**. Tyto knihovny nám umožňují přijímat notifikace ze serveru přes cloudové služby společnosti Google. Vizuálně jsou tyto push notifikace shodné s těmi lokálními, ovšem jejich vyvolání je odlišné.

```
Future<void> sendNotification(String deviceToken, String title, String body) async {
final Uri uri = Uri.https(
  'fcm.googleapis.com',
  '/fcm/send',
);
await http.post(
  uri,
  headers: <String, String>{
    'Authorization': 'key=$apiToken',
    'Content-Type': 'application/json',
  },
  body: jsonEncode(<String, dynamic>{
    'to': deviceToken,
    'notification': <String, String>{
      'title': title,
      'body': body,
    }
  }),
);
```

Výpis 20 - Firebase notifikace (vlastní)

Jestliže je aplikace úspěšně zaregistrována a propojena na Firebase, uvnitř aplikace bychom měli být schopni získat token zařízení. Na backendu zasíláme notifikaci na konkrétní zařízení pomocí metody *sendNotification*. (Výpis 20 – Firebase notifikace) Vstupními parametry jsou právě *deviceToken*, *title* a *body*. Samotné zaslání notifikace poté funguje jako klasický požadavek HTTP POST. Požadavek je odeslán na servery provozované společností Google, které ho zpracují a následně vyvolají notifikaci prostřednictvím Firebase Cloud Messaging na konkrétním zařízení, které je odlišené unikátním *deviceTokenem*.

Závěr

Hlavním cílem této bakalářské práce bylo vyvinout funkční multiplatformní proof of concept frontend mobilní cashback aplikaci ve frameworku Flutter. Mezi hlavní cíle aplikace patřilo umožnit uživateli platbu pomocí QR čtečky, registraci partnerských účtů na základě identifikačního čísla a adresy, registraci obchodů, přihlášení do aplikace prostřednictvím služby BankId, ověření uživatele pomocí biometrických údajů, změnu motivu a jazyka, a zobrazení zůstatku a transakcí formou dashboardu.

Veškeré zásadní funkce aplikace byly implementovány a otestovány pro platformy Android a iOS. Po konzultaci s vedoucím práce jsme upustili od tvorby aplikace pro webové rozhraní z důvodu ztráty některých funkcionalit, pro které byla aplikace navržena.

Využití Flutter frameworku se ukázalo jako správná volba, a to hlavně díky jeho široké komunitě a komponentům, které ve velké míře usnadnily celkový vývoj aplikace.

Mobilní aplikaci Pointly dostalo k otestování celkem sedm uživatelů, a to z důvodu získání zpětné vazby. Vizuální prostředí aplikace uživatelé hodnotili jako přívětivé, ale samotné testování se neobešlo bez chyb. V průběhu bylo nalezeno pár grafických nedostatků, a to primárně ve světlém módů aplikace. Na odhalené nedostatky jsem byl uživateli upozorněn a opravil je. Funkční stránka aplikace se v této fázi testování obešla bez významných chyb. Aplikace uživatelům připadala přehledná a intuitivní.

Vzhledem k výše uvedenému lze považovat cíle práce za splněné.

Seznam použité literatury

1. *Architectural layers*. (2017). Flutter. 2022, dostupné z https://docs.flutter.dev/resources/architectural-overview
2. Basu, S. (2020). *What is the difference between hybrid and cross-platform apps ... - quora*. www.quora.com. 2022, dostupné z https://www.quora.com/What-is-the-difference-between-hybrid-and-cross-platform-apps-development
3. *Flutter architectural overview*. (2017). Flutter. 2022, dostupné z https://docs.flutter.dev/resources/architectural-overview
4. Gillis, A. S. (2020, 1. září). *native app*. SearchSoftwareQuality. 2022, dostupné z https://www.techtarget.com/searchsoftwarequality/definition/native-application-native-app
5. *Google Trends*. (2022). Google Trends. 2022, dostupné z https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11f03_rzbg,%2Fg%2F1q6l_n0n0,%2Fg%2F11h03gfxy9,Xamarin,%2Fm%2F0_lcrx4
6. *Hot reload*. (2017). Flutter. 2022, dostupné z https://docs.flutter.dev/development/tools/hot-reload
7. *Introduction*. (2017). Flutter. 2022, dostupné z https://docs.flutter.dev/resources/faq
8. Koďousková, B. (2021, 22. říjen). *Vývoj hybridní aplikace Pro Podnikání, srovnáme pro a proti*. Rascasone. 2022, dostupné z https://www.rascasone.com/cs/blog/co-je-hybridni-aplikace
9. Kukic, A. (2017, 10. leden). *Alternatives to Native Mobile App Development*. Auth0 - Blog. 2022, dostupné z https://auth0.com/blog/alternatives-to-native-mobile-app-development/
10. Medewar, S. M. (2022, 2. únor). *7 Best IDEs for Mobile App Development*. GEEKFLARE. 2022, dostupné z https://geekflare.com/best-ide-for-mobile-app-development/
11. *Meet Android Studio*. (2022). Developer Android. 2022, dostupné z https://developer.android.com/studio/intro
12. Miszewski, M. M. (2022, 26. květen). *Top 9 Mobile App Development Frameworks in 2022*. mDevelopers. 2022, dostupné z https://mdevelopers.com/blog/top-9-mobile-app-development-frameworks-in-2022

13. Silas K. (2021, 3. květen). *Flutter 2: Null Safety in a nutshell*. Medium. 2022, dostupné z https://medium.com/nerd-for-tech/flutter-2-null-safety-in-a-nutshell-f20aeb74772
14. stackpath. (2022). *What is a web application?* stackpath.com. 2022, dostupné z https://www.stackpath.com/edge-academy/what-is-a-web-application
15. Sullivan, M. S. (2019, 4. leden). *Flutter: Don't Fear the Garbage Collecto*. Medium. 2022, dostupné z https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30
16. *Technology*. (2017). Flutter. 2022, dostupné z https://docs.flutter.dev/resources/faq
17. *The language*. (2017). Dart. 2022, dostupné z https://dart.dev/overview#platform
18. *The platforms*. (2017). Dart. 2022, dostupné z https://dart.dev/overview#platform
19. *What is Kotlin*. (2022). Kotlin. 2022, dostupné z https://kotlinlang.org/docs/multiplatform-mobile-faq.html#what-is-kotlin-native-and-how-does-it-relate-to-kotlin-multiplatform-mobile
20. *Xcode*. (2022). Developer Apple. 2022, dostupné z https://developer.apple.com/documentation/xcode

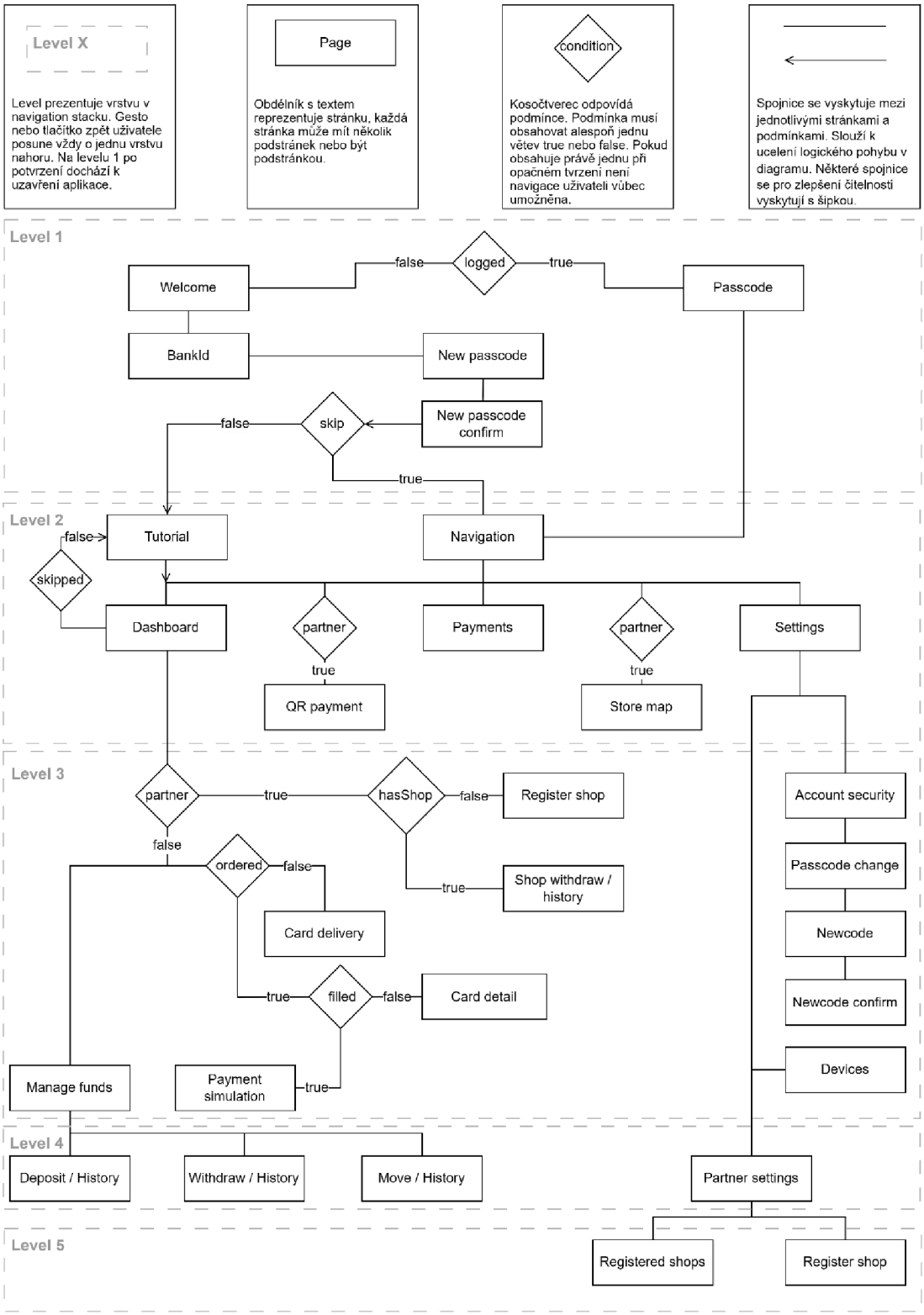
Seznam obrázků

Seznam výpisů zdrojového kódu

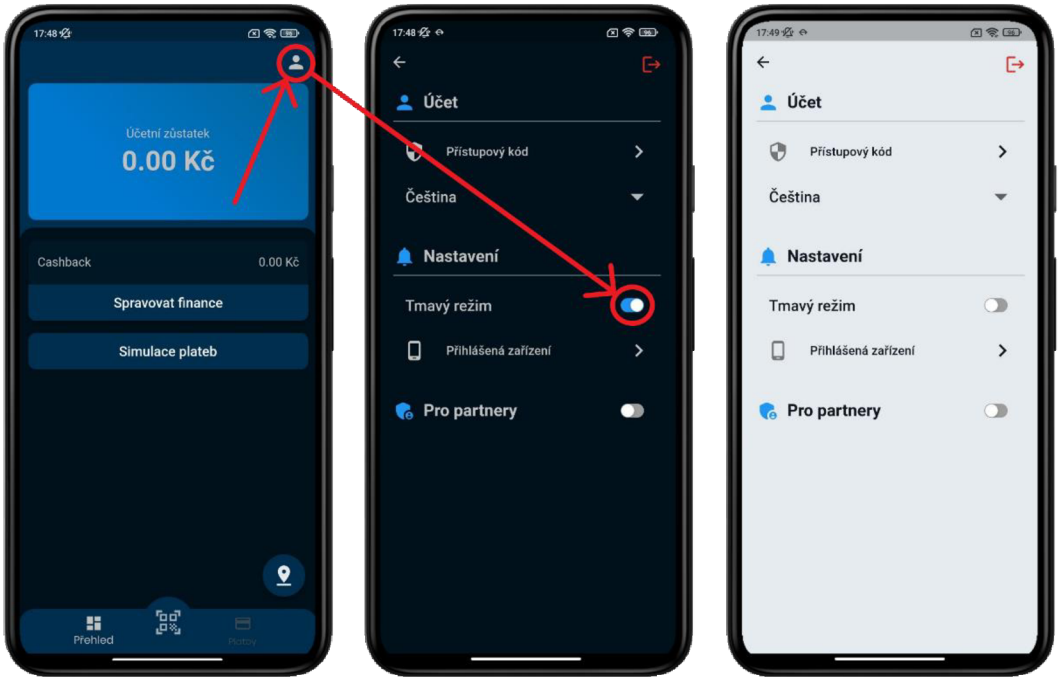
Seznam tabulek

Seznam příloh

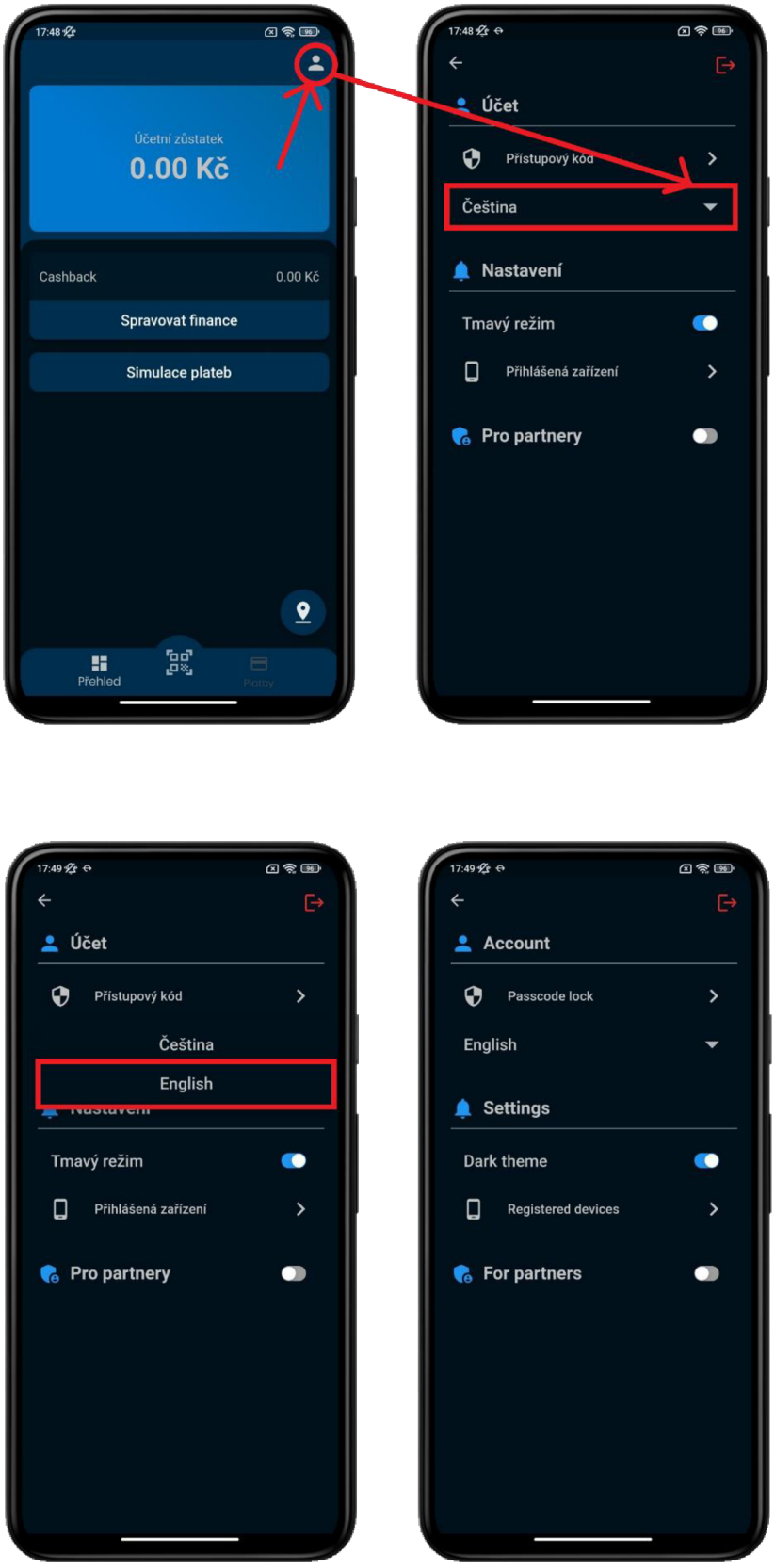
Přílohy



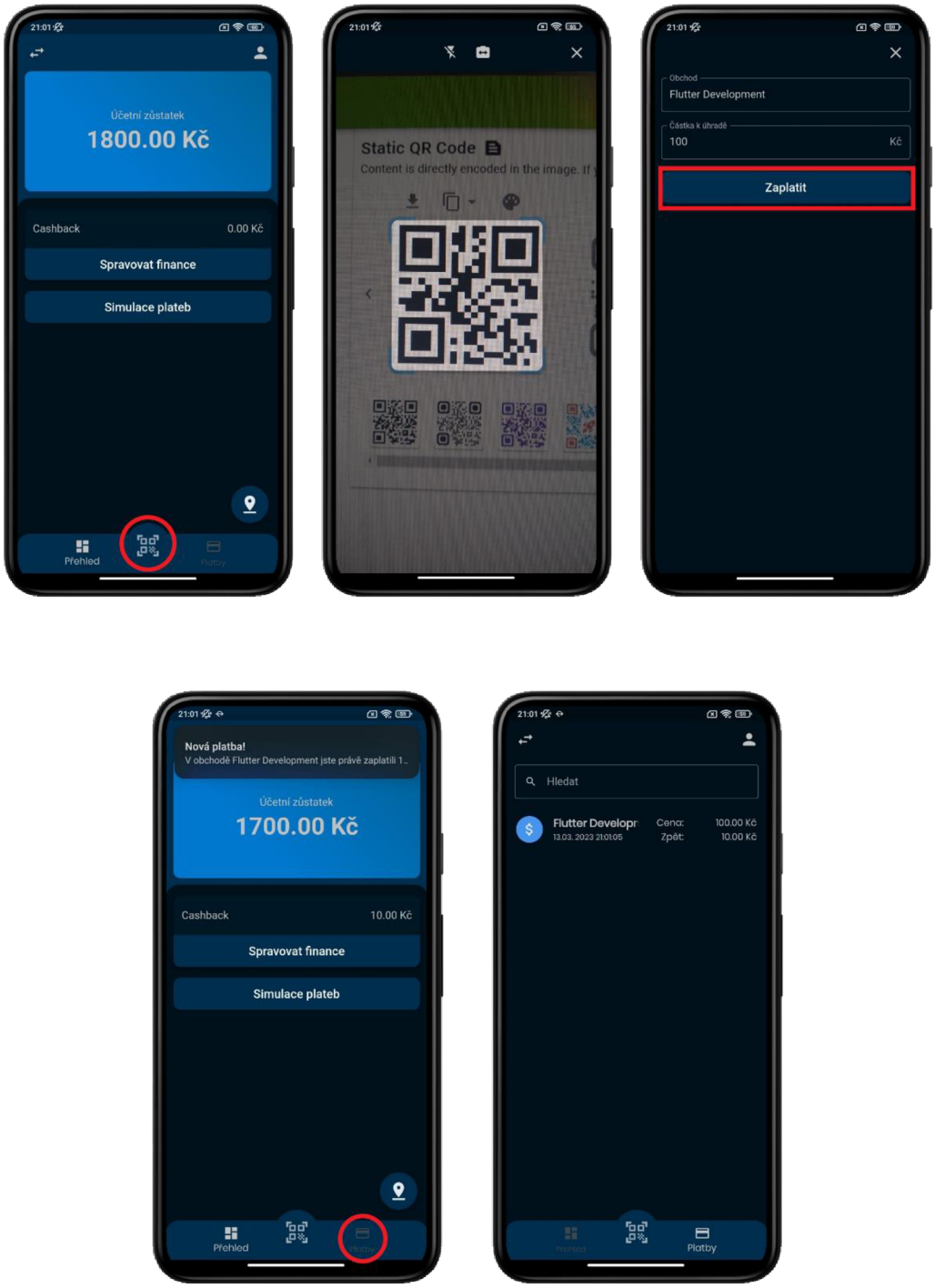
Příloha 1 - Navigační diagram (vlastní)



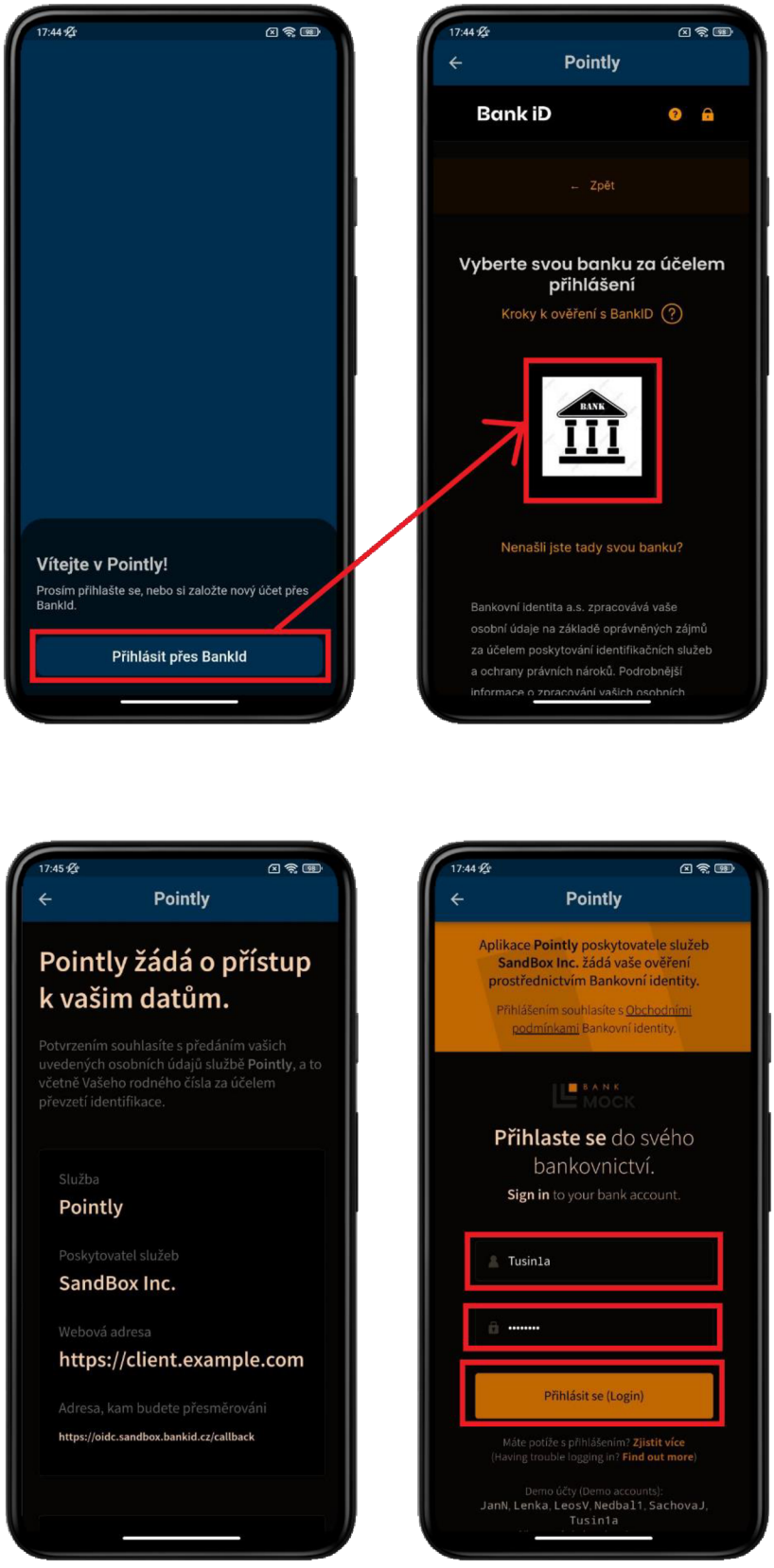
Příloha 2 - Proces změny motivu (vlastní)



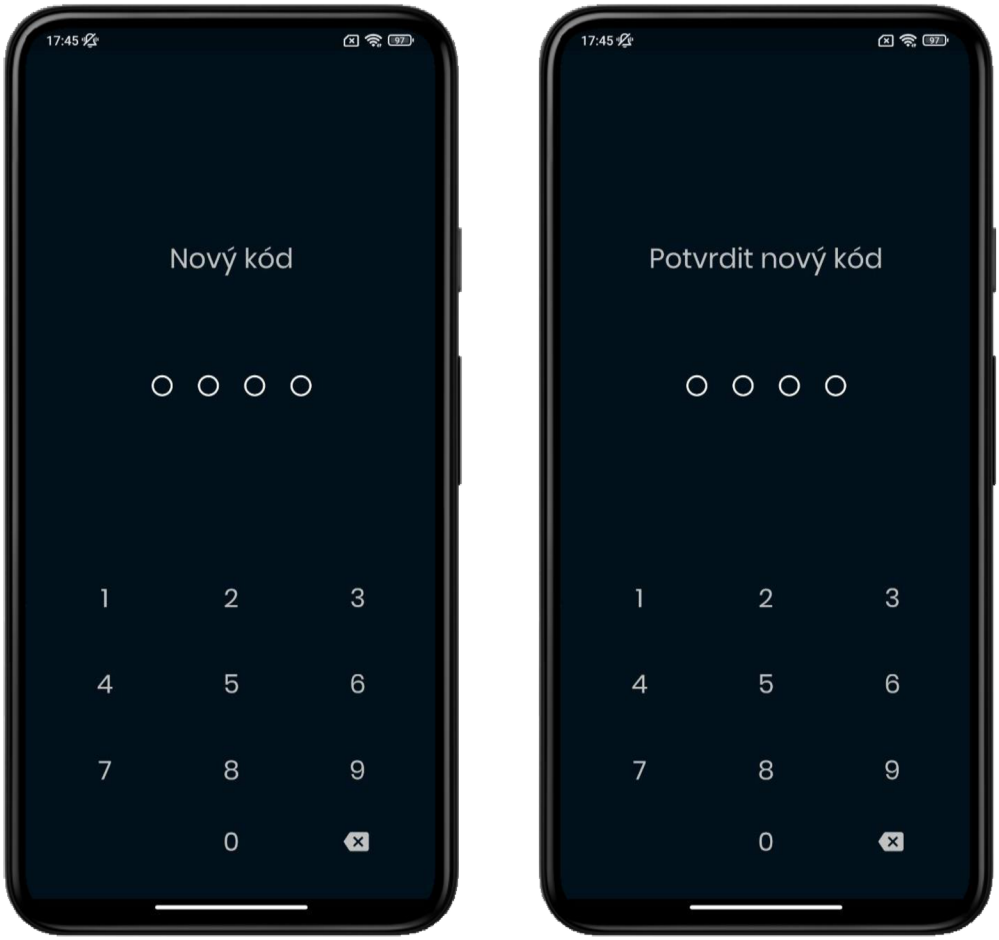
Příloha 3 - Proces změny jazyka (vlastní)



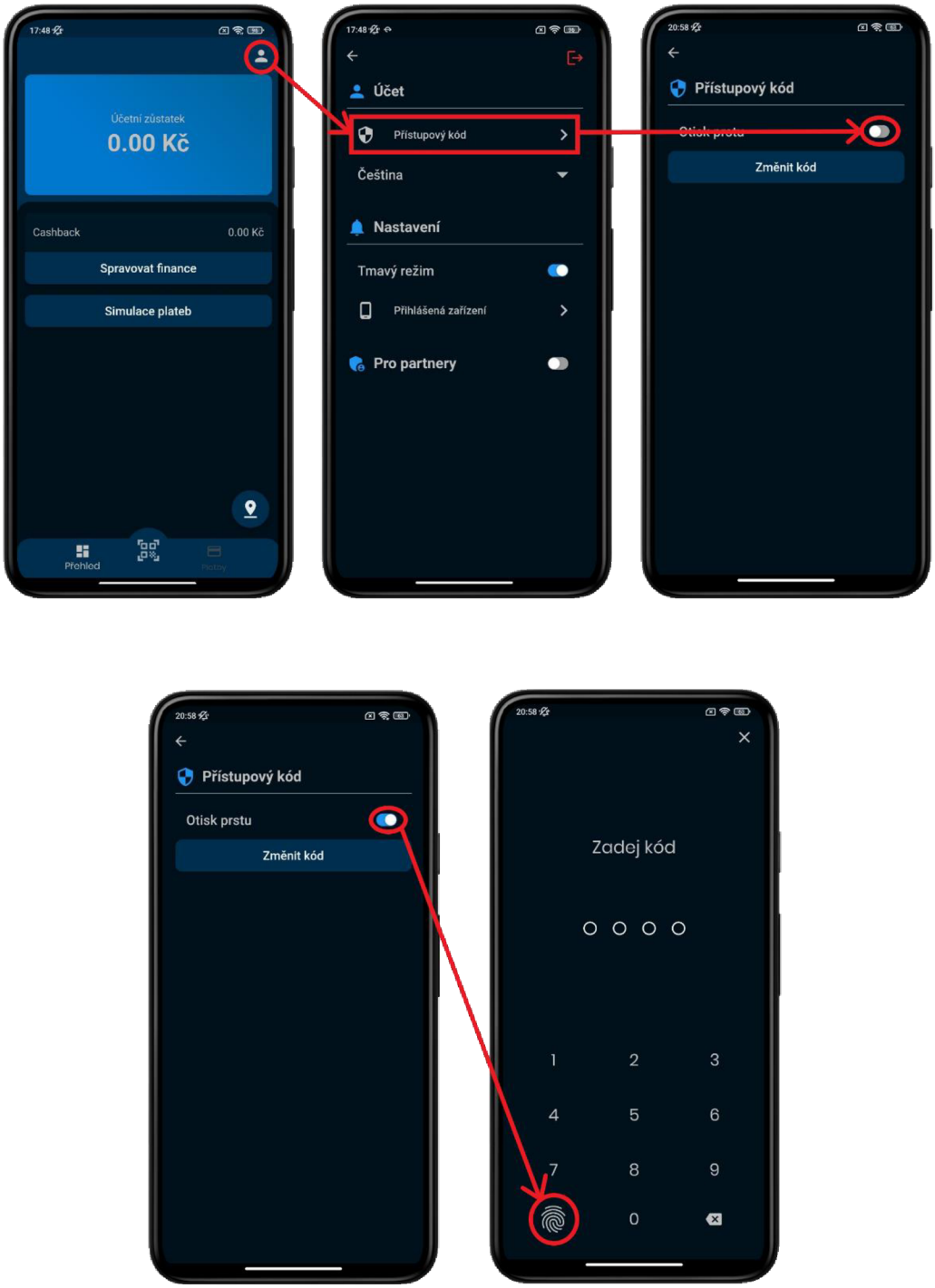
Příloha 4 - Proces platby pomocí QR kódu (vlastní)



Příloha 5 - Proces registrace pomocí BankId (vlastní)



Příloha 6 - Proces nastavení pinu (vlastní)



Příloha 7 - Proces nastavení biometrického zámku (vlastní)