Multiplatformní mobilní aplikace pomocí technologie Flutter

Bc. Martin Pokorný

Diplomová práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martin Pokorný**
Osobní číslo:	**A17246**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačové a komunikační systémy**
Forma studia:	**prezenční**
Téma práce:	**Multiplatformní mobilní aplikace pomocí technologie Flutter**
Téma anglicky:	**A Multiplatform Mobile Application using Flutter Technology**

Zásady pro vypracování:

1. **Stručně definujte základní metody vývoje mobilní aplikace z oblasti nativní i hybridní.**
2. **Popište některé aktuálně dostupné frameworky pro vývoj mobilních aplikací a proveďte jejich srovnání.**
3. **Zpracujte podrobnou rešerši o technologii Flutter od Googlu.**
4. **Navrhněte vhodné řešení projektu mobilní aplikace za využití technologie Flutter od Googlu.**
5. **Realizujte projekt mobilní aplikace pro tracking odpracovaného času a integrujte jej do existujícího systému ve firmě PRIA SYSTEM.**
6. **Popište klíčové části projektu vzniklého zpracováním bodu 5.**
7. **Demonstrujte výsledky a zhodnoťte využití technologie Flutter od Googlu.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **Flutter – Beautiful native apps in record time. Flutter.io [online]. Mountain View: Google, 2018 [cit. 2018-11-26]. Dostupné z: https://flutter.io/**
2. **WINDMILL, Eric. Flutter in Action [online]. Portland: Manning Publications, 2018 [cit. 2018-11-26]. ISBN 9781617296147. Dostupné z: https://www.manning.com/books/flutter-in-action**
3. **Dart programming language. Dart programming language [online]. 2018: Google, 2018 [cit. 2018-11-26]. Dostupné z: https://www.dartlang.org/**
4. **FRANCESCHI, Hervé. Android app development. Burlington, MA: Jones & Bartlett Learning, [2018]. ISBN 978-1284092127.**
5. **GAUCHAT, J. D. IOS Apps for Masterminds 4th Edition: How to take advantage of Swift 4.2, iOS 12, and Xcode 10 to create insanely great apps for iPhones and iPads. 4. Toronto: Amazon Digital Services, 2018. ISBN 978-1724466440.**
6. **PANHALE, Mahesh. Beginning hybrid mobile application development. New York, NY: Apress, [2016]. ISBN 978-148-4213-148.**
7. **MEIER, Reto. Professional Android 4e. 4. Indianapolis, IN: John Wiley, 2018. ISBN 978-1118949528.**
8. **IVERSEN, Jakob a Michael EIERMAN. Mobile App Development for iOS and Android, Edition 2.0. 2. Burlington, VT: Prospect Press, 2017. ISBN 978-1943153282.**

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **30. listopadu 2018**

Termín odevzdání diplomové práce: **17. května 2019**

Ve Zlíně dne 10. prosince 2018

doc. Mgr. Milan Adámek, Ph.D.
děkan

Ing. Miroslav Matýsek, Ph.D.
ředitel ústavu

Jméno, příjmení: Martin Pokorný

Název diplomové práce: Multiplatformní mobilní aplikace pomocí technologie Flutter

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že 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;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že 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 samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 13.5.2019 Martin Pokorný, v. r.

ABSTRAKT

Hlavním cílem této diplomové práce je popis techniky vývoje multiplatformních mobilních aplikací s využitím frameworku Flutter. V úvodu práce je shrnuta problematika vývoje mobilních aplikací z hlediska různých metodik vývoje a popis tržní situace z hlediska globálního podílu dílčích platforem. Teoretická část práce se věnuje srovnání frameworků pro multiplatformní vývoj mobilních aplikací, představení technologie Flutter do hloubky a popisu možných architektur mobilních aplikací vyvinutých touto technologií. V praktické části je potom na základě poznatků získaných v teoretické části navržena implementační architektura aplikace, která je demonstrována na výstupní aplikaci.

Klíčová slova: Mobilní aplikace, Flutter, iOS, Android, Dart, BLoC, multiplatformní vývoj

ABSTRACT

Primary goal of this diploma thesis is to describe techniques of cross-platform mobile application development using the Flutter framework. In the beginning there is a summary of mobile application problematics regarding the methods of development, and the description of mobile platforms market share on the global scale. Theoretical part focuses on comparison of individual cross-platform mobile application development frameworks, in-depth introduction of Flutter framework and description of architectural patterns which are applicable on mobile applications built with Flutter framework. Practical part then includes implementation of architectural pattern based on the knowledge gained from the theoretical part. This pattern is then applied on demonstrative application.

Keywords: Mobile application, Flutter, iOS, Android, Dart, BLoC, cross-platform development

Tímto chci poděkovat vedoucímu práce Ing. Radku Valovi, Ph.D. za odborné rady, organizaci a konzultaci diplomové práce.

„Technology, like art, is a soaring exercise of the human imagination. “

Daniel Bell

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

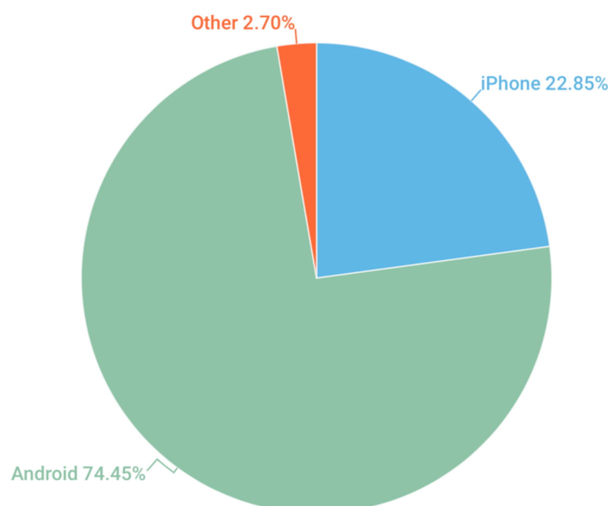
OBSAH

I OBSAH 7
II ÚVOD 9
III I. 11
IV TEORETICKÁ ČÁST 11
V 1 AKTUÁLNĚ DOSTUPNÉ FRAMEWORKY 12
1.1 DŮVODY POUŽITÍ MULTIPLATFORMNÍHO FRAMEWORKU 12
1.2 FLUTTER (GOOGLE) 13
1.3 REACT NATIVE (FACEBOOK) 13
1.4 XAMARIN.FORMS (MICROSOFT) 14
1.5 CORDOVA (APACHE) – DŘÍVE TÉŽ PHONEGAP (ADOBE) 15
1.6 IONIC A CAPACITOR (DRIFTY) 16
1.7 QT 16
1.8 SHRNUTÍ SROVNÁNÍ 17
VI 2 FLUTTER – PODROBNÁ REŠERŠE 18
2.1 DART 19
2.1.1 Asynchronní programování 19
2.1.2 Kompilace 20
2.1.3 Generační garbage collector 20
2.2 STREAM, SINK A STREAMCONTROLLER 22
2.3 WIDGET 23
2.3.1 Stateless widget 24
2.3.2 State 24
2.3.3 Stateful widget 24
2.3.4 Inherited widget 25
2.4 PROPOJENÍ S NATIVNÍM KÓDEM 25
2.5 REFLEXE, SERIALIZACE A DESERIALIZACE JSON DAT 26
2.6 ŽIVOTNÍ CYKLUS FLUTTER APLIKACE 27
2.6.1 Inactive 27
2.6.2 Paused 27
2.6.3 Resumed 28
2.6.4 Suspending 28
VII 3 VHODNÉ NÁVRHOVÉ MODELY (ARCHITEKTURY) 29
3.1.1 Redux 29
3.1.2 ScopedModel 30
3.1.3 Business Logic Component (BLoC) 31
VIII II. 33
IX PRAKTICKÁ ČÁST 33
X 4 DEMONSTRAČNÍ PROJEKT – MOBILNÍ APLIKACE 34
4.1 VOLBA NÁVRHOVÉHO VZORU PRO ZPRACOVÁNÍ PROJEKTU 35

4.1.1 KNIHOVNA FLUTTER_BLOC ... 35
4.1.2 INVERSION OF CONTROL KONTEJNER GET_IT ... 36
4.2 NÁVRH ARCHITEKTURY APLIKACE ... 37
4.3 STRUKTURA FLUTTER ČÁSTI PROJEKTU ... 38
4.4 DŮLEŽITÉ SOUČÁSTI NATIVNÍCH ČÁSTÍ PROJEKTU ... 39
4.4.1 INICIALIZACE FRAMEWORKU FLUTTER ... 39
4.4.2 KONFIGURACE APLIKACÍ ... 40
XI 5 REALIZACE DEMONSTRAČNÍHO PROJEKTU ... 41
5.1 VÝCHOZÍ BOD APLIKACE – SOUBOR MAIN.DART ... 41
5.1.1 REGISTRACE SERVISNÍCH TŘÍD A SPUŠTĚNÍ APLIKACE ... 41
5.1.2 KOŘENOVÝ WIDGET APLIKACE ... 42
5.1.3 KOŘENOVÝ STATE (STAV) APLIKACE ... 42
5.2 ENTITY, SERIALIZACE, DESERIALIZACE, JSON ... 47
5.3 SERVICE (SERVISNÍ) TŘÍDY ... 48
5.3.1 TŘÍDY PRO OBSLUHU LOKÁLNÍHO ÚLOŽIŠTĚ ... 49
5.3.2 TŘÍDA PRO OBSLUHU LOKÁLNÍCH NOTIFIKACÍ ... 51
5.3.3 TŘÍDY PRO OBSLUHU NAPOJENÍ NA SERVER PROSTŘEDNICTVÍM API ... 52
5.4 BLOC TŘÍDY ... 54
5.4.1 STATE ... 54
5.4.2 EVENT ... 56
5.4.3 BLOC ... 58
5.4.4 SHRNUTÍ ... 63
5.5 TŘÍDY UŽIVATELSKÉHO ROZHRANÍ ... 64
XII ZÁVĚR ... 75
XIII SEZNAM POUŽITÉ LITERATURY ... 76
XIV SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 80
XV SEZNAM OBRÁZKŮ ... 81
XVI SEZNAM TABULEK ... 82
XVII SEZNAM PŘÍLOH ... 83

ÚVOD

V oblasti mobilních zařízení jsou dnes relevantní dvě platformy – Google Android a Apple iOS, které k datu psaní této práce mají globální podíl na trhu rozdělený přibližně poměrem 74 % ku 23 %. Zbývající 3 % trhu obsazují alternativní platformy, kam se může řadit např. dnes již neaktualizovaná mobilní varianta Microsoft Windows. [1]



Obrázek 1 Rozdělení trhu z pohledu mobilních systémů z kraje roku 2019 [1]

Přestože se tržní situace postupně vytříbila do stavu, kdy zákaznickou relevanci již mají jen dva operační systémy, z technologického hlediska si výrobce každé platformy šel svou cestou, čímž se z hlediska vývoje mobilních aplikací v lepším případě duplikuje práce, která vede na stejný konečný výsledek, anebo v horším případě může způsobit i nezbytné přehodnocení projektu, pokud vývojář např. zjistí, že nějakou funkci je možné implementovat jen na jedné z platforem. [2]

Problémy vzniklé dílčími rozdíly mezi mobilními platformami se snaží vyřešit frameworky, které vývoj sjednocují pod jeden programový základ a maximálně vyžadují pro platformu specifickou implementaci, která je následně opět volána z úrovně sdíleného kódu, který řeší fungování aplikace z makroskopického hlediska. [2]

Přístup, jakým tyto frameworky pracují, se významně liší jak už na úrovni koncepční (tedy zda jde o vývoj nativní či hybridní), tak na úrovni technologické (použitý programovací jazyk, značkovací jazyk, vnitřní fungování technologie). Do této oblasti lze také z jistého úhlu pohledu počítat i webové aplikace, které mají responzivní rozhraní, což je činí plnohodnotně ovladatelnými i na menších dotykových obrazovkách mobilních zařízení. [4]

Mezi hybridní aplikace se dřív řadily spíše aplikace, které v sobě kombinovaly webové technologie s technologiemi mobilními. Typicky tak, že uživatelské rozhraní bylo realizováno prostřednictvím uzpůsobeného webového kódu, který byl nezávislý na platformě, a z tohoto webového kódu byla volána vrstva kódu nativní, která by u běžné webové aplikace nebyla přístupná – může zde patřit např. přístup k dílčím senzorům mobilního zařízení, přístup na datové úložiště, GPS a další funkce, které lze ve webovém prohlížeči zprostředkovat jen v omezené formě. [42]

Dnes už je ale pojetí hybridních aplikací širší, protože právě technologie Flutter od Googlu, o které je tato diplomová práce, není ve svém principu nativní, ale zároveň ani nevyužívá prvky webového vývoje – webovou vrstvu totiž supluje svým vlastním vykreslovacím frameworkem, čímž se koncepčně přibližuje spíše tomu, jak typicky funguje herní engine. O vnitřním fungování technologie Flutter se pojednává dále v teoretické části. [4]

Tabulka 1 Obecné shrnutí metod vývoje [3] [4] [5] [43]

	Nativní	Hybridní (web)	Hybridní (Flutter)	Web
UI	XML, Storyboard	HTML, CSS, JS	Dart	HTML, CSS, JS
Logika	Java, Swift	JS + nativní	Dart + nativní	JS, PHP, C#
Sdílený kód	Ne	Ano	Ano	-
Distribuce	Play/App Store	Play/App Store	Play/App Store	-
Výkon	Nativní	Téměř nativní	Nativní	Omezený
Náklady	Vysoké	Střední	Střední	Nízké

Tabulka 1 uvádí příklady technologií potřebných pro vývoj UI a logiky aplikace, dále také definuje distribuční kanály a porovnává výkon, kde u webových aplikací dochází k mírnému overheadu v důsledku nepřímého přístupu k nativnímu API, což způsobuje pokles výkonu oproti nativnímu pojetí. Flutter tímto netrpí, neboť využívá výkonné C++ jádro. Výkon webu je pak omezen samotným prohlížečem a neblíží se žádné z variant samostatných aplikací. [5]

Náklady na vývoj jsou v tabulce č. 1 zamýšleny z hlediska času, tzn. jak dlouho vývoj které varianty aplikace teoreticky bude trvat. Nativní vývoj trvá nejdéle právě kvůli zmíněné duplikaci vývoje na dvě různé platformy, ostatní varianty přináší významnou míru sjednocení. [3]

I. TEORETICKÁ ČÁST

1 AKTUÁLNĚ DOSTUPNÉ FRAMEWORKY

Jak už bylo nastíněno v úvodním srovnání metod vývoje, existuje více frameworků pro vývoj multiplatformních mobilních aplikací, Flutter patří k nejmladším technologiím na trhu, ovšem na základě průzkumů se těší velké popularitě mezi vývojáři. [6]

Dalším významným frameworkem je v současnosti React Native, což je odnož Reactu od společnosti Facebook, která se zabývá vývojem mobilních aplikací za využití webových technologií. React se v době vzniku práce těší velkému tržnímu zájmu. [6]

Ze starších technologií je významným průkopníkem multiplatformního vývoje Xamarin, který dnes vlastní a dále vyvíjí Microsoft, ovšem jeho popularita výrazně klesá a v uznávaných průzkumech patří dokonce k nejméně oblíbeným frameworkům současnosti. Ještě méně populární, ovšem tržně i historicky také velmi významná technologie, je Cordova, která patří k jedné z prvních technologií, které nastartovaly hybridní multiplatformní vývoj. [6]

1.1 Důvody použití multiplatformního frameworku

Shrneme-li si rozdíly mezi platformami iOS a Android, nastíněnými i v úvodu, je viditelné, že mezi těmito platformami existuje prakticky naprostá vývojová nekompatibilita. Android je vyvíjen primárně v Javě, případně v posledních letech v jazyce Kotlin, a uživatelské rozhraní se skládá buď v těchto jazycích, anebo více typicky v XML formátu. [39]

Naopak na iOS se využívá Obj-C, případně u nových projektů Swift, ve kterém také lze skládat uživatelské prostředí aplikace, ale typičtější je použití storyboardů, což jsou z technického hlediska speciálně vyskládané značkovací soubory. [40] Ovšem jejich struktura je natolik komplexní, že bez Xcode editoru je práce s nimi časově příliš náročná a nepraktická.

Kompatibilita nativního vývoje mezi oběma platformami neexistuje, a proto na trhu působí multiplatformní frameworky. V některých ohledech se ale vždy všechny multiplatformní frameworky rozcházet budou, a tím je minimálně definice aplikace pro danou platformu.

Ta se na iOS nachází v konfiguračním plist souboru, kde jsou definovány veškeré údaje popisující aplikaci z hlediska systému a App Store. [40] Analogicky se tato principiálně podobná data na Androidu nachází v tzv. manifestu. [41] V těchto souborech jsou popsána

např. potřebná oprávnění (ta se také liší dle platformy), název aplikace, její identifikátor a další data, bez kterých aplikaci nejde na danou platformu vydat. [40] [41]

Dalším problémem, jakkoliv se z technologického hlediska vývoje může zdát malicherný, je estetické zasazení aplikace do dané platformy. Apple byl dříve striktní v tom, že by aplikace měly dodržovat jistý designový jazyk, nicméně s rostoucím množstvím aplikací byly tyto kontroly uvolněny. Stále by však aplikace na iOS měla splňovat ten předpoklad, že je pro uživatele systému snadno uchopitelná i bez předešlé znalosti této aplikace. [42]

Naopak Android více koncepčně sází na přístupnost, tedy ujištění v tom, že aplikace může být snadno přístupná i uživateli s omezením smyslových funkcí těla. Patří zde např. pravidla, že by aplikace měla být plně operovatelná beze zvuku, bez barev, v režimu vysokého kontrastu apod. [42]

I tuto problematiku se snaží multiplatformní frameworky napříč druhy sjednocovat. [42]

1.2 Flutter (Google)

Flutter je v době psaní práce nový framework (první stabilní verze vyšla v prosinci 2018), který se nejvíce od ostatních odlišuje tím, že má vlastní vykreslovací vrstvu, čímž se ve vnitřních principech více podobá frameworkům (engine) pro vývoj počítačových her. [6]

Flutter tak prvořadě používá základní UI prvky, které ale lze doplnit o vlastní, a navíc tyto prvky stejně vzhledově odpovídají tomu, jak na příslušné platformě mají vypadat v nativním podání – ve skutečnosti ale o nativní prvky nejde. [7]

Flutter se koncepčně inspiruje Reactem, ze kterého si vzal reaktivní přístup k programování. To znamená, že se obecně doporučuje používání architektury, jež využívá aktualizaci vnitřních stavů dle událostí, ze kterých vznikají nové vnitřní stavy, jejichž změny se propagují na UI úroveň. [8]

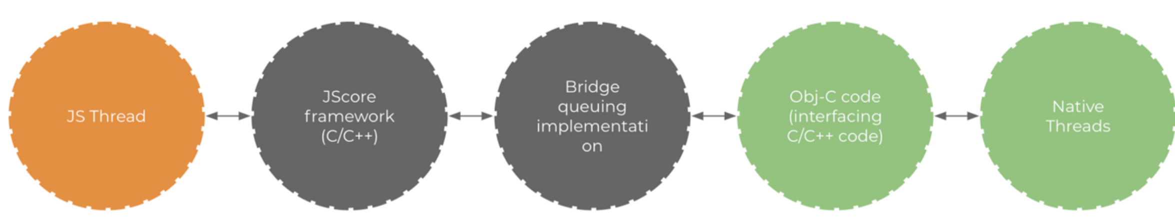
Flutter využívá programovací jazyk Dart pro sestavování UI i logiky, a kromě mobilních platforem je ve vývoji i podpora desktopových aplikací. [7] Flutter je hlavním tématem této diplomové práce, do hloubky je představen v kapitole 2.

1.3 React Native (Facebook)

React Native je technologie, která získává významnou trakci mezi webovými vývojáři, kteří zároveň chtějí pro projekt vytvořit mobilní rozhraní, jež využívá stejný kódový základ pro logiku.

React Native se nejvíce podobá Flutteru z hlediska architektury aplikace, protože také využívá událostmi řízené chování aplikací, nicméně už zde vzniká podstatný rozdíl v tom, jak je samotná aplikace zpracovávána při překladu na finální produkt. [9]

Napsaný kód se totiž překládá do úplně čistých nativních prvků dané platformy, což na jedné straně vývojářům umožňuje hlubší úpravy podoby dílčích prvků, ale nese to s sebou i negativum v podobě mnohem pracnějšího stylování finální podoby aplikace. [9]



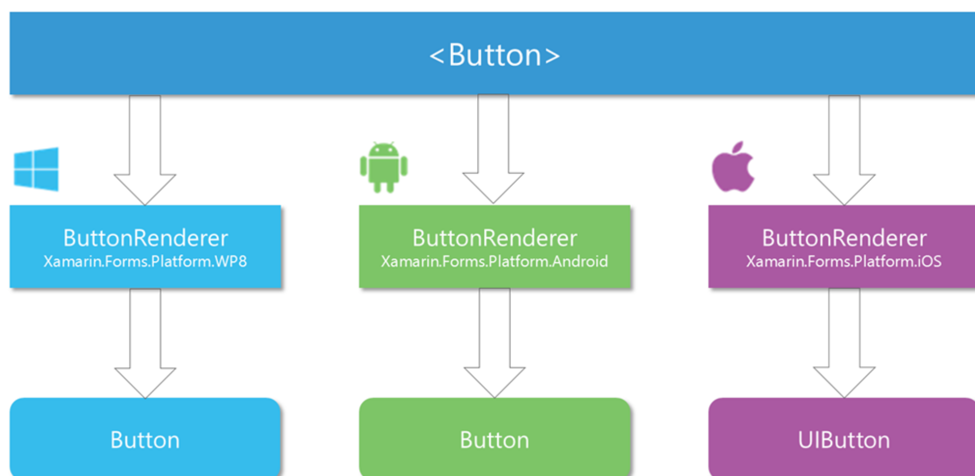
Obrázek 2 Flow z JavaScript vrstvy na vrstvu nativního iOS v Reactu [9]

Navíc je s tím spojena i vyšší závislost na dané platformě, tzn. pokud by např. Apple zcela přepracoval designový jazyk iOS, byla by tu velká pravděpodobnost, že React Native aplikace by se po aktualizaci přestaly vykreslovat korektně, neboť by došlo k zásadní změně základních prvků, ze kterých je poskládána. Další nevýhodou je nejistá kvalita komponent dodávaných třetí stranou – to je dáno tím, že React vlastní sadou nedisponuje a spoléhá čistě na komunitu vývojářů. React Native aplikace se ve své ryzí podobě programují pomocí JavaScriptu, ale lze využít např. i TypeScript. [10]

1.4 Xamarin.Forms (Microsoft)

Xamarin.Forms navázal na tradici frameworku Mono, který je open source implementací .NET frameworku bez závislosti na Microsoft Windows knihovnách. [11] Podobně jako React Native, i Xamarin.Forms provádí překlad UI kódu na nativní prvky a umožňuje stylování prakticky na nativní úrovni, kdy lze dokonce využívat nativní dokumentaci a jen uvažovat základní syntaktické a koncepční rozdíly mezi Javou (či Obj-C) a C#. [12]

Xamarin.Forms je ale poměrně stará technologie a už si s sebou nese jisté nedostatky, kterými novější technologie netrpí – patří zde např. absence živé aktualizace UI během vývoje (tzv. hot-reloading) a podstatně vyšší míra zapouzdření celé knihovny. To vede na to, že se části kódu chovají vyloženě jako černá skříňka – což s sebou může nést problémy. Typicky pokud se někde v kódu nachází kritická chyba, v důsledku je třeba čekat na aktualizaci celé knihovny, protože ji nejde jednoduše lokálně opravit. [13]



Obrázek 3 Xamarin.Forms a znázornění komunikace s nativní vrstvou [12]

Co do architektury, Xamarin.Forms vychází z WPF a s tím je spojená i doporučená architektura MVVM. Tato architektura funguje poněkud odlišně od obecně používaných architektur ve Flutteru a React Native v tom, že není explicitně řízena událostmi, nýbrž změnami dílčích proměnných, jejichž změna v backendu se automaticky propaguje do UI vrstvy (ovšem jen příslušnému UI prvku) a naopak (opět jen příslušné proměnné). [13]

Xamarin.Forms obecně cílí spíše na větší projekty a vývojáře, kteří již mají zkušenosti s vývojem desktopových aplikací pro Windows. Cílení Xamarinu je poznat i v té rovině, že jde o jeden z mála frameworků, který nabízí i zpoplatněnou verzi dodatečných služeb – dříve se jejich speciální analytický systém jmenoval Xamarin Insights, dnes už je ale tento systém zahrnut do Visual Studio App Center. [13]

1.5 Cordova (Apache) – dříve též PhoneGap (Adobe)

Cordova je jeden z tradičních frameworků v oblasti multiplatformního vývoje, jedná se o typický webově hybridní framework, protože v sobě kombinuje UI vrstvu vytvářenou prostřednictvím HTML, CSS a JavaScriptu s vrstvou, která dovoluje nativní funkcionalitu nad rámec toho, co by s běžným webovým rozhraním bylo možné. Nevýhodou ovšem je, že vytvořit webové UI, které vypadá jako nativní mobilní aplikace, je poměrně pracné. [14]

V Cordově nedochází k překladu prvků do nativní úrovně, ba naopak – jejich vykreslování probíhá na úrovni webového wrapperu, který je upraven tak, aby se na obou platformách choval stejně i přes dílčí rozdíly mezi WebView na iOS a Androidu. [14]

Cordova pro nativní funkcionalitu disponuje předpřipravenými pluginy a v případě potřeby dává i možnost vytváření pluginů vlastních. Obecně je Cordova populární v těch oblastech, kde je mobilní vývoj spíše doplňkem k vývoji webové aplikace. [14]

Jedná se o framework, který bude na trhu ještě poměrně dlouho působit s ohledem na množství projektů, které v něm za posledních cca 10 let vznikly, nové projekty ale už vznikají spíše prostřednictvím progresivnějších frameworků. [6]

1.6 Ionic a Capacitor (Drifty)

Ionic začal jako rozšíření frameworku Cordova a dával si za cíl zprostředkovat pro tento framework podporu UI prvků, které vypadají jako ty z nativních mobilních aplikací, což byl zásadní nedostatek Apache Cordova, protože ve své čisté podobě šlo spíše jen o důmyslný webový wrapper. V následných verzích už Ionic využíval Angularu, což je webový framework od Googlu. [15]

V zásadě jde tedy také o webový wrapper, jako tomu je u Cordova, ovšem v tomto případě je využíváno komplexních UI prvků, které vypadají jako nativní uživatelské rozhraní i bez potřeby vynaložit tak velké úsilí při vytváření stylu aplikace. [15]

V nových verzích Ionic už využívá vlastní implementaci pro napojení na nativní funkcionalitu, které říká Capacitor – jedná se v podstatě o úplně nový framework, který je nepřímým následníkem pluginů z Cordova frameworku. I Capacitor funguje jako wrapper pro webovou aplikaci, s tím že do implementace injektuje přemostění mezi webovou a nativní vrstvou – na úrovni sdíleného kódu existuje provider, který teprve z nižší úrovně volá dílčí nativní vrstvu. Capacitor navíc nabízí zpětnou kompatibilitu s pluginy pro Cordova framework. [15]

S ohledem na webový charakter frameworku, Ionic využívá HTML, CSS a JavaScript.

1.7 Qt

Qt je jeden z nejstarších frameworků, jehož vznik dalece předchází existenci mobilních platforem Android a iOS. Podpora pro vývoj na tyto platformy byla doplněna v pozdějších verzích a s ohledem na stáří frameworku jde i o framework, který používá relativně nízko-úrovňový programovací jazyk – tím je C++. [16]

Tento framework podobně jako Flutter využívá vlastní vykreslovací vrstvu a tím pádem i vlastní UI prvky. Jedná se o framework, který má ze všech multiplatformních mobilních

frameworků zdaleka největší platformní zásah – plnohodnotně podporuje desktop, mobilní zařízení i vestavěné Linux systémy. [17]

Mobilní vývoj pro Qt vyžaduje používání Qt creatoru, případně znalost QML značkovacího jazyka pro tvorbu UI. Nejde o úplně běžný framework zaměřený na klientské aplikace, ale ve velkých business systémech, kde je součástí i mobilní implementace, nachází svoje uplatnění. [16]

1.8 Shrnutí srovnání

Tabulka 2 Shrnutí srovnání frameworků [7]...[15]

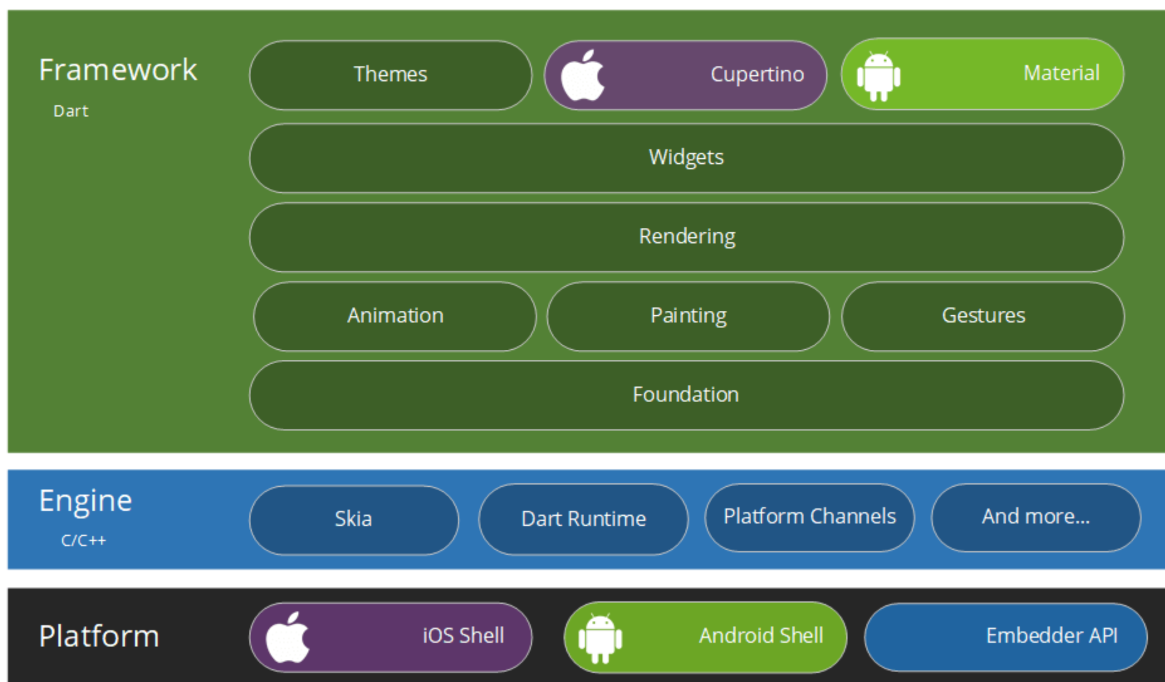
	Flutter	**React Native**	**Xamarin.Forms**	**Cordova**	**Ionic**	**Qt**
Typ	Hybridní	Nativní	Nativní	Hybridní	Hybridní	Hybridní
Prvky UI	Vlastní	Nativní	Nativní	Webové	Webové	Vlastní
Rok vzniku	2017	2015	2011	2009	2013	1995
Hot-reload	Ano	Ano	Ne	Ne	Ano	Ne
Jazyk	Dart	JavaScript	C#	JavaScript	JavaScript	C++

Existují samozřejmě i další frameworky, nicméně v tomto shrnutí jsou zahrnuty především ty nejvýznamnější z hlediska dosavadního tržního využití a potenciálu do budoucna. [6]

Poznámka k jazykům – většina frameworků jistým způsobem podporuje více jazyků, případně při napojování na nativní kód je velmi výhodná znalost i Javy a Obj-C (resp. Kotlin a Swift), v tabulce 2 jsou však uváděny jazyky, pro které je framework od začátku navržen.

2 FLUTTER – PODROBNÁ REŠERŠE

Flutter od Googlu je nová technologie pro multiplatformní vývoj mobilních aplikací, která staví na reaktivním přístupu k toku dat mezi UI vrstvou a vrstvou s logikou samotné aplikace. Přestože operuje na vlastní vykreslovací vrstvě realizované prostřednictvím grafického frameworku Skia, tak nabízí v základu prvky uživatelského rozhraní pro běžné použití, a to jak v designu Androidu (tzv. Material Design), i v designu iOS. Tyto prvky na první pohled vyvolávají dojem nativního rozhraní, ale ve skutečnosti tomu tak není. [5]



Obrázek 4 Základní struktura frameworku Flutter [5]

K základním výhodám frameworku Flutter patří podpora živého obnovení UI při vývoji bez potřeby nového sestavení aplikace (hot-reloading), prakticky nativní výkon a možnost v rámci jednoho kódu vytvořit uživatelské rozhraní, které na obou platformách vypadá a funguje identicky. [18]

Flutter navíc kompletně vzniká v Googlu, čímž se těší podpoře velkého hráče na trhu a buduje kolem sebe poměrně silnou komunitu, takže přestože jde o velmi mladou technologii, v době vzniku této práce už není problém sehnat online podporu pro různé problémy spojené s vývojem. [19]

2.1 Dart

Základním předpokladem pro vývoj ve frameworku Flutter je znalost programovacího jazyka Dart, který také vzniká v Googlu. Dart je typickým reprezentantem moderních programovacích jazyků, které se vyznačují minimalistickou syntaxí a větší volností ve vztahu k tomu, jaké syntaktické konstrukty programátor chce využívat. [20]

To na jednu stranu dává určitou flexibilitu, na straně druhé je ale vhodné, aby se týmy vývojářů v rámci jednoho organizačního celku dohodly na interních syntaktických standardech – aby došlo k zachování konzistentní podoby kódu napříč celým projektem. [21]

2.1.1 Asynchronní programování

Dart podporuje asynchronní programování, které je vhodné pro exekuci dlouho trvajících operací, které by jinak zablokovaly vlákno realizující vykreslování uživatelského prostředí. Z hlediska uživatelského zážitku není vhodné, aby aplikace při načítání dat „zmrzla“ a nijak nereagovala na vstupy až do ukončení načítání. Už jen proto, že takový stav je pro uživatele netransparentní, protože neví, jestli aplikace přestala fungovat, anebo zda má počkat, než dojde k dokončení práce na pozadí. [26]

Pro tyto účely se v Dartu vyskytuje typ Future, který reprezentuje výsledky asynchronních operací, tedy operací, které nejsou prováděny na hlavním (UI) vlákně, místo toho je na jejich vykonání počkáno a teprve až je výsledek dodán vláknem provádějícím daný Future, tak se s tímto výsledkem dále pracuje. [26]

Future je tudíž generická třída Future<T>, kde T reprezentuje datový typ výsledku operace, kterou Future vykonává. Pokud návratová hodnota Future není použitelná a jde jen o vykonání operace, lze Future definovat i jako Future<void>. [26]

Práce s Future je proveditelná dvojím způsobem – jedna z cest je s využitím klíčových slov async a await, typických klíčových slov i pro jiné programovací jazyky s podporou asynchronního programování.

Async definuje asynchronní exekuci kódu a await, že je před dalším vykonáním kódu třeba počkat na výsledek volání. Await je možno používat jen v asynchronních funkcích. Kromě tohoto přístupu lze využívat i Future API, které umožňuje např. jednoduché řetězení dílčích asynchronních volání, či čekání na dokončení operace všech asynchronních volání. [26]

2.1.2 Kompilace

Dart pro finální produkt využívá ahead-of-time (AOT) kompilaci, což znamená, že celý kód je zkompilován před tím, než je vůbec aplikace spuštěna – tím je zajištěna vyšší předvídatelnost výkonu exekuce kódu, na rozdíl od těch jazyků, které využívají just-in-time (JIT) kompilaci za runtime. [21]

Just-in-time kompilace se naopak výrazně hodí v případě, že se často mění kód aplikace (tedy v průběhu vývoje). Dart je unikátní v tom, že podporuje oba druhy kompilace, tudíž lze kombinovat výkonnost finální AOT kompilace, i rychlost průběžné JIT kompilace. Kromě těchto dvou metod kompilace, které se hodí právě pro nasazení v rámci frameworku Flutter, je tu i možnost kompilovat Dart kód do JavaScriptu, díky čemuž lze Dart používat i pro vývoj webových aplikací. Z hlediska nativní kompilace je možné kompilovat jak do x86 kódu, tak do ARM kódu. To zajišťuje, že Dart kód je schopný běžet téměř na všech moderních výpočetních zařízeních. [21]

2.1.3 Generační garbage collector

Dart je od začátku zamýšlen jako jazyk pro reaktivní programování, což je dáno jeho schopností rychle alokovat paměť novým objektům a zároveň se rychle zbavovat objektů nepotřebných prostřednictvím integrovaného generačního garbage collectoru. [22]

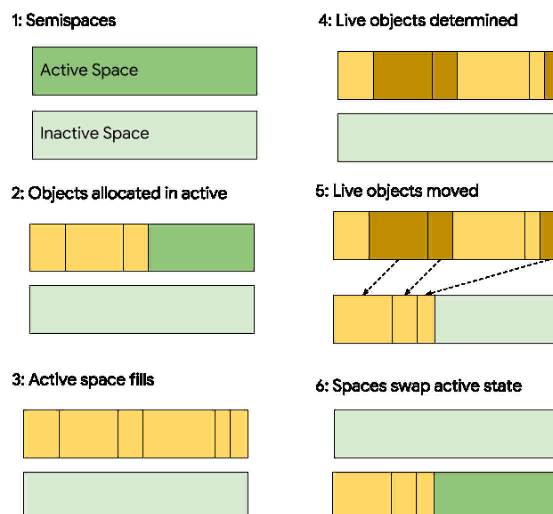
Flutter obecně má vysoké nároky na garbage collector, protože celý koncept reaktivního programování je postavený na tom, že aplikace se nachází v konečných stavech, které lze nahrazovat pouze stavy novými, nikoliv ale měnit stavy existující. Stavem je v tomto případě jeden objekt a ke každému stavu se mohou vázat i desítky objektů na základě stejného principu – změna stavu v rámci aplikace tak může vyvolat potřebu znovu inicializovat i desítky objektů, proto takové nároky na výkon garbage collectoru. U aplikací s komplexnějším uživatelským rozhraním může jít i o stovky, až tisíce objektů v krátkém časovém horizontu. [22]

Samozřejmě se nabízí varianta, že by vývojáři mohli objekty s neměnnou strukturou inicializovat natrvalo a neměnit je i navzdory změně stavu – to ovšem není doporučovaná praktika. Garbage collector v Dartu je implementačně uzpůsoben rapidnímu tempu konstruování a destruování objektů. Garbage collector navíc obsahuje provázání s Flutter enginem, který vyvolá upozornění v případě, že uživatel v aplikaci neprovádí žádnou interakci – nejvhodnější dobou pro vyčištění paměti je totiž okamžik, kdy uživatel od

aplikace neočekává okamžité výstupy. Během těchto neaktivních intervalů dochází také k defragmentaci paměti, což vede na celkovou optimalizaci výkonu aplikace. [22]

2.1.3.1 Young Space Scavenger generace

Garbage collector v Dartu operuje ve dvou generacích, první z nich je Young Space Scavenger. Tato generace funguje tak, že se při čištění zaměřuje na objekty s krátkou dobou životnosti, typicky bezstavové objekty. Přestože jde o proces, který může zablokovat hlavní vlákno aplikace, jde o tak rychlou operaci, že uživatel ve většině případů zpomalení nezaznamená – zejména pak díky zkombinování s výše zmíněným vhodným plánováním, díky němuž celá operace proběhne v okamžiku, kdy uživatel s aplikací aktivně nepracuje. [22]



Obrázek 5 Popis algoritmu za Young Space Scavenger generací garbage collectoru [22]

Ve své podstatě jde o to, že se dostupný paměťový prostor rozdělí na aktivní a neaktivní části. Nově vzniklé objekty jsou alokovány do aktivní části, dokud v ní zbývá paměťový prostor. Jakmile je tento prostor naplněn, proběhne analýza. Ta prochází jednotlivé objekty, začne tedy u jednoduchých adresových proměnných a prozkoumá, zda se odkazují na živé objekty – pokud ano, prozkoumá odkazované objekty. Tímto způsobem se na konci cyklu pro každou strukturu referencí rozhodne, zda jde o aktivní, či nepoužívané objekty – a na základě toho živé objekty přesune do dosud neaktivní části paměti. [22]

Po tomto procesu si obě části paměťového prostoru prohodí aktivní stav a celý proces začíná znovu. Neaktivní objekty budou v dalším cyklu používání přepsány novými. [22]

2.1.3.2 Parallel Marking and Concurrent Sweeping generace

Druhá generace se dá volně přeložit jako „paralelní značkování a souběžné čištění“, což přesně vystihuje probíhající procesy. První generace garbage collectoru se zabývá objekty s kratší dobou životnosti, a než vůbec ke zpracování generace druhé může dojít, objekt musí dosáhnout určitého stáří v rámci běhového cyklu – po této době je přesunut do mark-sweep collectoru. [22]

Druhá generace operuje ve dvou fázích, tou první je vypracování grafu objektů, v němž jsou označeny stále používané objekty. Během druhé fáze dojde k analýze celé paměti, přičemž neoznačené objekty z první fáze jsou odstraněny. Tato forma garbage collectoru se může zablokovat na první (označovací) fázi – pokud nedojde ke změně paměti, nastane zablokování UI vlákna. Z pohledu uživatele tudíž aplikace po dobu běhu této operace nereaguje, proto je velmi důležité, že je důkladně plánována a provázána s Flutter enginem tak, aby byla prováděna jen v okamžicích, kdy s aplikací uživatel nepracuje. [22]

Druhá generace není zdaleka tolik využívána jako ta první, neboť většina objektů v reaktivním programování má krátký životní cyklus, ale i přes pravidelné čištění v první generaci je občas potřeba na úrovni Dart runtime provést kompletní vyčištění, ke kterému slouží právě Parallel Marking and Concurrent Sweeping generace. [22]

2.2 Stream, Sink a StreamController

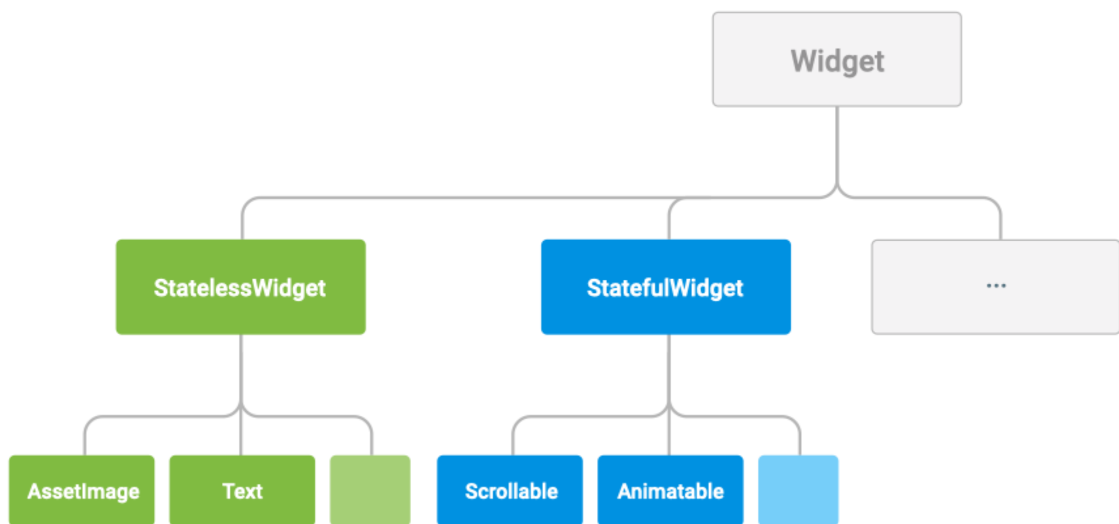
Práce se streamy (proudy) dat je klíčovou součástí toho, jak se ve Flutteru pracuje s upozorněním UI vrstvy o aktualizaci dat na úrovni backendu a naopak. Stream je ve své podstatě nekončící proud dat, kde mohou diskrétně přicházet stále nová data. V kontextu StreamControlleru jde prakticky o výstupní data, na které se v aplikaci dále reaguje. [20]

Sink je potom analogicky vstupní bod pro data, který je předává do streamu. Funkcionalitu lze do jisté míry přirovnat k původu anglického názvosloví – stream je anglicky proud, sink je anglicky umyvadlo. Analogie z reálného prostředí je taková, že když se do „vstupu“ umyvadla naleje voda, vznikne ve „výstupním“ potrubí umyvadla proud vody.

StreamController je v podstatě řídící prvek, který v sobě Stream i Sink zahrnuje – může tak přijímat data a na základě změn v proudu provádět nezbytné operace nad nimi. [20]

2.3 Widget

Jednou ze základních filozofií frameworku Flutter je „vše je Widget“ – co je to tedy Widget? Widget je základní stavební blok aplikace vytvářené v tomto frameworku, který souvisí s uživatelským rozhraním aplikace. Každý widget je neměnnou definicí části uživatelského rozhraní a jde o základní odlišnost od jiných frameworků, kde existují různá View, ViewControllery, rozložení stránek a široká nabídka vlastností různých dle příslušného View – Flutter v tomto ohledu razí filozofii sjednocení celé koncepce za jedním prvkem, a tím je právě Widget. Widgetem je tudíž funkční tlačítko, textové pole, checkbox, ale třeba i míra odsazení od krajů, velikost fontu a další. [23]



Obrázek 6 - struktura widgetů [23]

Widgety se do sebe vnořují a vytvářejí tak poměrně složité hierarchické kompozice, které lze znázornit stromem. Widgetem je ve své podstatě i samotná aplikace – definice základního programu operuje z pozice kořenového widgetu, na který jsou navázány všechny další. [23]

Podobně jako je tomu např. u programovacího jazyka Swift, i v případě Dartu (tudíž i Flutteru) má vývojář možnost se ponořit libovolně hluboko do fungování celého frameworku, protože na rozdíl třeba od .dll knihoven v C#, je zde možnost nahlížet do kódu i systémových částí frameworku během vývoje. Vývojář tak může buď pracovat jen s tím, co mu framework ve výchozím stavu poskytuje, ale v případě potřeby může zasahovat na nižší úrovně. [23]

Každý widget disponuje build funkcí, která definuje, jakým způsobem se widget sestaví v rámci uživatelského rozhraní. Četnost volání této funkce potom záleží dle typu widgetu. [23]

2.3.1 Stateless widget

Stateless widget je takový typ widgetu, který v průběhu svého životního cyklu nepotřebuje měnit svůj vnitřní stav – tedy hodnoty z inicializace si drží po celou dobu své existence v neměnném stavu. [24]

Jedná se typicky o widget, který v sobě zahrnuje další widgety – teprve ty jsou pak složeny ze stavově proměnlivých widgetů, jež definují uživatelské rozhraní více konkrétně. V případě stateless widgetu se build funkce typicky volá jen při vložení widgetu do hierarchie a její výstup záleží jen na vnitřní konfiguraci widgetu. Dále se build funkce může volat v případě, že se změní vnitřní konfigurace widgetu, anebo pokud se změní závislý inherited widget. [23]

V případě častého překreslování stateless widgetu by měl vývojář zvážit přechod na stateful widget, který je optimalizovaný pro aktualizace dle změny vnitřního stavu. [24]

2.3.2 State

State vyjadřuje interní logický stav stateful widgetu, který přímo ovlivňuje průběh sestavování widgetu. Je možné z něj synchronně číst při sestavení, a dá se předpokládat, že se v průběhu životního cyklu widgetu změní. Je zodpovědností implementujícího widgetu, aby správně notifikoval svůj stav o změnách, nejjednodušší technikou takového upozornění je zavolání funkce setState. [23]

Stav se v kódu definuje tak, že je frameworkem vytvořen předefinováním (prováděno klíčovým slovem override) funkce createState ze stateful widgetu, který je následně při buildu vložen do hierarchie widgetů. [23]

2.3.3 Stateful widget

Stateful widget se od stateless liší tím, že má proměnlivý stav. Kvůli tomu je velmi používaný jako widget definující proměnlivá textová pole, či komplexnější stránky s dynamickým obsahem. Sestavovací proces widgetu rekurzivně sestaví postupně celou hierarchii příslušící widgetu, nebo jinými slovy, dokud nedojde k vykreslení celého widgetu. [24]

Instance samotného stateful widgetu je však neměnná, proměnnou hodnotu v rámci životního cyklu Flutter realizuje právě pomocí state, tedy vnitřního stavu. Takový stav se může například přihlásit pro odběr dat ze streamu, tudíž jakákoliv nová data v proudu vyvolají aktualizaci i stavu – a tedy i widgetu. [23]

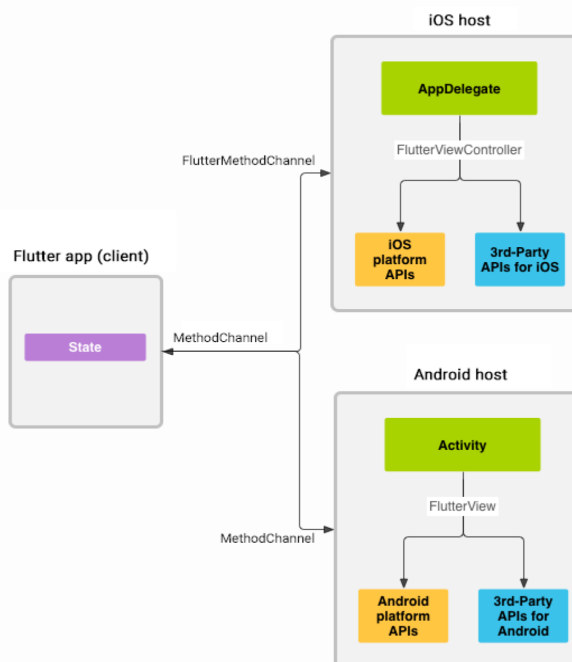
2.3.4 Inherited widget

V momentě, kdy se v aplikaci rozroste struktura widgetů, může nastat komplikace s předáváním dat – tou koncepčně nejjednodušší je předávání si datového kontextu přes konstruktory, nicméně to přináší několik komplikací. Především se ale komplikuje proces aktualizování kódu aplikace, pokud dojde ke změně v rámci struktury dat, případně struktury widgetů, je potřeba přepsat i veškeré předávání mezi konstruktory, čímž vzniká časová zátěž na vývojáře, které lze předcházet. [25]

Právě tomuto problému je předcházeno u inherited widgetu, jehož použitím je frameworku sdělena informace o tom, že by měl projít celý strom widgetů a najít datový kontext na vrcholu, tedy na zastřešujícím inherited widgetu. Klíčovou vlastností inherited widgetu je to, že je po dobu své existence neměnný, tudíž jej lze změnit jen kompletním rebuildem. Proto není vhodné, aby inherited widget obaloval celou obrazovku aplikace, ale byl využíván s rozvahou, neboť nadměrné překreslování aplikace vede ke ztrátě optimálního výkonu. [25]

2.4 Propojení s nativním kódem

V případě, že je potřeba přistupovat k nativní funkcionalitě dané platformy, je potřeba buď využít komponentu, která tuto funkcionalitu již implementuje, případně vytvořit komponentu vlastní – ve frameworku Flutter k tomu slouží systém předávání zpráv mezi nativní a sdílenou vrstvou. [27]



Obrázek 7 Struktura komunikace mezi nativní a sdílenou vrstvou [18]

V praxi to vypadá tak, že vrstva klienta (tedy sdíleného kódu) vyšle zprávu hostitelské vrstvě (tedy nativní vrstvě dané platformy), kde probíhá odposlouchávání příchozích zpráv. Na základě těchto zpráv je provedeno požadované volání nativního kódu, který je napsaný v nativním programovacím jazyce dané platformy, tedy v Javě (či Kotlinu) na Androidu, případně Swiftu (či Obj-C) na iOS. [18]

Po provedení nativního kódu je klientské vrstvě předána odpověď s výsledkem, se kterou už může být libovolně naloženo dále. Předávání zpráv mezi klientem a hostitelem je plně asynchronní kvůli zajištění, aby se kvůli čekání na nativní API nezastavilo uživatelské prostředí aplikace. [27]

Jelikož se na každé platformě liší datové typy, je zajištěna integrovaná binární serializace podobná tomu, jak serializace funguje i v případě JSON formátu – tato serializace a deserializace je od vývojáře odstíněna a jde o vnitřní funkcionalitu, přesto je dobré brát na vědomí, že pokud např. z Dart kódu je vyslána proměnná typu double, na nativní iOS straně je pracováno s NSNumber datovým typem. [27]

2.5 Reflexe, serializace a deserializace JSON dat

Přestože Dart reflexi podporuje, z Flutteru je její podpora odstraněna z toho důvodu, že výsledný balíček s aplikací má při kompilaci odstraněny nepoužívané závislosti, aby velikost výsledné aplikace byla co nejmenší. [18] To je přímo v konfliktu s podporou reflexe, která

umožňuje za chodu programu měnit vnitřní chování programu – nelze dynamicky měnit vnitřní chování programu, pokud při kompilaci dojde k odstranění tříd, které se sice na začátku běhu nevyužívají, ale v průběhu by využívat mohly (díky reflexi). [29]

Reflexe jako taková je vcelku pokročilý programátorský konstrukt, který má ale i naprosto běžné využití – patří sem snadná serializace objektů do JSON formátu a deserializace zpět z JSON formátu na objekt. [28]

Ve Flutteru tak existuje několik cest, jak s tímto omezením pracovat. Jednou z nich je ruční tvorba všech objektů a konstruktorů v obou směrech serializace, která je v principu nejjednodušší, ale zvyšuje množství kódu, které programátor musí napsat pro každou serializovatelnou třídu. [30]

Další metody už vycházejí z možnosti generování kódu, kdy vývojář anotuje základní struktury třídy a nad těmito strukturami zavolá přes terminál funkcionalitu dodanou knihovnou třetí strany, načež tato knihovna potřebnou režii spojenou se serializací vygeneruje automaticky. Např. knihovna json_serializable vygeneruje parciální soubory k již existujícím třídám, tudíž při běhu je s nimi pracováno jako s jedním souborem. Více funkcionality nabízí např. knihovna built_value, která navíc umí generovat kód pro převody na textový řetězec, hashování a další. [30]

2.6 Životní cyklus Flutter aplikace

Životní cyklus Flutter aplikace je popsán výčtem o 4 možných stavech, mezi kterými aplikace může po dobu své existence v operační paměti přecházet. [31]

2.6.1 Inactive

Neboli neaktivní stav aplikace je takový stav, kdy aplikace nemůže reagovat na uživatelské vstupy. Na iOS tento stav nastává např. pokud se zpracovává TouchID požadavek, když telefon vyřizuje hovor, nebo když uživatel vstoupí do hlavního menu iOS, či do ovládacího centra. Na Androidu tento stav nastává v podobných situacích. [18]

2.6.2 Paused

Do paused neboli pozastaveného stavu, aplikace přejde, když není viditelná pro uživatele, tudíž nemůže ani reagovat na vstupy a může maximálně provádět operace na pozadí. Aplikace v tomto stavu by měly programově očekávat, že mohou kdykoliv přejít do suspendovaného stavu. [18]

2.6.3 Resumed

Resumed neboli navrácený (aktivní) stav, je takový stav, kdy aplikace je v popředí a je tak schopna plně využívat systémové prostředky a reagovat na uživatelské vstupy. [31]

2.6.4 Suspending

Suspending neboli suspendovaný stav, je takový stav, kdy vykreslovací engine frameworku Flutter nevolá vykreslovací callbacky Window.onBeginFrame a Window.onDrawFrame. Tento stav se týká jen Androidu, na iOS se momentálně nevyužívá. [18]

3 VHODNÉ NÁVRHOVÉ MODELY (ARCHITEKTURY)

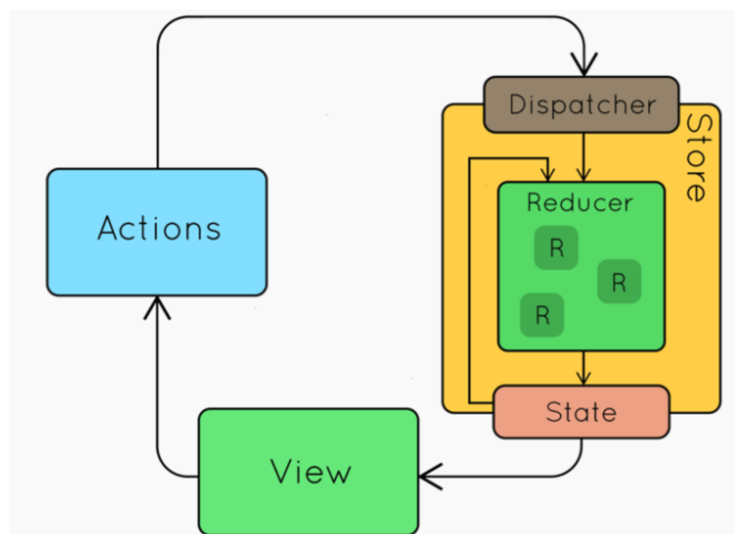
Při návrhu vhodného řešení projektu bylo zvažování několik cest, jak k celému vývoji přistupovat. Flutter je velmi flexibilní technologie z hlediska možnosti nasadit téměř libovolný návrhový vzor, na rozdíl např. od Xamarin.Forms, kde je značně doporučováno používání návrhového vzoru Model-View-ViewModel (MVVM). [13]

Přímo vývojáři z Google nejčastěji na vývojářských konferencích propagují přístup Business Logic Component (dále jen BLoC), který sami vymysleli přímo pro účely implementování mobilních aplikací ve frameworku Flutter. [8]

Dále je zde např. možnost využití architektonického vzoru Redux, Scoped Model, běžného Model-View-Controller (MVC) či úplně vlastního návrhového vzoru dle uvážení.

3.1.1 Redux

Redux vychází z návrhového modelu Flux od Facebooku, se kterým sdílí architekturu, ale do jisté míry ubírá na jeho komplexnosti vyšší mírou abstrakce dílčích struktur. Vyznačuje se především tím, že má jednosměrný tok dat a jedno místo (tzv. Store), kde se drží celkový aktuální stav aplikace. [33]



Obrázek 8 Struktura fungování návrhového vzoru Redux [33]

Tento stav lze získat pomocí jednoho getteru a interakce probíhá prostřednictvím Actions, které se využívá ke zpracování akcí, jež mohou vést na změnu stavu (nikoliv jej však přímo změnit). MiddleWare je další vrstva, která obvykle operuje asynchronně a stará se o provádění akcí, ovšem v tomto případě nemá možnost měnit stav. Poslední částí je tzv.

Reducer, který je běžnou synchronní funkcí vycházející ze stavu a akcí. Reducer je jediná část kódu aplikace, která může změnit stav. [32]

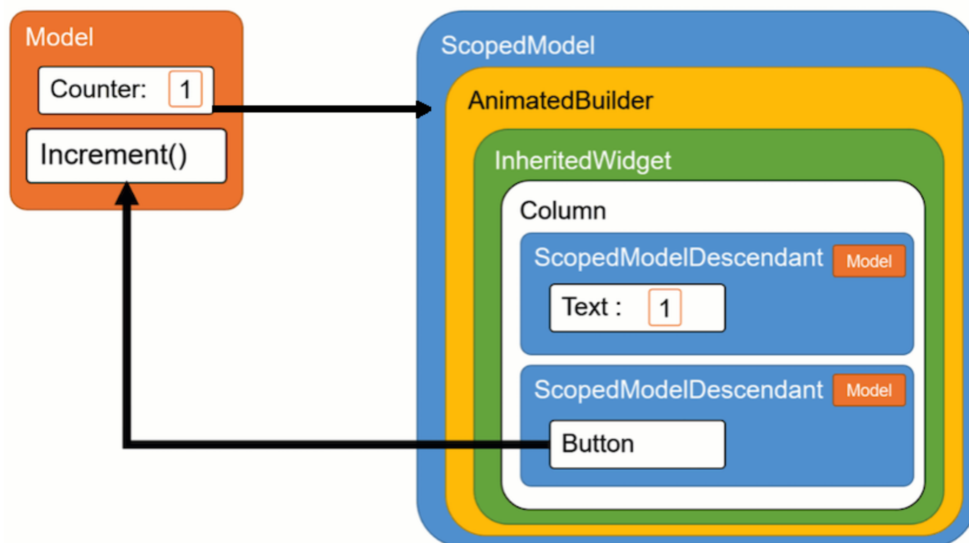
Příklad toku událostí: Uživatel na UI vrstvě stiskne tlačítko, vytvoří se akce a ta se pošle na Store, z něhož se vyvolá jeden z nakonfigurovaných MiddleWare. MiddleWare může provést další zpracování a následně se zavolá Reducer, který přepíše předešlý stav aplikace novým. Na základě nového stavu aplikace je překreslena UI vrstva. [33]

K výhodám návrhového vzoru Redux patří centralizace stavu aplikace a nucení vývojáře jasně dodržovat tok událostí, čímž se značně zpřehledňuje proces debuggování. K nevýhodám lze zařadit to, že u komplexních aplikací může být Store až nepřehledně velký. [32]

3.1.2 ScopedModel

ScopedModel je velmi minimalistický a snaží se spíše jen o rozšíření základní funkcionality frameworku Flutter. Jedná se o sadu nástrojů, které umožňují předávání dat z rodičovského widgetu jeho potomkům. [32]

Dělí se na tři hlavní částí, z nichž první je Model, což je třída, která drží data a byznysovou logiku spojenou s daty (tedy např. načítání), implementuje třídu Listenable, díky níž se může libovolný element přihlásit k poslechu změn třídy Model. Druhou částí je ScopedModel, který je widgetem držícím Model. Umožňuje přístup k Modelu podřízeným objektům, či registraci kontextu coby závislost pro podřízený inherited widget, ze kterého ScopedModel vychází. Třetí částí je widget ScopedModelDescendant, který reaguje na změny v Modelu a znovu se sestavuje pokaždé, když k takovým změnám dojde. [34]



Obrázek 9 Struktura fungování návrhového vzoru ScopedModel [32]

Příklad toku událostí: Uživatel stiskne tlačítko v UI vrstvě, vyvolá se funkce náležící danému tlačítku. V této vrstvě se provedou potřebné aktivity a zavolá se funkce, která upozorní posluchače na změnu. Tu odchytí AnimatedBuilder, ze které ScopedModel dědí a dojde k opětovnému sestavení podřízeného inherited widget. [34]

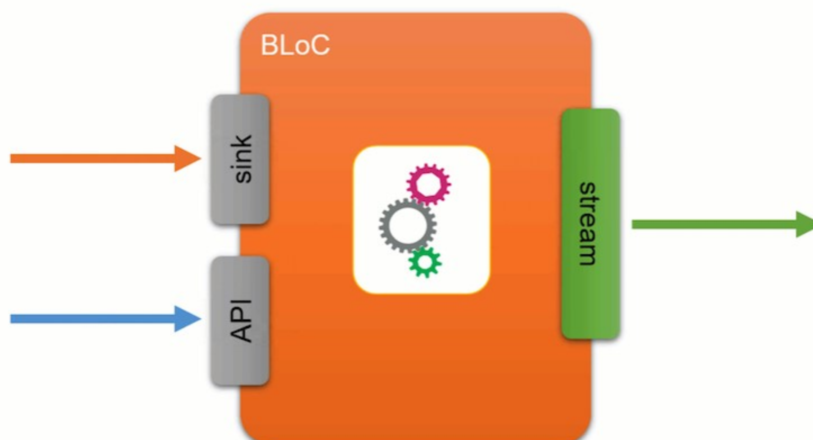
Výhodou ScopedModelu je jeho jednoduchost a snadná přístupnost i vývojářům, kteří nejsou jinak příliš obeznámeni s problematikou streamů. Nevýhodou je, že u komplexnějších aplikací dochází k příliš častému opětovnému sestavování widgetového stromu – to je dáno tím, že není možné zjistit, která část Modelu se změnila, vždy je upozorněno na změnu celého objektu. [34]

3.1.3 Business Logic Component (BLoC)

BLoC vychází ze streamů, a přestože jde o jeden z architektonických přístupů, který ve Flutteru nevyžaduje explicitní použití balíčku, často se používá v kombinaci s RxDart, či jiným pomocným balíčkem, který při práci s proudy zjednodušuje celkové pojetí abstrahováním některých režijních nezbytností. [34]

Základem je výše zmíněný StreamController, který pomocí sinku vkládá data do streamu, který je následně odposloucháván. Dalším podstatným prvkem je widget StreamBuilder, jenž daný stream prostřednictvím StreamSubscription poslouchá a na základě změn znovu sestavuje uživatelské rozhraní. To celé je obvykle na úrovni UI obaleno ve widgetu obecně

označovaném za BlocProvider, který drží referenci na příslušnou BLoC komponentu a zpřístupňuje ji svým potomkům. [34]



Obrázek 10 Struktura fungování návrhového vzoru BLoC [32]

Příklad toku událostí: Uživatel na UI vrstvě stiskne tlačítko, data informující o kliku jsou předána do sinku v příslušné BLoC komponentně, kde jsou zpracována a předána do streamu. Stream je odposloucháván UI vrstvou a dojde k překreslení s novými informacemi. [32]

K výhodám BLoC se řadí velká přehlednost kódu, schopnost sledovat tok událostí a využití streamů umožňuje širokou škálu operací, které by se jinak musely implementovat složitě – patří zde např. debouncer, transformace dat a další. Díky BLoC lze navíc snadno omezit počet opětovných překreslení UI vrstvy, neboť StreamBuilder může reagovat jen na dílčí změny. BLoC také na rozdíl od Redux není limitován koncepcí udržování stavu – může být jeden globální, ale i spousta dílčích. [32]

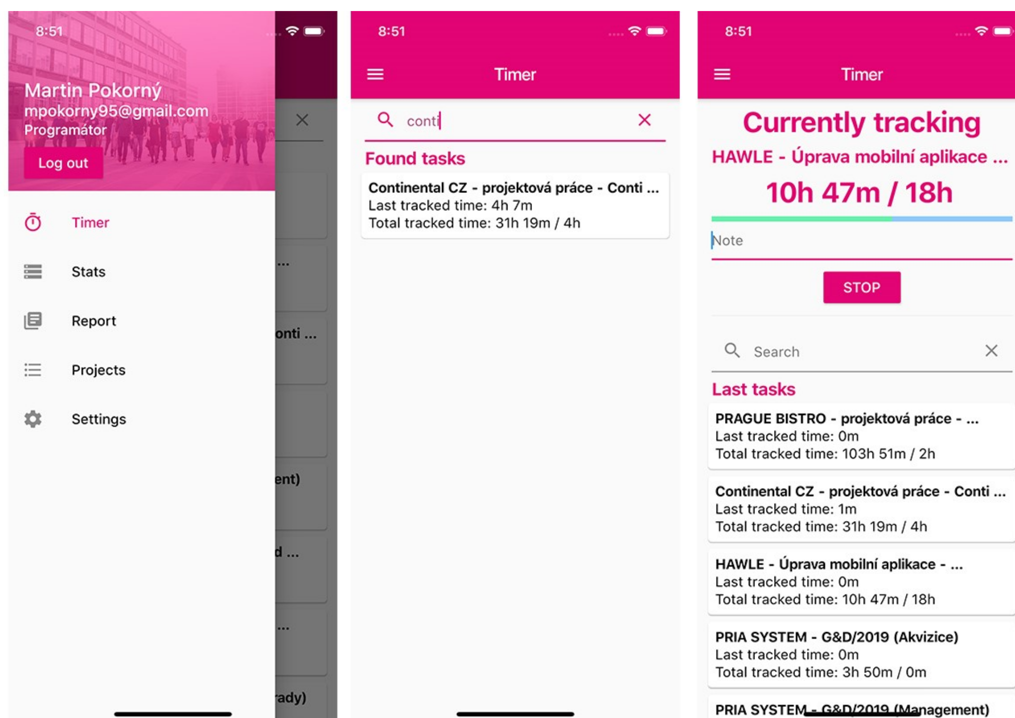
II. PRAKTICKÁ ČÁST

4 DEMONSTRAČNÍ PROJEKT – MOBILNÍ APLIKACE

Demonstračním projektem pro realizaci mobilní aplikace v technologii Flutter je aplikace pro obsluhu systému pro vykazování odpracovaného času a řízení práce dle projektů a klientů, který se používá ve firmě PRIA SYSTEM s.r.o.

Tato aplikace využívá REST API s ověřením OAuth 2.0 pro napojení na existující řešení vykazování odpracovaného času, pracuje s lokálním úložištěm na zařízení pro perzistenci přihlášeného uživatele a v případě, že uživatel má tak nastaveno, upozorňuje uživatele na to, že na daný úkol vykazuje práci už 4 hodiny v kuse.

Aplikace je vyvinuta na základě architektonických poznatků z kapitoly 3 a odlišuje nabídnuté obrazovky dle role uživatele, tedy např. pokud je uživatel administrátorem, má možnost i přistupovat na obrazovku s vytvářením projektů, zatím co pokud je jen běžným zaměstnancem, má přístup jen k aktivním úkolům, na které může vykazovat činnost. Pro všechny uživatele je pak zpřístupněna i možnost nahlížet do statistik a reportů své pracovní aktivity.



Obrázek 11 Screenshoty z bočního menu, vyhledávání v úkolech a vykazování úkolu

Dohromady aplikace disponuje 7 hlavními obrazovkami, přičemž k těm lze přidat ještě další 3 podřízené obrazovky a několik vyskakovacích oken s další interakční logikou. Kořenovou

stránkou pro aplikaci je přihlašovací obrazovka, ze které se dále pokračuje na tzv. Drawer, který nabízí boční vysouvací menu, ze kterého se již dá navigovat v rámci celé aplikace.

4.1 Volba návrhového vzoru pro zpracování projektu

Pro vývoj projektu mobilní aplikace v rámci této diplomové práce byl využit návrhový vzor BLoC, protože jde o aplikaci, která benefituje z možnosti udržovat pro jednotlivé obrazovky dílčí stavy – protože se data postupně dle potřeby stahují ze serveru, bylo by naopak velmi nepraktické je udržovat na jednom globálním místě.

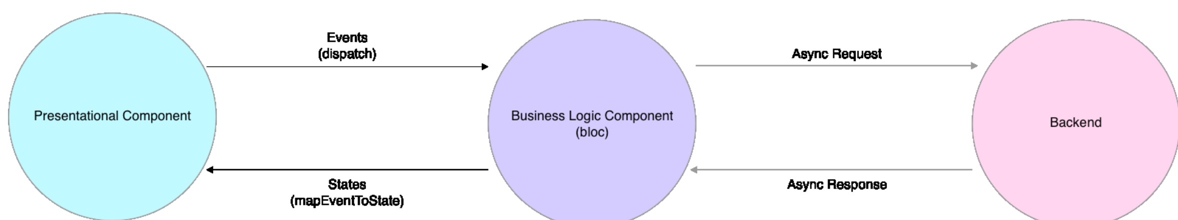
Pro realizaci je použita pomocná knihovna flutter_bloc, která abstrahuje část práce se Streamy a zrychluje tak psaní kódu. K této knihovně je pak dále ještě z hlediska architektury přidán Inversion of Control (IoC) kontejner get_it, který realizuje udržování servisních tříd v paměti a jejich přístupnost odkudkoliv bez nutnosti inicializování nových instancí.

4.1.1 Knihovna flutter_bloc

Tato knihovna vychází z výše uvedených pravidel BLoC návrhového vzoru a je postavena nad knihovnou RxDart, ovšem samotný tok dat pro vývojáře do jisté míry zjednodušuje na události (Event), stavy (State), přechody (Transition), proudy (Stream) a samotný Bloc. [35]

Poznámka – pokud se v této práci hovoří o BLoC, jde o obecný návrhový vzor. Pokud jde o Bloc, jde o konkrétní implementační třídu v rámci knihovny flutter_bloc.

Příklad toku událostí pro BLoC architekturu v implementaci flutter_bloc knihovny je tedy: Bloc komponenta poslouchá proud událostí. Uživatel v UI vrstvě klikne na tlačítko, na Bloc komponentně se vyvolá událost popisující stisk tlačítka, na základě události Bloc komponenta vykoná operace a vrátí nový stav, proběhne přechod ze starého stavu na nový a znovu se sestaví potřebná část UI vrstvy. [35]



Obrázek 12 Zjednodušená struktura návrhového vzoru BLoC v implementaci flutter_bloc knihovny [35]

Jak je ukázáno v obrázku, v implementaci knihovny je rozdělení logiky obecně děleno na tři části – prezentační (UI vrstva), Bloc a backend. Komunikace mezi Bloc a UI probíhá

prostřednictvím propagace změn stavů, naopak posíláním událostí do streamu. Mezi Bloc a backendem pak probíhá oboustranná asynchronní komunikace. Pod backendem je v kontextu mobilní aplikace myšleno buď realizování zpracování lokálních dat v zařízení, případně napojení na server, např. prostřednictvím REST API. [35]

4.1.2 Inversion of Control kontejner get_it

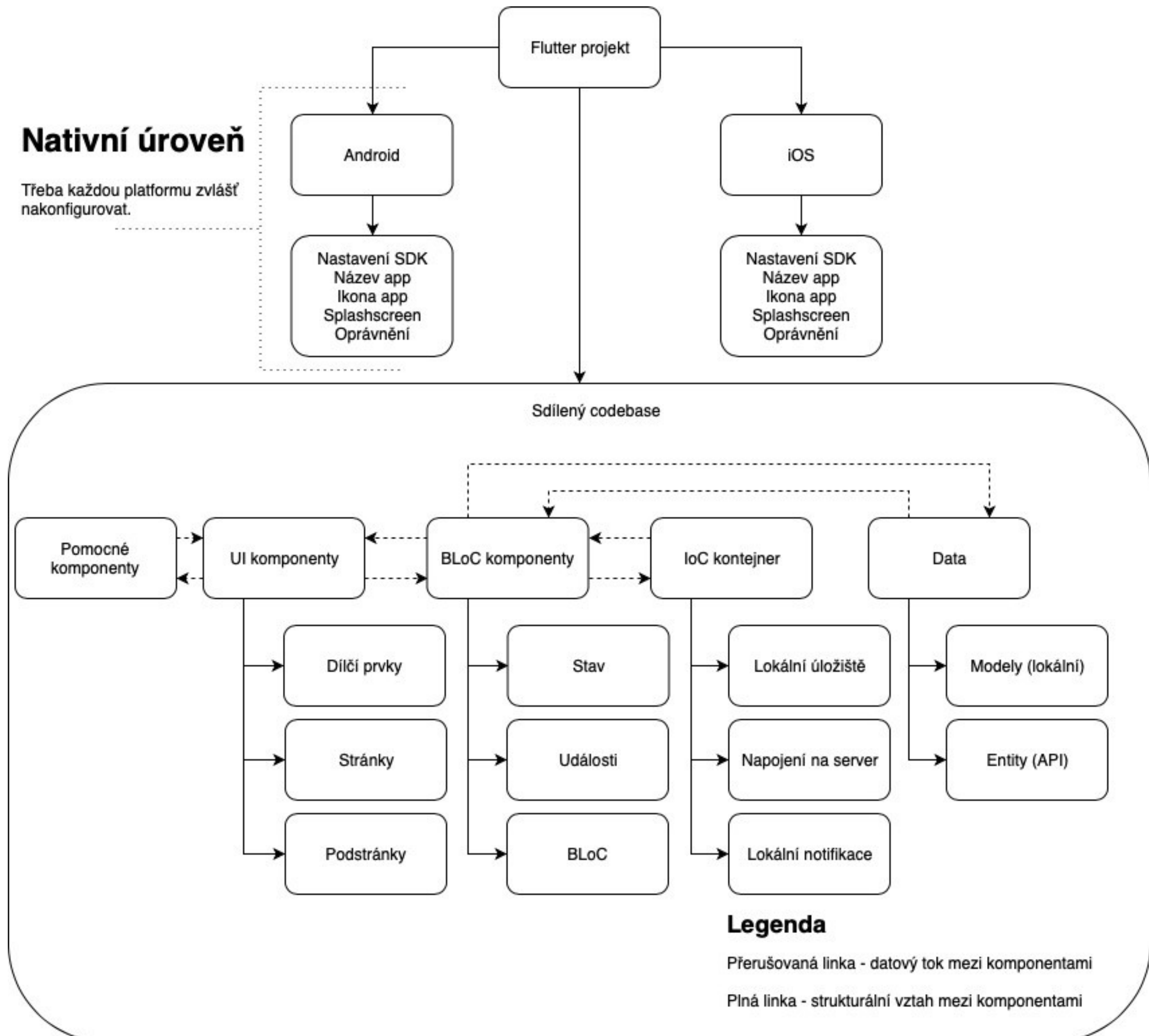
Inversion of Control (IoC) je programátorský princip, kdy dochází k inverzi řídícího toku programu. V principu při vývoji dochází k tomu, že je oddělen interface (v případě Dartu abstraktní bázová třída slouží jako interface) od samotné implementující třídy a zároveň je umožněno odkudkoliv z aplikace přistupovat ke konkrétní implementaci prostřednictvím onoho definujícího interface. [36]

Tato technika činí předávání referencí na servisní třídy přes konstruktory redundantním, stejně tak jako případné vytváření nové instance servisní třídy v každé užívající třídě. Místo toho je třída jednou pod daným interface na začátku runtime zaregistrována, a následně je k ní přistupováno libovolně dle potřeby. [36]

Kontejner get_it tuto funkcionalitu implementuje pro využití ve Flutter aplikacích a v rámci aplikace bude takto využíváno získávání reference na servisní třídy spojené s načítáním a ukládáním dat – a to jak z lokálního úložiště, tak ze serveru. [37]

Jelikož Dart umožňuje registrovat globální proměnné, je do těchto globálních proměnných uložena jedna instance get_it kontejneru na začátku běhu aplikace, se kterou už se dále pracuje odkudkoliv v kódu. [37]

4.2 Návrh architektury aplikace



Obrázek 13 Navržená struktura aplikace realizované v praktické části práce

Přestože Flutter velkou část vývoje sjednocuje, u některých platformních specifik to není technicky možné, a na takové úrovni je potřeba provést konfiguraci (popř. implementaci) pro každou platformu zvlášť. Typicky zde patří oprávnění, ikona aplikace, název aplikace, verze systémového SDK, dle kterého je aplikace sestavena, a další parametry.

V obrázku 13 lze vidět, že máme datovou vrstvu, která je dále rozdělena na data, která se využívají čistě lokálně a na data, která pracují s API. Mezi lokální data patří např. model reprezentující každou stránku aplikace v rámci navigační struktury. Mezi data pracující s API je to např. požadavek na přihlášení uživatele a odpověď na něj.

S datovou vrstvou se v obou směrech pracuje výhradně na úrovni Bloc komponent, které implementují veškerou funkční (byznys) logiku aplikace. Bloc komponenty dle potřeby

mohou přistupovat do IoC kontejneru, odkud získávají přístup k jednotné implementaci napojení na lokální úložiště a server.

Každá Bloc komponenta má pro účely svého operování zvlášť definovány stavy, v jakých se může nacházet, události, jaké v nich mohou nastat a jak na tyto události a stavy mají reagovat.

S UI komponentami komunikují Bloc komponenty tím způsobem, že je upozorňují na změny stavů, na základě nichž se provádí překreslení informací zobrazovaných uživateli, a UI komponenty naopak Bloc komponentám posílají do streamu události reagující na uživatelskou interakci.

UI komponenty dle potřeby využívají ještě pomocné komponenty, kam patří dodatečné pomocné převodníky datových typů, či dat, která byla v nevhodném formátu získána ze serveru – např. převod časového formátu na takový, který je snadno čitelný pro uživatele.

Zdánlivě jednoduchý koncept architektury se ve výsledné aplikaci může rozrůst dle počtu jednotlivých stránek, protože pro každou stránku je potřeba vytvořit minimálně 4 nové třídy – jedna definující stavy, jedna definující události, jedna definující reakce na tyto stavy a události a jedna zobrazující vrstva. Obvykle je ale struktura ještě větší, protože BLoC může vyžadovat dodatečnou servisní vrstvu, případně UI vrstva může potřebovat prvky, které je lepší napsat v oddělených souborech pro snadnou recyklaci kódu.

BLoC tak obecně sice vede na poměrně velké množství kódu, které je potřeba pro obsluhu aplikace napsat, např. když se porovná s možností přímo in-line zápisem toho, co se má v kliku na tlačítko stát, nicméně výměnou za to je kód, který je velmi přehledný, snadno jde odpozorovat, do jakého stavu se aplikace jakým způsobem dostala – a i v případě návratu k vývoji po delší době je snadné se ve vnitřní struktuře aplikace opět zorientovat.

4.3 Struktura Flutter části projektu

Flutter část projektu se nachází ve složce lib, která je dále dělena následujícím způsobem:

- Složka *bloc* obsahuje veškeré realizace BLoC komponent, které jsou dále rozdělené do podsložek dle příslušné obrazovky.
- Složka *common* zahrnuje pomocné třídy, jako jsou konvertory datových typů, či speciální parser mezi doublem a integerem.
- Složka *entity* obsahuje modely spojené s komunikací přes API a dále se dělí:

    - Podsložka *api* obsahuje modely spojené pouze s komunikací přes API, tedy přímé odpovědi či dotazy, které jsou nejdříve zpracovány.
    - Podsložka *common* obsahuje modely získávané z API, ale zároveň jde o modely, které jsou už dále předávány i Bloc či UI vrstvě aplikace.
- Složka *model* disponuje pouze lokálně používanými modely, jako jsou obrazovky nabízené v menu, pojmenované prvky v seznamech, či nastavení aplikace.
- Složka *service* obsahuje servisní služby a dělí se dále do podsložek dle určení, zda jde o třídy vztažené na komunikaci s API, či lokálním úložištěm. Je zde zařazena také služba pro obsluhu lokálních notifikací.
- Složka *ui* obsahuje implementace uživatelského rozhraní, ty se dále dělí:
    - Podsložka *page* obsahuje logiku hlavních stránek.
    - Podsložka *subpage* obsahuje logiku podřízených stránek.
    - Podsložka *view* obsahuje víceúčelové prvky uživatelského rozhraní.

Dále se přímo v kořenovém adresáři celého projektu nachází důležitý soubor *pubspec.yaml*, který v sobě definuje cílovou verzi frameworku Flutter a knihovny, na kterých je závislý. Vývojové prostředí Android Studio je uzpůsobeno tomu, aby tyto balíčky dle definice v souboru stáhlo a nainstalovalo do projektu. Následně k těmto knihovnám lze přistupovat jako k běžnému kódu.

Podstatná je také složka *assets*, která zahrnuje integrované multimediální soubory, případně dle potřeby dokumenty, které jsou potom v aplikaci přístupny. V implementaci lokalizace, která je v aplikaci použita, se zde nachází i textové řetězce ve třech jazykových lokalizacích, to ale není pravidlem.

4.4 Důležité součásti nativních částí projektu

Nativní části projektu jsou v podstatě mobilní aplikace, které po startu inicializují Flutter vrstvu, ze které je pak už operováno po celou dobu, jen s výjimkou v případech, kdy je potřeba zavolat nativní funkcionalitu platformy.

4.4.1 Inicializace frameworku Flutter

Na Android části projektu je tato inicializační logika prakticky skryta tím, že základní MainActivity, která je výchozím bodem všech Android aplikací, dědí z FlutterActivity, která má inicializaci Flutter vrstvy implementovanou.

Na iOS části projektu je situace analogická s ohledem na platformní rozdíly, tudíž – výchozím bodem iOS aplikace je AppDelegate, který v případě Flutter projektu dědí z FlutterAppDelegate. V tom je provedena inicializace Flutter aplikace.

4.4.2 Konfigurace aplikací

Na Androidu je systémová definice aplikace popsána v AndroidManifest souboru, který je formátu XML a obsahuje údaje o aplikaci, jako např. která oprávnění vyžaduje, jaká je její ikona v menu, jaký je její název v menu, či jaký je její unikátní identifikátor. [43]

Kromě toho jsou zde vydefinovány i posluchače notifikací, které lze aktivovat i v reakci na to, že systém vyšle broadcast zprávu o tom, že dokončil boot systému. Aplikace na tuto zprávu mohou reagovat např. nastartováním služby, která hlídá právě lokální notifikace:

```xml
<receiver android:name="com.dexterous.flutterlocalnotifications.ScheduledNotificationBootReceiver">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED"></action>
        </intent-filter>
</receiver>
```

Na iOS se systémová definice aplikace nachází v Info souboru, který je formátu plist, nicméně vnitřní strukturou jde o upravené XML. V jiné variantě zápisu obsahuje prakticky identické informace s AndroidManifest. [43]

Zajímavostí na iOS je to, že pro podporu lokalizace je potřeba vydefinovat podporované jazyky přímo v Info souboru, jinak vznikne problém, kdy i přes implementaci ve Flutter části projektu dochází k používání výchozího jazykového nastavení (tedy angličtiny).

5 REALIZACE DEMONSTRAČNÍHO PROJEKTU

Aplikace realizovaná ve frameworku Flutter má výchozí bod v souboru main.dart, proto je zde kromě zavedení výchozí obrazovky aplikace provedeno i registrování závislostí spojených se servisními třídami, předefinován delegát informující vývojáře o přechodech mezi stavy a ošetření reagování na notifikace.

5.1 Výchozí bod aplikace – soubor main.dart

V souboru main.dart se nachází více tříd, které ale všechny přímo souvisí s koncepčním fungováním aplikace, první z nich je třída AppBlocDelegate.

```
class AppBlocDelegate extends BlocDelegate {
  @override
  void onTransition(Transition transition) {
    super.onTransition(transition);
    print(DateTime.now().toString() + " " + transition.toString());
  }
}
```

Ta je jednoduchá a dědí z BlocDelegate, který vystavuje předpisy funkcí pro reagování na přechody mezi stavy, případně i na stavy chybové. V této konkrétní implementaci jde potom jen o výpis, který do vývojářské konzole ohlašuje přechody mezi dílčími stavy aplikace.

5.1.1 Registrace servisních tříd a spuštění aplikace

Další v pořadí souboru je implementace funkce main:

```
void main() {
  BlocSupervisor().delegate = AppBlocDelegate();

  getIt.registerSingleton<UserRepository>(UserRepository());
  getIt.registerSingleton<ApiService>(ApiService(getIt.get<UserRepository>()));
  getIt.registerSingleton<SettingsRepository>(SettingsRepository());
  getIt.registerSingleton<FlutterLocalNotificationsPlugin>(FlutterLocalNotificationsPlugin());
  getIt.registerSingleton<LocalNotificationsService>(LocalNotificationsService());
  getIt.registerSingleton<NotificationPermissions>(NotificationPermissions());

  runApp(PriaTrackingApp());
}
```

V té je provedena registrace výše uvedeného delegátu, a registrace dílčích servisních tříd do IoC kontejneru, který je uchováván v rámci celého běhu aplikace v globální proměnné – ta je definována ve standardně definovaném souboru globals.dart.

Díky těmto registracím není třeba vytvářet nové instance v každé komponentě vyžadující napojení na API, lokální úložiště, či notifikace. Navíc jak je vidět ze zápisu, třída obsluhující napojení na API má závislost na třídě pracující s úložištěm dat o uživateli – to proto, že u požadavků posílaných na server je potřeba dokládat autenticitu uživatele tokenem, který je uložen lokálně pro zachování perzistence přihlášení mezi jednotlivými instancemi celé aplikace.

V rámci registrace závislostí je tak třeba dbát na to, aby bylo dodrženo správné pořadí registrování dílčích tříd – pokud třída ApiService má závislost na UserRepository, musí být UserRepository zaregistrována jako první.

Na posledním řádku main funkce se nachází volání runApp, která provede inicializaci kořenového prvku ve struktuře widgetů – tento kořenový prvek je potřeba dodat v parametru funkce a v tomto případě je to nová instance třídy PriaTrackingApp.

5.1.2 Kořenový widget aplikace

Jak bylo uvedeno v teoretické části, filozofií Flutteru je „vše je widget“, a to platí i pro celou strukturu aplikace. Kořenovým prvkem aplikace je tudíž v tomto případě třída dědící z třídy StatefulWidget.

```
class PriaTrackingApp extends StatefulWidget {
  PriaTrackingApp({Key key}) : super(key: key);
  @override
  State<PriaTrackingApp> createState() => _PriaTrackingAppState();
}
```

Ta je v zásadě velmi jednoduchá, neboť veškerá komplexní realizace se odehrává až v implementaci jejího vnitřního stavu, který je popsán ve třídě _PriaTrackingAppState.

Podtržítko na začátku názvu třídy značí, že jde o třídu přístupnou jen interně ze souboru main. Její instance tudíž nelze vytvářet mimo tento soubor. Ten je hned po inicializaci widgetu přiřazen v předefinování funkce createState.

5.1.3 Kořenový state (stav) aplikace

Pro každou implementaci třídy dědící z třídy State, jsou nejdůležitější dvě funkce, které lze předefinovat implementací – tou první je initState, která má na starost inicializaci počátečního stavu widgetu (v tomto případě celé aplikace), a ta druhá je build, která sestavuje

widgetový strom na základě aktuálního stavu, jinými slovy – inicializuje a aktualizuje podobu uživatelského rozhraní aplikace podle aktuálního stavu.

```
@override
 void initState() {
  AndroidInitializationSettings initializationSettingsAndroid =
  AndroidInitializationSettings('app_icon');
  IOSInitializationSettings initializationSettingsIOS = IOSInitializationSettings(
    onDidReceiveLocalNotification: onDidRecieveLocalNotification);
  InitializationSettings initializationSettings = InitializationSettings(
    initializationSettingsAndroid, initializationSettingsIOS);
  getIt.get<FlutterLocalNotificationsPlugin>().initialize(initializationSettings,
    onSelectNotification: onSelectNotification);
  _loginBloc = LoginBloc();
  _loginBloc.dispatch(LoginInitialization());
  _subscription = Connectivity()
    .onConnectivityChanged
    .listen((ConnectivityResult result) {
   if (result == ConnectivityResult.none) {
    _loginBloc.dispatch(GoneOffline());
   } else {
    _loginBloc.dispatch(LoginInitialization());
   }
  });
  super.initState();
 }
```

V inicializaci stavu je potřeba odlišit konfiguraci iOS a Android notifikací, která se liší v důsledku rozdílů jednotlivých platforem, teprve po inicializaci těchto konfigurací je do IoC kontejneru vložena inicializovaná služba obsluhující notifikace, které jsou i předány reference na funkce obsluhující práci s notifikacemi – viz níže. Po notifikacích probíhá inicializace výchozího bodu aplikace z pohledu logiky rozhraní, tím je komponenta LoginBloc, která obsluhuje přihlašovací logiku.

Ještě před zavoláním bázového super.initState() pro dokončení inicializační logiky je zprovozněn Connectivity plugin, který má na starosti hlídání, zda je zařízení připojeno k internetu. Aplikace svou povahou nemá funkční smysl bez připojení na obsluhující server, tudíž závisle dle stavu připojení je případně do kořenového widgetu propagována událost ohlašující změnu konektivity.

Předefinování funkce build je natolik obsáhlé, že její uvedení v tomto textu bude rozděleno na dílčí části. Funkce build přijímá jako parametr BuildContext, který jí dává informace o zasazení widgetu ve widgetovém stromě, velikosti obrazovky a další parametrech spojených s vykreslováním uživatelského rozhraní.

```
@override
Widget build(BuildContext context) {
  return BlocProvider<LoginBloc>(
    bloc: _loginBloc,
    child: MaterialApp(
      debugShowCheckedModeBanner: false,
      localizationsDelegates: [
        FlutterI18nDelegate(useCountryCode: false, fallbackFile: "en.json"),
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate
      ],
      supportedLocales: [Locale("en"), Locale("cs"), Locale("sk")],
```

Celá build funkce v podstatě vrací jeden objekt, kterým v tomto případě je instance typu BlocProvider, který za generický parametr dostává LoginBloc. BlocProvider je třída, která má na starost propojení s danou komponentou s funkční (byznys) logikou, v tomto případě již v initState vytvořená instance LoginBloc. V reakci na změny stavu v tomto Bloc prvku potom provádí vykreslovací logiku popsanou kódem níže.

Základním potomkem je MaterialApp, což je třída definující mobilní aplikaci vycházející ze základních designových prvků Material Designu, tedy designu, který je nativní pro Android aplikace. Zahrnuje v sobě podporu navigování mezi stránkami aplikace, lokalizaci a další funkce.

Lokalizace funguje prostřednictvím těchto delegátů, kde je potřeba předem vydefinovat podporované jazyky – parametr supportedLocales, a také delegáta, který funguje coby dodavatel samotných překladů. Tito delegáti mohou fungovat různě závisle dle implementace, přičemž v rámci tohoto projektu byl zvolen balík FlutterI18n, který splňuje lokalizační standardy požadované po moderním software. [45] Důvodem pro zvolení tohoto balíku bylo především jeho jednoduché implementování prostřednictvím textových řetězců skrytých za unikátními klíči v JSON souborech, které jsou k aplikaci přiložené formou assetu. Parametr debugShowCheckedModeBanner v demonstračním kódu vypíná překryvný štítek označující aplikaci jako běžící v debug režimu.

```
theme: Theme.of(context).copyWith(
      platform: TargetPlatform.iOS,
      primaryColor: AppTheme.primaryColor,
      primaryColorDark: AppTheme.primaryDarkColor,
      accentColor: AppTheme.accentColor,
      textTheme: TextTheme(
        title: TextStyle(
          fontSize: 18.0,
```

```
        fontWeight: FontWeight.bold,
        color: AppTheme.primaryTextColor),
```

Dalším důležitým parametrem MaterialApp je theme, kde lze dosadit základní tematické schéma. To může být zcela vlastní, nebo může vycházet z některého předdefinovaných a to upravovat. V tomto případě je využito jako základní schéma iOS, neboť v době vývoje aplikace Android schéma obsahovalo chybu, která způsobovala pády u kontextového menu v původním designu.

Dále už jsou jen nadefinovány různé fonty a barvy, které vychází z vlastní statické třídy AppTheme. Takto definovaných stylů je spousta, ale syntakticky jsou totožné s ukázkou, tudíž byl zbytek z textu pro zkrácení odstraněn. Definice dílčích stylů textových řetězců se řídí podobným systémem, jako tomu je u webových aplikací, tedy dělí se na title, body, button a další i s číslováním dle úrovně.

```
home: BlocBuilder<LoginEvent, LoginState>(
      bloc: _loginBloc,
      builder: (BuildContext context, LoginState state) {
        if (state is LoginStateNoUser) {
          return LoginPage();
        }
        if (state is LoginStateUser) {
          return DrawerPage();
        }
        if (state is LoginStateOffline) {
          return Scaffold(body: Center(child: Text("Device is offline")));
        }

        return Scaffold(body: LoadingWidget());
      },
    )
```

Posledním parametrem MaterialApp je parametr home, který přijímá BlocBuilder s generickými parametry LoginEvent a LoginState. Zde se již dostáváme k samotné podstatě použití knihovny flutter_bloc. BlocBuild dostává referenci na svoji příslušnou komponentu s byznys logikou a na základě změn stavů je volána funkce předána parametru builder.

Zde se podle stavu login komponenty aplikace rozhoduje:

- Uživatel je odhlášený – zobrazit přihlašovací obrazovku.
- Uživatel je přihlášený – zobrazit hlavní menu aplikace.
- Aplikace je offline – zobrazit zprávu o tom, že aplikace v offline režimu neoperuje.
- V jakémkoliv jiném stavu (např. probíhá-li inicializace) zobrazí načítací obrazovku.

V main souboru je obslouženo ještě i reagování na příchozí notifikace v případě, že je aplikace v popředí (pokud je na pozadí, zobrazí se systémová notifikace). Chování je rozděleno na dvě funkce.

```
Future onDidRecieveLocalNotification(
    int id, String title, String body, String payload) async {
  await showDialog(
    context: context,
    builder: (BuildContext context) => CupertinoAlertDialog(
      title: Text(title),
      content: Text(body),
      actions: [
        CupertinoDialogAction(
          isDefaultAction: true,
          child: Text('Ok'),
          onPressed: () async {
            debugPrint('notification shown: ' + payload);
          },
        )
      ],
    ),
  );
}
```

První funkcí je onDidReceiveLocalNotification, která kdekoliv v aplikaci vykreslí vyskakovací dialog informující o notifikaci. Jelikož notifikace mají v aplikaci ryze informativní charakter, veškerá reakce na potvrzovací tlačítko je jen o propsání potvrzení akce do vývojářské konzole.

Z uživatelského hlediska se tedy jen zavře dialog, to je zajištěno tím, že to tlačítko je označeno parametrem isDefaultAction.

```
Future onSelectNotification(String payload) async {
  if (payload != null) {
    debugPrint('notification payload: ' + payload);
  }
}
```

Druhou funkcí je onSelectNotification a ta reaguje na situaci, že uživatel klikne na systémovou notifikaci, přičemž platí výše uvedené, tedy že notifikace jsou ryze informativního charakteru a nevyžadují další uživatelskou interakci. V opačném případě by ale šlo do aplikace přidat další chování.

5.2 Entity, serializace, deserializace, JSON

Přestože dílčí entity nejsou koncepčně složité, aplikace je s ohledem na komplexní strukturu dat na straně serveru vcelku rozsáhlá – obsahuje přes 20 entit spojených s komunikací se serverem, a ještě pár lokálních modelů, které jsou využívány čistě pro vnitřní logiku mobilní aplikace.

Entity nabývají různé komplexnosti, příkladem zastupujícím všechny možné problémy spojené s jejich realizací, je např. třída UserReport, která v aplikaci reprezentuje data používaná v reportu uživatele z hlediska pracovního výkonu za určité období.

```
class UserReport {
  Hours hours;
  Expenses expenses;
  Summary summary;
  List<ProjectReport> projects;
  ProjectSum projectSum;
  UserReport(this.hours, this.expenses,
                      this.summary, this.projects, this.projectSum);
  factory UserReport.fromJson(Map<String, dynamic> json) =>
     _userReportFromJson(json);
}
```

Samotná třída v sobě implementuje další entity:

- Hours reprezentuje odpracované hodiny rozdělené dle různých kategorií (interní či klientský náklad, celkový počet, dovolené, a další).
- Expenses reprezentuje náklady na pracovní činnosti zaměstnance, opět děleno do několika kategorií.
- Summary obsahuje shrnující informace o jednotlivých pracích pro klienty, či obecné činnosti v rámci fungování společnosti. Summary se dále větví do dalších polí dat, celková datová struktura tak je už vcelku komplexní.
- List<ProjectReport> obsahuje reporty o projektech odpracovaných za dané období.
- ProjectSum obsahuje celkové součty dat dle daných kategorií.

Tyto entity lze inicializovat dvěma způsoby, buďto prostřednictvím standardního konstruktoru, anebo pomocí speciálního factory konstruktoru, který se využívá pro načtení dat z JSON souboru. Tento konstruktor je implementován ve stejném souboru, jako třída entity a v případě potřeby jej může doplnit i inverzní funkce, která existující data převede do JSON formátu.

```
UserReport _userReportFromJson(Map<String, dynamic> json) {
  Hours hours;
  Expenses expenses;
  Summary summary;
  List<ProjectReport> projects;
  ProjectSum projectSum;
  if (json.containsKey("projects")) {
    projects = (json["projects"] as List)
        .map((item) => ProjectReport.fromJson(item))
        .toList();
  }
  if (json.containsKey("hours")) {
    hours = Hours.fromJson(json["hours"]);
  }
  if (json.containsKey("expenses")) {
    expenses = Expenses.fromJson(json["expenses"]);
  }
  if (json.containsKey("summary")) {
    summary = Summary.fromJson(json["summary"]);
  }
  if (json.containsKey("projectsSum")) {
    projectSum = ProjectSum.fromJson(json["projectsSum"]);
  }
  return UserReport(hours, expenses, summary, projects, projectSum);
}
```

Factory konstruktor dostane do parametru mapu String a dynamického datového typu, to je z toho důvodu, že hodnotou skrytou za daným textovým klíčem může být libovolný standardní datový typ, od double, přes int, až po String, či bool.

Pokud je podřízená entita jednoprvková, je její převod z JSON formátu přímočarý – zavolá se její factory konstruktor, který je podobný výše uvedenému a který vrátí výsledný objekt. Situaci komplikuje až pole prvků, např. zde uvedený List s generickým parametrem ProjectReport. V takovém případě je třeba nejdříve příslušný JSON převést na List, následně všechny jeho prvky mapovací funkcí (či jinou formou cyklu) převést na instance typu ProjectReport a teprve toto pole převést na finální List o daném datovém typu.

5.3 Service (servisní) třídy

Za servisní třídy se obecně v kontextu tohoto projektu označují třídy, na kterých jsou závislé třídy s byznysovou logikou aplikace, ale jedná se o závislost pouze jednosměrnou. Servisní třídy tak fungují zcela nezávisle na funkční byznys logice a jen jí poskytují specifickou funkcionalitu, či přístup ke konkrétním datům, které vrací na základě konkrétních požadavků.

5.3.1 Třídy pro obsluhu lokálního úložiště

Pro realizaci lokálního úložiště je použita knihovna flutter_secure_storage, která na iOS využívá ukládání dat do Keychain, tedy do systémové šifrované knihovny. Na Android je potom využíváno šifrování přes AES s klíčem, který je uložen v Keystore, což je systémové bezpečné úložiště pro šifrovaná data na Androidu.

Koncept práce s lokálním úložištěm je vytvořen takovým způsobem, že jej zastřešuje servisní třída DataRepository, která obsahuje konečný klíč a funkce pro načítání, odstraňování a zápis dat pod daným klíčem. Tato třída je zamýšlena jako výchozí bod, ze které by měly všechny další třídy implementující lokální úložiště nad konkrétními daty vycházet. Tudíž DataRepository definuje samotnou práci s daty, a ostatní Repository třídy již logiku spojenou se zpracováním těchto dat. V realizaci aplikace jde tedy o data spojená s uživatelským účtem a nastavení.

5.3.1.1 Třída UserRepository

Třída má v sobě napevno zmíněný klíč:

```
class UserRepository {
  final dataRepository = DataRepository("User");
```

Tím je zajištěno, že kdykoliv je volána instance UserRepository, pracuje se nad jednou sadou dat popsanou klíčem. Nemůže se stát, že by programátor v jiné části kódu uvedl špatný klíč, a tím pracoval nad nekorektní (či neexistující) sadou dat.

```
Future<User> tryLogin(
      {@required String username, @required String password}) async {
    TokenResponse tokenResponse =
        await getIt.get<ApiService>().loginUser(username, password);
    if (tokenResponse != null && tokenResponse.accessToken.isNotEmpty) {
      User userResponse = await getIt
          .get<ApiService>()
          .getCurrentUser(tokenResponse.accessToken);
      if (userResponse != null) {
        userResponse.token = tokenResponse;
        await saveUser(userResponse);
        return userResponse;
      }
    }
    return null;
  }
```

Ve funkci tryLogin je vidět realizace asynchronního volání, kdy se využívá typ Future s generickým typem User, tudíž po dokončení vykonávání této funkce v sekundárním vlákně (k tomu vykreslujícímu UI vrstvu) dojde k navrácení uživatele v případě, že je proces přihlašování uživatele na serveru úspěšný.

Sekvence kódu je postavena tak, že nejdříve přes API servisní třídu pošle požadavek s uživatelským jménem a heslem. Pokud dostane odpověď s platným tokenem, pošle na API další požadavek o získání bližších informací o aktuálním uživateli identifikovaném daným tokenem.

Pokud i tato data jsou úspěšně získána, je uživatel považován za přihlášeného, data se uloží na lokální úložiště a aplikace předá zpět do byznys logiky data o přihlášeném uživateli.

Třída obsahuje další logiku spojenou s kontrolou platnosti tokenu, ukládáním či odstraněním uživatele z lokálního úložiště.

```
Future<User> loadUser() async {
  try {
    String jsonString = await dataRepository.loadData();
    if (jsonString != null && jsonString.isNotEmpty) {
      Map mappedUser = jsonDecode(jsonString);
      User user = User.fromJson(mappedUser);
      return user;
    }
  } on Exception {
    print("[UserRepository] user structure invalid, logging out");
    await logout();
  }
  throw Exception("[UserRepository] loading failed");}
```

V případě načítání uživatele jde o opět o asynchronní volání, které vrací uživatele jako objekt. V tomto případě se servisní třída pokusí o načtení dat z lokálního úložiště a pokud kdekoliv v průběhu selže, vypíše do vývojářské konzole chybové hlášení a zavolá funkci pro odhlášení, která odstraní pravděpodobně neplatná data v lokálním úložišti.

Data jsou ukládána formou JSON a pokud tedy převod z JSON do objektu proběhne v pořádku, je vrácený přihlášený uživatel. Důvod, proč funkce je obalena do try-catch bloku, a následně ještě na konci vyhazuje výjimku je ten, že se jinak mohlo stát, že načítání dat sice selhalo a odstranění bylo úspěšné, nicméně informace o tom již neprošla do byznys logiky. Proto v byznys logice je ještě další try-catch blok, který hlídá úspěšnost několika kroků spojených s přihlašováním uživatele po spuštění aplikace – k těm patří i načtení z lokálního úložiště.

Další funkce používané v ostatních repository třídách, tedy UserRepository i v SettingsRepository, fungují analogicky k uvedeným příkladům a liší se jen ve specifické implementaci vycházející z jiné povahy dat.

5.3.2 Třída pro obsluhu lokálních notifikací

Přestože se v rámci aplikace notifikace momentálně zapínají jen z obrazovky nastavení a jejich plánování probíhá jen z obrazovky s časovačem, pro případné rozšiřování funkcionality spojené s notifikacemi je lepší od samého začátku provádět implementaci v nezávislé servisní třídě. Proto je celá notifikační logika oddělena do samostatné třídy LocalNotificationsService.

Ta obsahuje funkci pro zrušení všech naplánovaných notifikací:

```
void cancelTrackingNotification() async {
   _notifications.cancelAll();
 }
```

S tím, že pokud by v budoucnu v aplikaci existovalo více druhů notifikací, je možné rušení dílčích typů oddělit. V aktuální implementaci ale postačuje varianta se zrušením všech notifikací, neboť existuje jen jeden typ notifikace. A ten je plánován v následující funkci:

```
void setupTrackingNotification(Timesheet timesheet) async {
   _notifications.cancelAll();
   final settings = await _settingsRepository.loadSettings();
   if (settings.trackingNotification) {
     DateTime scheduledNotificationDateTime =
        DateTime.now().add(Duration(hours: 4));
     AndroidNotificationDetails androidPlatformChannelSpecifics =
        AndroidNotificationDetails("tracking_notifications",
           "Tracking notifictaions", "Showing notifications about tracking", icon: "@drawable/app_icon",
importance: Importance.Max, priority: Priority.Max);
     IOSNotificationDetails iOSPlatformChannelSpecifics =
        IOSNotificationDetails();
     NotificationDetails platformChannelSpecifics = NotificationDetails(
        androidPlatformChannelSpecifics, iOSPlatformChannelSpecifics);
     await _notifications.schedule(
        0,
        'You are still tracking',
        'You are still tracking ' + timesheet.about,
        scheduledNotificationDateTime,
        platformChannelSpecifics);
   }
 }
```

Notifikace dostane datový model aktuálně vykazovaného úkolu, zkontroluje pomocí servisní třídy pro perzistenci nastavení, zda jsou notifikace povolené – pokud ano, provede registraci notifikace na čas 4 hodiny od aktuálního času. Tento čas je zvolen s ohledem na běžnou 8hodinovou pracovní dobu – pokud je vykazována práce na úkol po 4 hodiny v kuse, je uživatel upozorněn.

5.3.3 Třídy pro obsluhu napojení na server prostřednictvím API

Třídy spojené s napojením na server jsou celkem tři, přičemž ta první z nich je AppHttpClient, dědí z výchozího HTTP klienta BaseClient a k požadavkům automaticky připojuje nezbytný token, prostřednictvím kterého je uživatel vůči serveru autentifikován.

```
class AppHttpClient extends http.BaseClient {
  final String accessToken;
  final http.Client _inner = http.Client();
  AppHttpClient(this.accessToken);

  Future<StreamedResponse> send(BaseRequest request) async {
    if (accessToken != null && accessToken.isNotEmpty) {
      request.headers['Authorization'] = "Bearer " + accessToken;
    }
    return await _inner.send(request);
  }
}
```

V ukázkovém kódu je vidět, že v podstatě se využívá standardního vnitřního HTTP klienta, ovšem ke každému požadavku je přidán ještě hlavičkový parametr Authorization, doplněný o autentifikační token. Tento token uživatel získá při prvním přihlášení spolu s tokenem pro obnovení dle standardu OAuth2, takže pokud běžný token vyprší, aplikace vůči serveru pošle požadavek pro dodání nového tokenu na základě obnovovacího tokenu. Pokud je vše v pořádku, aplikace obdrží token nový. V případě problému je uživatel odhlášen.

Druhá třída spojená s API je nazvaná ApiConst, je velmi jednoduchá a jen uchovává dílčí textové řetězce potřebné pro skládání celých URL adres, přes které probíhá komunikace se serverem.

Např. pro vytvoření požadavku na získání dat o aktuálně přihlášeném uživateli jsou potřeba následující dva řetězce:

```
static String baseUrl = "http://test-tracking-api.pria.cz/";
static String getCurrentUser = "api/current-user";
```

Ty jsou následně použity v třídě ApiService v příslušné funkci:

```
Future<User> getCurrentUser(String token) async {
  AppHttpClient appHttpClient = AppHttpClient(token);
  String url = ApiConst.baseUrl + ApiConst.getCurrentUser;
  final currentUserResponse = await appHttpClient.get(url);
  if (currentUserResponse.statusCode == 200 &&
      currentUserResponse.body.isNotEmpty) {
    final currentUserResponseMap = jsonDecode(currentUserResponse.body);
    CurrentUserResponse currentUser =
        CurrentUserResponse.fromJson(currentUserResponseMap);
    url = ApiConst.baseUrl + ApiConst.getUser + currentUser.id.toString();
    final http.Response response = await appHttpClient.get(url);
    if (response.statusCode == 200 && response.body.isNotEmpty) {
      final userResponse = jsonDecode(response.body);
      return User.fromJsonNoToken(userResponse);
    }
  }
  return null;
}
```

Tato funkce asynchronně nejdřív inicializuje klienta s aktuálním přístupovým tokenem, následně sestaví výslednou URL adresu z výše uvedených textových řetězců, kterou předá GET požadavku nad inicializovaným klientem. Pokud je odpověď serveru v pořádku (kód 200) a tělo odpovědi není prázdné, proběhne načtení dat z JSON formátu do formátu UserResponse.

Z UserResponse je získáno unikátní identifikační číslo uživatele, které je použito v dalším requestu pro získání informací o uživateli dle id, pokud i zde je odpověď v pořádku, funkce vrací zpět do hlavního vlákna výsledného uživatele reprezentovaného instancí třídy User.

Tímto způsobem jsou realizovány téměř všechny serverové požadavky, rozdíl je jen v tom, že jiné druhy požadavků, typicky např. POST či PUT požadavky, mají ještě definované tělo a jiný formát dat. Toho je dosaženo např. následujícím způsobem:

```
final body = jsonEncode(<String, dynamic>{"internal": false, "string": search});
final headers = <String, String>{'Content-type': 'application/json','Accept': '*/*',};
```

Tělo i hlavička jsou doplněny k požadavku a zbytek logiky operuje analogicky.

Speciální výjimku v požadavcích na server představují ještě požadavky, kde je potřeba data dalším dodatečným způsobem zpracovat. Např. seřadit ve vztahu k pořadí, v jakém jsou načítané, toho může být docíleno prostřednictvím mapovací funkce, která je přímo součástí jazyku Dart:

```
if (response.statusCode == 200 && response.body.isNotEmpty) {
    final json = jsonDecode(response.body);
```

```
    List<Project> list =
      (json as List).map((item) => Project.fromJson(item)).toList();
    list.sort((a, b) => b.id.compareTo(a.id));
    return list;
}
```

V tomto uvedeném případě je JSON objekt převeden na List objektů, přes které je mapována funkce, která je převede na instance třídy Project. Tento list instancí třídy Project je následně řazen podle parametru id.

5.4 Bloc třídy

Než přejdeme k demonstraci tří uživatelského rozhraní, je potřeba nejdříve předvést tématiku Bloc tříd, které v tomto návrhovém vzoru realizují veškerou funkční logiku výsledné mobilní aplikace. Bloc třídy se v implementaci knihovny flutter_bloc dělí dle příslušné obrazovky, případně určitého logického celku, a vždy obsahují minimálně tři separátní třídy, z nichž každá reprezentuje dílčí část funkčního celku.

Demonstrace fungování je předvedena na celku obsluhujícím obrazovku s přehledem úkolů, časovačem a možností vyhledávání v úkolech – to proto, že zde existuje vcelku komplexní logika z hlediska návaznosti, která bude níže vysvětlena blíže.

5.4.1 State

State obecně popisuje všechny stavy, do jakých se aplikace může v rámci daného funkčního celku dostat. Díky konečnému počtu možných stavů je vždy jasné, v jakém stavu se daný funkční celek nachází, to vede na snadnější údržbu kódu a kontrolu případných chyb.

```
abstract class TimerState {}
```

Pro snadné předávání stavových objektů je vhodné mít definovanou bázovou třídu, ze které všechny ostatní v určité hierarchii vychází. Jelikož interface se obecně v programovacím jazyce realizuje jako abstraktní třída, je výchozím bodem právě abstraktní třída, která by neměla obsahovat žádné dodatečné definice vnitřního chování – od toho tu jsou až další stavy (technicky vzato běžné třídy), které z této abstraktní třídy dědí.

```
class TimerStateNotInitialized extends TimerState {}
class TimerStateNotLoaded extends TimerState {}
class TimerStateLoading extends TimerState {}
```

Tři základní stavy aplikace v podstatě nevyžadují žádná další vnitřní data a jen popisují aktuální stav, jsou jimi stavy popisující situaci, kdy TimerState ještě není inicializován, kdy TimerState nemá načtená data, a poslední popisuje stav, když načítání dat probíhá.

Přestože jde v podstatě o tři prázdné třídy, jejich význam nelze mezi sebou zaměňovat, neboť na jejich základě se potom provádí vykreslování příslušné informace na úrovni uživatelského rozhraní – že komponenta není inicializována by mělo nastat jen na nepatrný okamžik, než proběhne první volání Bloc struktury. Nenačtená data ale už mohou vycházet např. z chyby při připojení na server, přestože komponenta samotná je inicializována bez problémů. A načítání dat je stav, o kterém je vhodné uživatele informovat, aby věděl, že aplikace na něčem aktivně pracuje.

```
class TimerStateLoaded extends TimerState {
  final List<Timesheet> lastTimesheets;
  final String searchString;
  TimerStateLoaded(this.lastTimesheets, this.searchString);
}
```

Vnitřní struktura je součástí až třídy popisující načtený stav. Tato třída obsahuje seznam posledních vykazovaných úkolů a textový řetězec ve vyhledávacím poli. Důležité je, že všechny tyto prvky jsou doplněny předpisem final, tudíž po inicializaci v konstruktoru jsou již pro danou instanci neměnné. Stav Bloc celku tak lze nahradit pouze úplně novým stavem, případně novým stavem, který si převezme jako argumenty konstruktoru vnitřní data stavu předešlého. Tím je zajištěna konzistence přechodů mezi jednotlivými stavy – kdyby šlo vnitřní data stavu měnit po dobu jeho existence, byl by narušen princip konzistentního přecházení mezi stavy aplikace, a to protože by nedošlo k propagaci změny stavu do UI vrstvy.

```
class TimerStateSearch extends TimerStateLoaded {
  final TimerStateLoaded previousState;
  TimerStateSearch(this.previousState, String searchString)
      : super(previousState.lastTimesheets, searchString);
}
```

TimerStateSearch je stav, který nastane v případě, že uživatel zahájí vyhledávání úkolů. Tento stav si drží referenci na jemu předcházející stav, a to z toho důvodu, že vyhledávání na UI vrstvě překrývá standardní obsah stránky. Jakmile tedy vyhledávání uživatel dokončí, tak se uživatelské rozhraní vrátí do stavu, v jakém bylo před zahájením vyhledávání.

Důležité je také to, že tento stav již nevychází z abstraktní třídy, ale z načteného stavu – začíná se tak formovat určitá struktura stavů, které v návaznosti na sebe mohou nastat. Aplikace musí být nejdříve v načteném stavu, než uživatel může zahájit vyhledávání.

```
class TimerStateSearchLoading extends TimerStateSearch {
 TimerStateSearchLoading(TimerStateLoaded previousState, String searchString)
    : super(previousState, searchString);
}
class TimerStateSearchLoaded extends TimerStateSearch {
 final List<Timesheet> searchResults;
 TimerStateSearchLoaded(
    TimerStateLoaded previousState, String searchString, this.searchResults)
    : super(previousState, searchString);
}
```

Stejný princip je uplatněn i u stavů spojených s dalším zpracováním vyhledávání – stav načítání vyhledávání jen jednoduše dědí a předává argumenty nadřazenému konstruktoru obecného stavu vyhledávání, protože podstatnou informací je jen samotné načítání vyhledávaných dat. Naopak u načtených výsledků vyhledávání už je doplněn i List nalezených výsledků.

```
class TimerStateTracking extends TimerStateLoaded {
 final Timesheet currentTimesheet;
 TimerStateTracking(List<Timesheet> lastTimesheets, String searchString,
    this.currentTimesheet)
    : super(lastTimesheets, searchString);}
```

Poslední stavová třída definuje situaci, kdy uživatel aktivně vykazuje práci na daný úkol. V takovém případě se dědí ze standardního načteného stavu, ke kterému je doplněn objekt aktuálně vykazovaného úkolu.

5.4.2 Event

Aby bylo možné mezi jednotlivými stavy každého Bloc celku přecházet, je potřeba kromě stavů definovat i eventy, tedy události, které jsou na základě různých akcí vyvolávány. V reakci na tyto události se potom vykonává funkční logika a aktualizují se stavy.

```
abstract class TimerEvent {}
```

Technická koncepce událostí je přitom analogická ke koncepci stavů. Výchozím bodem je také abstraktní třída, ze které všechny události vycházejí.

```
class TimerEventInitialization extends TimerEvent {}
```

Jestliže u stavů existovala třída popisující situaci, kdy funkční blok ještě není inicializován, tak v událostech je zase naopak definována taková událost, která inicializaci spustí. Pro spuštění inicializace není potřeba dalších dat, tudíž jde jen o jednoduchou dědičnost z abstraktní třídy.

```
class TimerEventStartTracking extends TimerEvent {
  final Timesheet timesheet;
  final bool isAlreadyTracking;
  TimerEventStartTracking(this.timesheet, this.isAlreadyTracking);
}
```

Událost pro zahájení vykazování času na daný úkolu definuje úkol, na který je vykazování prováděno, a také bool příznak, který určuje, zda v době vyslání události již probíhá vykazování úkolu jiného – na základě tohoto příznaku je použita rozhodovací logika chování, viz níže v kapitole o Bloc třídě. Události mají velmi krátkou životnost objektu, protože v podstatě jen upozorní na událost, jednou projdou rozhodovací strukturou v Bloc třídě a pak zanikají.

```
class TimerEventStopTracking extends TimerEvent {
  final Timesheet timesheet;
  TimerEventStopTracking(this.timesheet);
}
```

Událost o ukončení vykazování času dostává jako argument jen právě ukončený úkol. V případě událostí se také využívá argumentů, které jsou předepsány jako final, neboť změna obsahu události po dobu její existence v paměti ani nedává koncepční smysl – data jsou z události obvykle převzata, zpracována a v případě potřeby předána dále do konstruktoru stavové třídy.

```
class TimerEventSearchingChanged extends TimerEvent {
  final String searchString;
  TimerEventSearchingChanged(this.searchString);
}
class TimerEventUpdateTaskNote extends TimerEvent {
  final String note;
  TimerEventUpdateTaskNote(this.note);
}
```

Třídy popisující události spojené se změnou vyhledávaného řetězce a poznámky, kterou lze připsat k jakémukoliv vykazovanému úkolu, jsou v zásadě úplně stejné, liší se však zacílením na jiné chování celku.

```
class TimerEventUpdateTracking extends TimerEvent {
  final TimerStateTracking state;
  TimerEventUpdateTracking(this.state);
}
```

Událost TimerEventUpdateTracking je zvláštní v tom, že ve své podstatě předává referenci jen na běžící stav vykazování úkolu. Jelikož pravidelné synchronizace času se serverem by představovaly nadbytečný datový přenos po síti, využívá se lokálního časovače, který v pravidelných intervalech aktualizuje čas v lokální aplikaci.

Jinými slovy – v průběhu vykazování úkolu se v pravidelných intervalech do streamu událostí posílá TimerEventUpdateTracking, aby se na UI vrstvě aktualizoval čas, po jakou dobu již běží vykazování daného úkolu.

5.4.3 Bloc

Poslední částí, která pro realizaci jednoho celého funkčního Bloc celku chybí, je implementace Bloc třídy, která má na starost udržování stavu, poslouchání a reagování na události příchozí do streamu a z toho vycházející změny stavu.

```
class TimerBloc extends Bloc<TimerEvent, TimerState>
```

Třída dědí z Bloc, které formou generických parametrů předává definice svých vnitřních stavů a událostí – tím jsou abstraktní třídy popsané v předešlých dvou podkapitolách. Zde už začíná být zřejmé, proč se musí dodržovat alespoň základní hierarchická struktura dědičnosti stavových a událostních tříd.

V obecné rovině lze totiž v Bloc třídě přijímat zpracovávat události a stavy, které z těchto dvou základních vychází. A díky polymorfismu potom není problém kdykoliv dle potřeby v rozhodovací struktuře zjistit, o jakou přesnou událost či stav se jedná, a dle toho volat další bloky kódu.

```
  final _apiService = getIt.get<ApiService>();
  final _localNotificationsService = getIt.get<LocalNotificationsService>();
```

Třída TimerBloc si při vzniku své instance načte zaregistrované servisní třídy, které potřebuje. V tomto případě to je ApiService, která zprostředkovává komunikaci se serverem, a LocalNotificationsService, díky níž může proběhnout nastavení lokální notifikace v případě, že probíhá vykazování úkolu.

```
@override
TimerState get initialState => TimerStateNotInitialized();
```

Třídě je potřeba nastavit počáteční stav, tím je dočasně stav bez inicializace.

```
@override
Stream<TimerState> mapEventToState(TimerEvent event) async*
```

Pravděpodobně nejdůležitější funkcí celé implementace prostřednictvím flutter_bloc knihovny, je právě mapEventToState, která pracuje s asynchronním streamem událostí dědících z TimerEvent, na základě kterých vrací nové stavy dědících z TimerState. Tyto typy z principu musí být stejné, jako generické parametry předané bázové třídě Bloc.

Speciální zápis async* popisuje, že se jedná o asynchronní stream, ze kterého lze vracet stavy, aniž by ale skončila exekuce celé struktury kódu – to je podstatný rozdíl oproti tomu, kdyby se použila standardní asynchronní funkce a data se vracela přes běžné klíčové slovo return. [20]

Místo toho prostřednictvím klíčového slova yield můžeme vrátit stav, provést nějaký blok kódu, a vrátit stav další – vše v rámci jednoho většího bloku kódu (v tomto případě if blok ve funkci). Struktura těla funkce mapEventToState tudíž obvykle zahrnuje několik if bloků, které odlišují dílčí stavy a události, které nastávají.

Díky asynchronní povaze funkce lze navíc bez problému volat komplexní operace, jako je např. stahování dat ze serveru, počkat na jejich výsledek a na základě tohoto výsledku vrátit nový stav – to vše ve vlákně jiném než vykreslujícím, tudíž bez viditelných dopadů na výkonnost UI.

```
if (event is TimerEventInitialization) {
    try {
      _localNotificationsService.cancelTrackingNotification();
      yield TimerStateLoading();
      final result = await _apiService.getLastFinishedTimesheets();
      final openTimesheet = await _apiService.getOpenTimesheet();
      if(result != null){
        yield TimerStateLoaded(result, "");
      }
      else{
        dispatch(TimerEventInitialization());
        return;
      }
      if (openTimesheet != null) {
        dispatch(TimerEventStartTracking(openTimesheet, true));
      }
```

```
    } on Exception {
      yield TimerStateNotLoaded();
    }
}
```

Prvním hlavním if blokem je reakce na událost vyvolávající inicializaci celé komponenty. Protože při dotazech na server mohou nastat výjimky (např. přerušení spojení), je celé chování drženo v try catch bloku, který v případě chyby vrátí stav popisující nenačtená data.

V případě korektního průběhu ovšem dojde k odhlášení aktivních notifikací, vrátí se stav popisující načítání dat a zavolá se samotné načítání dat, v tomto případě nejdříve seznam posledních vykazovaných úkolů, a následně kontrola, zda uživatel zrovna nemá aktivní úkol. Pokud se nepodaří získat seznam vykazovaných úkolů, aplikace se o to pokusí znovu.

Nakonec pokud je při inicializaci zjištěno, že na serveru probíhá vykazování úkolu, je do streamu zavolána událost TimerEventStartTracking. Proto se na začátku při inicializaci všechny lokální notifikace zruší, neboť jsou nahrazeny novými v reakci na tuto událost.

```
else if (event is TimerEventSearchingChanged) {
    if (event.searchString.isEmpty) {
      if (currentState is TimerStateSearch) {
        yield _getParentState(currentState);
      }
      return;
    }
    yield TimerStateSearchLoading(
        (currentState as TimerStateLoaded), event.searchString);
    final result = await _apiService.getSearchResults(event.searchString);
    if (result != null && result.timesheets != null) {
      yield TimerStateSearchLoaded((currentState as TimerStateLoaded),
          event.searchString, result.timesheets);
    }
}
```

Dalším if blokem je kontrola změny vyhledávání, kde je vidět schopnost programovacího jazyka Dart provádět převody datových typů na základě vnoření do nového scope – v else if bloku je ověření, že objekt event je TimerEventSearchingChanged, pokud tato podmínka

projde, není již třeba dělat další přetypování a lze přímo přistupovat k jeho obsahu hned v dalším (vnořeném) if bloku, kde probíhá kontrola, zda je vyhledávací řetězec prázdný.

A pokud prázdný je, vrátí se počáteční stav před zahájením vyhledávání. Protože se stavy do sebe mohou různě hluboko vnořovat závisle dle chování uživatele, je využíváno rekurzivní funkce pro opětovné získání skutečné původního stavu:

```
TimerState _getParentState(TimerState timerState) {
  if (timerState is TimerStateSearch) {
   if (timerState.previousState != null) {
    return _getParentState(timerState.previousState);
   }
  }
  return timerState;
 }
```

Pokud však vyhledávací řetězec není prázdný, je vrácen stav načítání vyhledávaných výsledků, poslán požadavek na server a po obdržení odpovědi ze serveru zpracován – pokud jsou výsledky v pořádku, je vrácen stav s načtenými výsledky. Jelikož je předem známo, že hledání v úkolech musí předcházet stav, kdy jsou data na obrazovce již načtena, je možné využívat i běžného přetypování pomocí klíčového slova *as* bez další kontroly.

```
else if (event is TimerEventStartTracking) {
   _localNotificationsService.setupTrackingNotification(event.timesheet);
   Timesheet result;
   if (event.isAlreadyTracking) {
    result = event.timesheet;
   } else {
    final trackingRequestResult = await _apiService.postTrackingTrack(
      event.timesheet.projectPart, true);
    if (trackingRequestResult != null) {
     result = await _apiService.getOpenTimesheet();
    }
   }
   if (result != null) {
    yield TimerStateTracking(
      (currentState as TimerStateLoaded).lastTimesheets, "", result);
    if (_timer != null && _timer.isActive) {
     _timer.cancel();
    }
    _timer = Timer.periodic(
      Duration(minutes: 1), (Timer timer) => _timerHandlerCallback());
   } else {
    yield currentState;
   }
}
```

If blok pro ošetření události spouštějící vykazování úkolu má dva možné zdroje volání, tím prvním je server, který vrátí data o tom, že již probíhá vykazování úkolu. V takovém případě je přiřazen příchozí úkol ze serveru. V opačném případě je zdrojem místní seznam úkolů. V reakci na to je poslán požadavek ze serveru, který v odpovědi vrátí zahájení vykazování. Dalším požadavkem je získán předpis nově spuštěného úkolu ze serveru, který je předán druhé části celého if bloku.

V té je v případě prázdných dat vrácen aktuální stav, jinak proběhne vrácení stavu, který informuje UI vrstvu o zahájení vykazování úkolu. Dále také dojde k inicializaci časovače, který v pravidelných minutových intervalech aktualizuje časovač na obrazovce. Pokud tento časovač je již aktivní, tak je resetován, jinak v pravidelných intervalech volá svoji callback funkci:

```
void _timerHandlerCallback() {
  TimerState state = currentState;
  if (state is TimerStateTracking) {
    int currentTime = TimeIntConverter.convert(
      state.currentTimesheet.projectPartCurrentHours);
    int newTime = currentTime + 1;
    state.currentTimesheet.projectPartCurrentHours =
      TimeIntConverter.convertBack(newTime);
    dispatch(TimerEventUpdateTracking(state));
  }
 }
```

V této funkci probíhá logika, která k aktuálnímu času úkolu každou minutu přičte minutu další, přičemž je potřeba používat i speciální konvertor časového formátu, neboť server pracuje s textovými údaji ve vlastním textovém formátu. Údaj je tudíž převeden na minutové časy, je k němu přičtena jedna minuta a následně je zkonvertován zpět do serverově kompatibilního formátu.

```
else if (event is TimerEventStopTracking) {
    _localNotificationsService.cancelTrackingNotification();
    final result = await _apiService.postTrackingTrack(
      event.timesheet.projectPart, false);
    if (result != null) {
      _timer.cancel();
      final lastTimesheets = await _apiService.getLastFinishedTimesheets();
      yield TimerStateLoaded(lastTimesheets, "");
    } else {
      yield currentState;
    }
}
```

If blok pro kontrolu události zastavující vykazování daného úkolu nejdříve zruší nastavené notifikace, následně pošle na server požadavek se zastavením vykazování – pokud server odpoví korektně, dojde k aktualizaci posledních vykázaných úkolů a přechodu do načteného stavu. Naopak pokud nepřijde správná odpověď ze serveru, funkční blok vrátí aktuální stav.

```
else if (event is TimerEventUpdateTaskNote) {
    await _apiService.postSaveNote(event.note);
    yield currentState;
}
```

Reakce na událost aktualizování poznámky u vykazovaného úkolu je nejjednodušší, pouze na server pošle pozměněnou poznámku a vrátí aktuální stav.

```
if (event is TimerEventUpdateTracking && currentState is TimerStateTracking){
  yield TimerStateTracking((currentState as TimerStateTracking).lastTimesheets,
  (currentState as TimerStateTracking).searchString,
  event.state.currentTimesheet);
}
```

Zvlášť vyčleněný if blok se používá pro lokální aktualizaci času, to proto, že při lokální aktualizaci času není vhodné používat událost pro start vykazování, která by na serveru spustila další úkol.

Proto pokud dojde k lokální aktualizaci vykázaného času, proběhne lokální změna stavu, beze změny dat na serveru – v podstatě jde o orientační synchronizaci dat na serveru a dat na lokálním zařízení, s maximální odchylkou jedna minuta.

Navíc se může v některých situacích stát, že než dojde na vyvolání callback události, tak uživatel už ukončí vykazování času na daný úkol. Tyto scénáře jsou ošetřeny dodatečnou podmínkou v if bloku, která kontroluje, zda aktuální stav skutečně stále odpovídá aktivnímu vykazování úkolu.

5.4.4 Shrnutí

Implementace ostatních šesti Bloc celků je v technickém principu velmi podobná a liší se zejména v tom, jak se dílčí implementace chovají. Obecně je tedy dáno, že Bloc třída implementuje rozhodovací logiku na základě událostí, ze které jsou na UI vrstvu propagovány nové stavy. Tyto změny stavu ovšem může UI vrstva zpracovat, zatím co v Bloc komponentě již probíhá asynchronní zpracování dat a připravuje se tak z toho vycházející další změna stavu.

Jak je vidět z jednoho uvedeného příkladu, hlavní nevýhodou BLoC architektury je relativně velké množství napsaného kódu např. ve srovnání s návrhovým vzorem MVVM, kde se pracuje s dílčími proměnnými přímo na úrovni jedné třídy, nikoliv v rámci rozdělení na další tři. Naopak velkou výhodou je snadno čitelný kód, kde rozhodovací struktura jde v přesně daném pořadí.

Jak je z předešlé podkapitoly vidět, všechny kontroly událostí v podstatě probíhají tak, že se zkontroluje odpovídající typ přímo prostřednictvím is operátoru v programovacím jazyce Dart. V tomto kontextu je potřeba si dávat pozor na pořadí kontrol jednotlivých typů. Je totiž nezbytné provádět kontrolu v pořadí od typů nejhlouběji ve struktuře po typy, které jsou ve struktuře výše.

Pokud by např. událost C dědila od události B a ta dědila od abstraktní události A, je třeba nejdříve provést kontrolu na událost C, pak dát else if blok na událost B. Jinak by se mohlo stát, že kód požadovaný při události C, se zavolá při události B, neboť is operátor vrací příznak true i pro situaci, kdy se testovaný typ vyskytuje hlouběji v hierarchii dědičnosti.

5.5 Třídy uživatelského rozhraní

V návaznosti na předešlou podkapitolu bude realizace uživatelského rozhraní demonstrována na obrazovce s přehledem posledních úkolů, vyhledáváním úkolů a případně informací o aktuálně vykazovaném úkolu.

Základním prvkem každé hlavní i podřízené stránky aplikace je StatefulWidget, a to protože všechny hlavní stránky využívají BLoC návrhového vzoru, který z principu potřebuje pracovat s proměnlivým vnitřním stavem. Podřízené stránky jsou potom poplatné vnitřnímu stavu nadřízené hlavní stránky, tudíž jsou typu StatefulWidget i přestože samy vlastní Bloc komponentu nemají.

StatelessWidget se v rámci aplikace využívá jen u položek reprezentujících zvolenou stránku v bočním menu a pro vykreslení koláčového grafu, který má po celou dobu existence své instance jen jednu podobu.

```
class TimerPage extends StatefulWidget {
  TimerPage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _TimerPageState createState() => _TimerPageState();
}
```

Stránka s časovačem v sobě definuje textovou podobou titulku, který se zobrazuje v horním pruhu aplikace (toolbar), a dostává jej při vzniku do konstruktoru. Po inicializaci instance třídy se volá předefinovaná funkce createState, která vrací novou instanci privátní třídy _TimerPageState. Princip je tedy stejný, jako u výchozího bodu aplikace, kde se sestavovala kořenová obrazovka.

```
class _TimerPageState extends State<TimerPage> {
  final _timerBloc = TimerBloc();
  final _searchTextController = TextEditingController();
  final _noteTextController = TextEditingController();
  Timer _searchDebouncer;
  Timer _noteDebouncer;
  bool _debounceInit = false;
```

Třída popisující vnitřní stav stránky v sobě kromě samotné inicializace příslušné instance Bloc třídy definuje několik prvků, které jsou pro celkové fungování uživatelského rozhraní důležité. TextEditingController řídí chování textových polí, od reakcí na změnu textů, reset obsahu a další funkce spojené s textem.

TextEditingController je na této stránce využíván v kombinaci s Timer objektem, který plní roli debounceru – v kontextu softwarového vývoje jde tedy o realizaci objektu, který zdrží zpracování změny dat s ohledem na to, zda se data v krátkém okamžiku opět nezmění. Není totiž v zájmu vývojáře zahltit server příliš mnoho požadavky, proto aplikace chvíli čeká, než dojde na zavolání API.

Poslední bool proměnná je jen pomocným příznakem, který předchází nežádoucímu chování debounceru. To vznikalo, neboť stránka po inicializaci chybně prováděla nechtěná volání nad textovými poli.

```
@override
  void initState() {
    _timerBloc.dispatch(TimerEventInitialization());
    _searchTextController.addListener(_onSearchChanged);
    _noteTextController.addListener(_onNoteChanged);

    super.initState();
  }
```

V předefinované funkci initState je vytvořena událost, která vynucuje inicializaci příslušné Bloc komponenty, a také nastavení posluchačů změn v textových polích pro zadávání vyhledávaného řetězce a poznámky k vykazovanému úkolu. Kdykoliv nastane změna textu v těchto prvcích, zavolá se callback na přiřazené funkce.

```
  @override
```

```
Widget build(BuildContext context) {
  return BlocBuilder(
    bloc: _timerBloc,
    builder: (BuildContext context, TimerState state) {
      if (state is TimerStateLoaded) {
        return ListView(
          children: <Widget>[
            _buildTimer(context, state),
            _buildSearchTextField(context, state),
            _buildTaskListView(context, state)
          ],
        );
      } else {
        return LoadingWidget();
      }
    },
  );
}
```

Výchozím bodem pro sestavování UI je předefinování funkce build, která v implementaci flutter_bloc knihovny vrací BlocBuilder, uvnitř kterého je anonymně doplněna funkce přijímající změnu stavu a kontext. Zde se následně rozhoduje, zda aplikace je načtená, či nikoliv. Pokud ještě data nejsou načtena, zobrazí se LoadingWidget, což je vlastní widget, který zobrazuje statickou načítací obrazovku s jednoduchou animací.

V případě načtení dat se ale vrátí ListView, které přijímá jako potomky hned tři další funkce. Postupná tvorba UI je rozdělena na dílčí funkce záměrně, protože vytvoření celé obrazovky v rámci jedné rozsáhlé funkce vede na značnou nepřehlednost a problematické členění kódu u komplexních obrazovek.

Navíc díky dalšímu členění do funkcí je možné dosáhnout komplexnějších možností skládání UI – např. pokud některá data nejsou dostupná, v samostatné funkci je jednodušší takovou situaci zkontrolovat a vrátit případně prázdný prvek. Podstatné je, že tyto funkce přebírají jak BuildContext, tak State z hlavní build funkce, tudíž mají přístup ke stejným datům, jako hlavní build funkce.

```
Widget _buildTaskListView(BuildContext context, TimerState state) {
  Widget content;
  if (state is TimerStateSearch) {
    content = _buildSearchResult(context, state);
  } else {
    content = _buildLastTasks(context, state);
  }
  return content;
```

```
}
```

A právě na funkci _buildTaskListView je ihned vidět, proč je výhodné sestavení uživatelského rozhraní členit na dílčí funkce – pokud aplikace vyhledává, vrátí na příslušnou pozici vyhledané úkoly. Pokud se nachází v jakémkoliv jiném načteném stavu, tak proběhne vrácení seznamu posledních vykazovaných úkolů. Takovou logiku by nebylo možné implementovat přímo jako součást pole prvků, jak je definováno v build funkci.

```
Widget _buildSearchTextField(BuildContext context, TimerState state) {
  return Container(
    margin: EdgeInsets.fromLTRB(16.0, 0.0, 16.0, 8.0),
    child: TextFormField(
      cursorColor: AppTheme.primaryColor,
      autofocus: false,
      decoration: InputDecoration(
        contentPadding: EdgeInsets.fromLTRB(0, 16.0, 0, 0.0),
        hintText: FlutterI18n.translate(context, "search") ,
        prefixIcon: Icon(Icons.search),
        suffixIcon: IconButton(
          icon: Icon(Icons.clear),
          onPressed: () => _searchTextController.clear())),
      controller: _searchTextController,
      style: TextStyle(color: AppTheme.primaryDarkColor)),
  );
}
```

Vyhledávací textové pole je vloženo do Container widgetu z toho důvodu, že je tak umožněno snadné nastavení odsazení od stran prostřednictvím parametru margin. Tělem tohoto kontejneru už je samotné textové pole, které kromě parametrů spojených se vzhledem přijímá i parametr controller, do kterého je mu předán příslušný TextEditingController.

Textové pole má pod parametrem suffixIcon nastaveno tlačítko, na které když se klikne, je zavolána anonymní funkce, která vyčistí textové pole.

Za pozornost stojí také nastavení parametru hintText, což je text zobrazovaný v případě, že v poli není zadána žádná hodnota – tedy v podstatě obecný popisek. Volání funkce translate říká, že pro daný BuildContext je potřeba získat zpět textový řetězec uložený v jazykových souborech pod klíčem search. V případě české lokalizace je tak vrácena hodnota z příslušného souboru cs.json, v případě slovenské lokalizace sk.json, a v ostatních případech je použita lokalizace anglická, která se nachází v souboru en.json.

```
Widget _buildTimer(BuildContext context, TimerState state) {
  TimerState timerState =
    state is TimerStateSearch ? state.previousState : state;
```

Funkce _buildTimer začíná kontrolou, zda se stav nachází ve vyhledávání – pokud ano, tak dále pracuje se stavem předcházejícím vyhledávání, jinak použije běžný stav. Toto ošetření se používá kvůli tomu, že musí být zachováno vykreslování aktuálně vykazovaného úkolu v situaci, kdy probíhá vyhledávání. Stavu vyhledávání předchází stav načtených dat, který je buď ve stavu aktivního vykazování, anebo nikoliv – proto se dále pracuje se stavem předcházejícím hledání.

Kdyby se toto ošetření nevyužívalo, v okamžiku vyhledávání by se skryl veškerý obsah poskládaný funkcí _buildTimer, protože by následující kontrola stavu neprocházela:

```
if (timerState is TimerStateTracking)
```

Zejména UI třídy mají ve frameworku Flutter poměrně dlouhý vertikální zápis, a tudíž je to v této práci rozděleno na několik dílčích blok, nicméně veškerý následující kód se provádí v případě, že výše uvedená podmínka platí.

```
_noteTextController.text = timerState.currentTimesheet.note;
    var inTime = TimeIntConverter.convert(
            timerState.currentTimesheet.projectPartCurrentHours) <=
        TimeIntConverter.convert(
            timerState.currentTimesheet.projectPartTargetHours);
```

Na začátku je potřeba inicializovat hodnotu controlleru pro poznámku, může se totiž stát, že je při inicializaci u úkolu již nastavena poznámka na straně dat získaných ze serveru. Do proměnné inTime se zjišťuje příznak, zda je na úkolu vykázáno více méně nebo právě tolik hodin, na kolik byl vypsán. K tomu je potřeba výše již zmíněné převodníky formátu z textu na číselný formát. Příznak inTime se dále používá pro nastavení podoby indikátoru progresu.

```
return Padding(
      padding: EdgeInsets.symmetric(horizontal: 16.0, vertical: 8.0),
      child: Column(children: <Widget>[…
```

Výstup funkce _buildTimer je obalen do widgetu Padding, přes který je možno realizovat vnější odsazení, v parametru child je mu potom přiřazen widget Column, který do sebe vertikálně vkládá všechny widgety, které jsou mu dodány v poli.

Jelikož se typy widgetů v následující části kódu po technické stránce opakují a jen se mění jejich obsah, budou v následujících příkladech uvedeny jen reprezentativní demonstrace pro každý druh widgetu.

```
SizedBox(height: 8,),
```

Widget SizedBox lze využít více způsoby, v kontextu této aplikace je ale využíván pro odsazení mezi prvky v rámci Column, protože Column přímo nenabízí odsazení mezi potomky.

```
Text(
  timerState.currentTimesheet.about,
  style: TextStyle(
    fontWeight: FontWeight.bold,
    fontSize: 20.0,
    color: AppTheme.primaryColor),
  maxLines: 1,
  overflow: TextOverflow.ellipsis,
),
```

Uvedený příklad widgetu Text zobrazuje popisek aktuálně vykazovaného úkolu, který je získaný ze stavu. Widgetu je dále nastavena velikost písma, tloušťka písma, barva a omezení velikosti na jeden řádek textu, s tím že přetékající text se ořízne elipsou a doplní se o tři tečky naznačující pokračování textu dále.

```
LinearProgressIndicator(
      value: inTime
        ? TimeIntConverter.convert(
            timerState.currentTimesheet.projectPartCurrentHours) /
          TimeIntConverter.convert(
            timerState.currentTimesheet.projectPartTargetHours):null,
      valueColor: inTime
        ? AlwaysStoppedAnimation<Color>(Colors.greenAccent)
        : AlwaysStoppedAnimation<Color>(AppTheme.primaryDarkColor),),
```

LinearProgressIndicator ukazuje lineární progres v rámci řady, kde existuje maximální hodnota a aktuální hodnota. Využívá se např. při načítání, kdy je známý počet kroků do konce načítání, anebo právě zde, kde znázorňuje, kolik hodin je na daném úkolu odpracováno z těch předem přiřazených.

Využívá se i výše získaného příznaku inTime, na základě kterého je rozhodnuto, zda bude do hodnoty dosazen poměr mezi vykázaným časem a cílovým časem, anebo zda bude null. Pokud je hodnota null, vykresluje se tzv. neurčitý indikátor, ve kterém je znázorněna aktivita bez konkrétního progresu. Dle inTime příznaku se také rozhoduje, zda indikátor budemít zelenou barvu (úkol je zpracováván v přiděleném čase), či v tmavé barvě (úkol je zpracováván nad rámec přiděleného času).

```
Padding(   padding: EdgeInsets.fromLTRB(0.0, 8.0, 0.0, 0.0),
      child: RaisedButton(
        textColor: AppTheme.secondaryTextColor,
        color: AppTheme.primaryColor,
```

```
        child: Text(FlutterI18n.translate(context, "stop") ),
        onPressed: () => _timerBloc.dispatch(
          TimerEventStopTracking(timerState.currentTimesheet))),)
```

Odsazené tlačítko RaisedButton na stisk vyvolává anonymní funkci, která na stream Bloc objektu pošle událost informující o záměru ukončit vykazování daného úkolu.

```
else {
    return SizedBox(height: 0, width: 0);
}
```

Obecně v případě, že pro danou funkci nenastal stav požadovaný pro vrácení příslušných dat, je na konci všech těchto sestavujících funkcí vracen SizedBox s nulovým rozměrem. Z pohledu uživatele se nic nezobrazí, ovšem vracet standardní prázdný null v tomto kontextu není možné, protože framework Flutter by padal na výjimky. Tomu lze předejít právě vracením prázdného widgetu s nulovou velikostí.

Funkce _buildSearchResult a _buildLastTasks jsou v technické rovině prakticky identické a jen operují nad rozdílnými daty, tudíž v následující části bude popsána jen první jmenovaná funkce, nicméně obsahově popis bude odpovídat i pro druhou zmíněnou funkci.

```
Widget _buildSearchResult(BuildContext context, TimerState state) {
  Widget content;
  if (state is TimerStateSearchLoaded) {
   if (state.searchResults != null && state.searchResults.length > 0) {
     content = ListView.builder(
      shrinkWrap: true,
      physics: NeverScrollableScrollPhysics(),
      itemCount: state.searchResults.length,
      itemBuilder: (BuildContext context, int index) {
        return _getTaskCell(state.searchResults[index]);
      },
     );
```

Pro sestavení seznamu prvku se obecně používají ListView, které je v tomto případě vráceno, jestliže existují nalezené výsledky vyhledávání. Využívá se funkce builder, která přijímá parametry pro nastavení jak podoby ListView, tak především parametr itemBuilder, do kterého je dosazena anonymní funkce. Tato anonymní funkce se volá pro sestavení každého dílčího prvku – a jak je vidět, v této implementaci jde opět o samostatnou další funkci _getTaskCell.

Důležitým parametrem ListView je ale také příznak shrinkWrap a physics. Tím prvním je shrinkWrap, který ListView sděluje, že má zabírat jen tak velký prostor, jaký samo potřebuje – nemá tudíž potřebu roztahovat se přes veškerý dostupný prostor. To je velmi důležité ve

vztahu k tomu, že celá tato stránka je vložena do jednoho nadřazeného ListView, které už je posuvné samo o sobě.

Posuvný obsah vnořený v jiném posuvném obsahu je obecně bez ohledu na framework či platformu považován za špatnou praktiku z hlediska uživatelského zážitku, ovšem v případě frameworku Flutter vzniká i další sada problémů.

Jelikož neexistují limity pro velikost obsahu, který lze posouvat, vnořené ListView by v případě povoleného roztahování vyhazovalo výjimku, neboť by se snažilo roztahovat přes prakticky nekonečně velkou plochu. Ovšem nastavením shrinkWrap v kombinací nastavením parametru physics na NeverScrollableScrollPhysics je vypnuto jak toto roztahování, tak posouvání v rámci vnitřního widgetu.

Veškerou zodpovědnost za posouvání obsahu tak přebírá nadřazené ListView, které je definováno ve funkci build. Tento přístup se používá napříč celou aplikací, neboť téměř všechny stránky pracují s posuvnými seznamy dat.

```
return Column(mainAxisSize: MainAxisSize.min, children: <Widget>[
        Padding(
          padding: EdgeInsets.fromLTRB(16.0, 0.0, 0.0, 0.0),
          child: Row(
            children: <Widget>[
              Text(
                FlutterI18n.translate(context, "found_tasks") ,
                textAlign: TextAlign.start,
                style: TextStyle(
                    fontWeight: FontWeight.bold,
                    color: AppTheme.primaryColor,
                    fontSize: 20.0),)],),),
        Flexible(child: content,)]);
```

Výsledné ListView je vloženo do sloupce, který po své hlavní ose zabírá co nejmenší množství místa, tedy jinými slovy se také nepokouší roztahovat ve vertikální ose. A v sobě potom definuje textový popisek nalezených úkolů, a úkoly samotné, které jsou ještě vloženy do Widgetu Flexible. Ten umožňuje ListView vyplnit prostor v rámci tohoto vráceného sloupce tak, aby jeho buňky nebyly prostorově omezeny.

```
else { return Text(FlutterI18n.translate(context, "nothing_found") ); }
```

A jelikož se jedná o funkci vracející výsledky vyhledávání, pokud žádné nejsou, je vrácen pouze text informující o absenci výsledků.

```
Widget _getTaskCell(Timesheet timesheet) {
   return Padding(
```

```
   padding: EdgeInsets.symmetric(horizontal: 8.0),
   child: Card(
     child: InkWell(
    child: …
    onTap: () =>
      _timerBloc.dispatch(TimerEventStartTracking(timesheet, false)),)),);}
```

Funkce _getTaskCell vrací buňky jak pro vyhledávané úkoly, tak pro poslední vykazované úkoly. Její obsah je složen z poměrně dlouhé struktury textových widgetů, které již byly demonstrovány výše, proto jsou z příkladu vynechány. Důležitý prvek je ale widget InkWell, který umožňuje z jakéhokoliv prvku uživatelského rozhraní udělat funkční tlačítko.

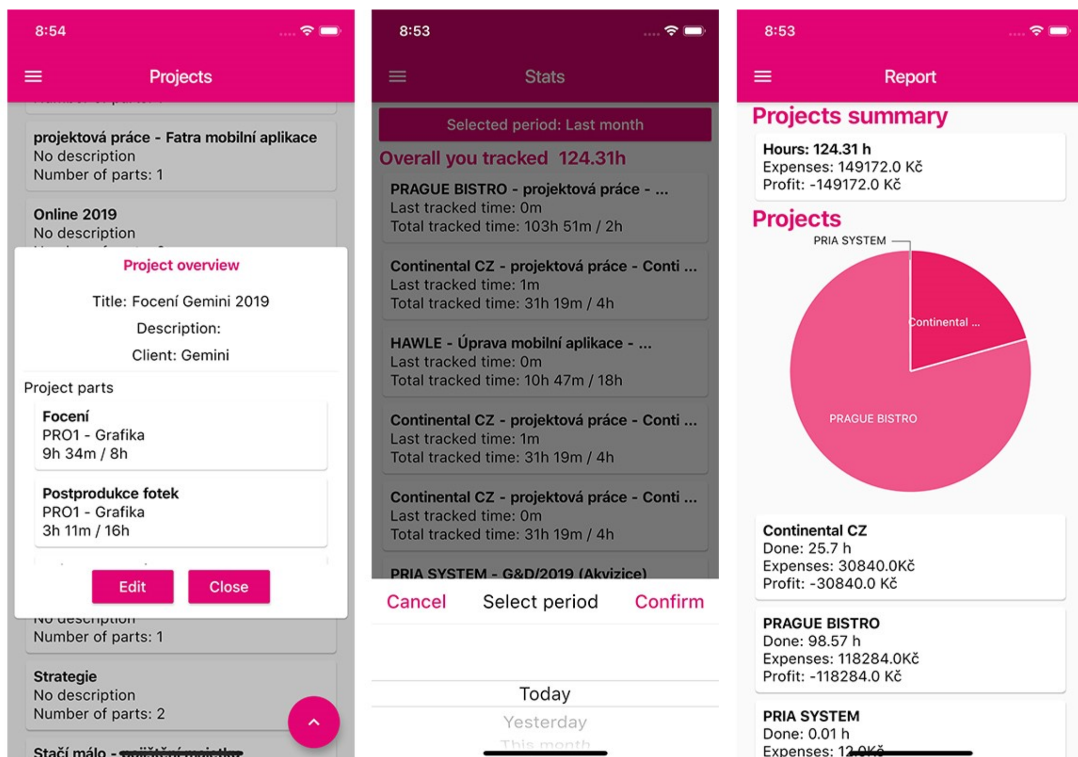
Při kliknutí na kartu s údaji o úkolu je tudíž pomocí anonymní funkce zavoláno poslání události. Tato událost se pošle na stream událostí a informuje Bloc objekt, že uživatel chce zahájit vykazování daného úkolu.

```
void _onSearchChanged() {
  if (!_debounceInit) {
    _debounceInit = true;
    return;
  }
  if (_searchTextController.text ==
      (_timerBloc.currentState as TimerStateLoaded)?.searchString) {
    return;
  }
  if (_searchDebouncer?.isActive ?? false) _searchDebouncer.cancel();
  _searchDebouncer = Timer(const Duration(milliseconds: 500), () {
    _timerBloc
        .dispatch(TimerEventSearchingChanged(_searchTextController.text));
  });
 }
```

Jelikož vyhledávací textové pole je na UI vrstvě vykreslováno neustále, při prvním vykreslení je v něm nastaven kurzor na nulovou pozici. To ovšem díky nastavenému posluchači změn textu spustí zavolání celé této callback funkce i v situaci, kdy to není žádoucí. Tudíž první volání je přeskočeno. V druhém if bloku probíhá kontrola, zda se text skutečně změnil od toho vyhledávaného, pokud nikoliv, volání končí.

V posledním if bloku proběhne už i kontrola debounceru. Pokud není aktivní, resetuje se a spustí. Pokud v následujících 500 milisekundách nedojde na změnu textu, debouncer pošle do streamu v Bloc objektu událost informující o tom, že se vyhledávaný text změnil – v reakci na to už Bloc objekt pošle požadavek na server a zpracuje výsledky.

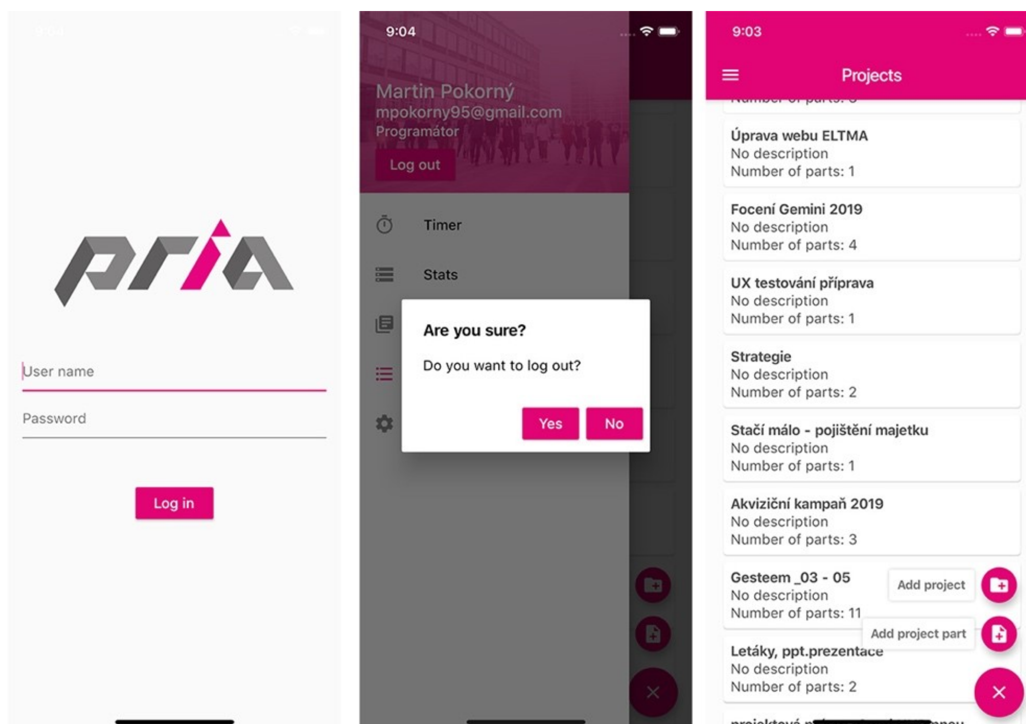
Na stejném principu funguje i callback funkce _onNoteChanged pro změnu poznámky u aktuálně vykazovaného úkolu, pouze s tím rozdílem, že nepotřebuje první ošetřující if blok, protože není na UI vykreslována neustále, nedochází ani k nekorektnímu prvotnímu volání – a samozřejmě ověření v druhém if bloku probíhá vůči aktuálně nastavené poznámce u daného úkolu.



Obrázek 14 Screenshoty z přehledu projektu, statistik a reportu uživatele

Tímto způsobem jsou v aplikaci vytvářeny všechny stránky i podstránky, s tím rozdílem, že podstránky jsou modální a nemají vlastní Bloc objekty. Naopak dostávají konkrétní zpracovávaná data od stránky hlavní, která Bloc objekt má.

Tato data zpracují a při zavření prostřednictvím callback funkce předají zpět nadřízené hlavní stránce, která už je prostřednictvím události předá svému Bloc objektu tak, jak bylo v uvedených demonstracích kódu předvedeno.



Obrázek 15 Screenshoty z přihlašovací obrazovky, vyskakovacího dialogu a menu překryvného tlačítka

ZÁVĚR

Výsledná mobilní aplikace napsaná prostřednictvím frameworku Flutter představuje komplexní demonstraci všech hlavních funkcí, které tento framework nabízí. Kromě toho praktická část práce nabízí vhled do návrhového vzoru BLoC, který sice vede na poměrně rozsáhlý kód, který je rozdělen mezi několik tříd i v rámci jednoho funkčního celku, na druhé straně ale výsledný kód je velmi přehledný a v zásadě jeho exekuce probíhá jednosměrně, čímž se snadno zjišťují návaznosti a případné nedostatky v kódu.

Flutter v době vzniku této práce není ani rok ve stabilní verzi. Práce na výstupní aplikaci začala ještě v době, kdy Flutter byl ve fázi beta vývoje. S tím přirozeně přichází nedostatek v podobě menšího množství dokumentace a podpůrných materiálů, než jakého se dostává v případě nativního vývoje, či vývoje prostřednictvím multiplatformních technologií, které na trhu působí delší dobu.

Naopak výhodou je, že si Flutter s sebou nenese žádnou technologickou zátěž a vzniká jako nová technologie podporující moderní techniky vývoje, mezi které patří právě i reaktivní přístup k vývoji.

I přes mladost celé technologie byly problémy při vývoji minimální a za celou dobu vývoje vyvstal jen jeden problém, který byl nedostatkem přímo na straně frameworku. Byl jím problém se zobrazováním kontextového menu, který způsoboval pád aplikace. Tento problém je sice stále otevřený, ale existuje způsob, jak jej obejít, tudíž ani to vývoj zásadně neovlivnilo.

Google vyvinul Flutter tak, aby při jeho implementaci šlo využít libovolných návrhových vzorů, obecně ale přímo v Google doporučují návrhový vzor BLoC, který se i při vývoji demonstrační mobilní aplikace osvědčil zejména pro přehlednost danou jednosměrnou exekucí kódu a jasné oddělení rolí dílčích tříd.

SEZNAM POUŽITÉ LITERATURY

[1] CASSERLY, Martin. IPhone vs Android market share. *Macworld* [online]. Londýn: IDG UK, 2019 [cit. 2019-04-27]. Dostupné z: https://www.macworld.co.uk/feature/iphone/iphone-vs-android-market-share-3691861/

[2] J. GARBADE, Michael. Native vs. cross-platform app development: pros and cons. *Codeburst* [online]. San Francisco: EDU Ecosystem, 2018 [cit. 2019-04-27]. Dostupné z: https://codeburst.io/native-vs-cross-platform-app-development-pros-and-cons-49f397bb38ac

[3] BUTUSOV, Mykhailo. Native vs Hybrid apps. What to choose in 2019. *TechMagic* [online]. Lvov: TechMagic, 2019 [cit. 2019-04-27]. Dostupné z: https://blog.techmagic.co/native-vs-hybrid-apps/

[4] DE KERF, Dimitri. FLUTTER: HYBRID APPS FOR MOBILE & BEYOND. *JWORKS TECH BLOG* [online]. Mechelen: Ordina JWorks, 2019 [cit. 2019-04-27]. Dostupné z: https://ordina-jworks.github.io/development/2019/01/10/Flutter.html

[5] PEDLEY, Adam. How Flutter Works. *BuildFlutter* [online]. 2018 [cit. 2019-04-27]. Dostupné z: https://buildflutter.com/how-flutter-works/

[6] *Developer Survey Results 2019* [online]. New York City: Stack Exchange, 2019 [cit. 2019-04-27]. Dostupné z: https://insights.stackoverflow.com/survey/2019

[7] SNEATH, Tim. Flutter 1.0: Google's Portable UI Toolkit. *Google Developers* [online]. Mountain View: Google, 2018 [cit. 2019-04-27]. Dostupné z: https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui-toolkit.html

[8] HRÁČEK, Filip a Matt SULIVAN. Build reactive mobile apps with Flutter (Google I/O '18). *YouTube* [online]. Mountain View: Google, 2018 [cit. 2019-04-27]. Dostupné z: https://www.youtube.com/watch?v=RS36gBEp8OI

[9] FRACHET, Marvin. Understanding the React Native bridge concept. *Hacker Noon* [online]. Hacker Noon, 2018 [cit. 2019-04-27]. Dostupné z: https://hackernoon.com/understanding-react-native-bridge-concept-e9526066ddb8

[10] MALIK, Nyma. React Native Pros and Cons. *CitrusBits* [online]. San Francisco: CitrusBits, 2018 [cit. 2019-04-27]. Dostupné z: https://citrusbits.com/react-native-pros-and-cons/

[11] *THE LEGENDS OF .NET* [online]. 2018 [cit. 2019-04-27]. Dostupné z: http://corefx.strikingly.com

[12] PROSISE, Jeff. Supercharging Xamarin Forms with Custom Renderers, Part 1. *Wintellect* [online]. Atlanta: Wintellect, 2015 [cit. 2019-04-27]. Dostupné z: https://www.wintellect.com/supercharging-xamarin-forms-with-custom-renderers-part-1/

[13] POKORNÝ, Martin. *Tvorba nativních multiplatformních mobilních aplikací s využitím knihovny Xamarin.Forms*. Zlín: Univerzita Tomáše Bati ve Zlíně, 2017, 80 s. (125 241 znaků). Dostupné také z: http://hdl.handle.net/10563/41289. Univerzita Tomáše Bati ve Zlíně. Fakulta

aplikované informatiky, Ústav automatizace a řídicí techniky. Vedoucí práce Král, Erik.

[14] MALIK, Sahil. Xamarin versus Cordova. *CodeMag* [online]. Houston: EPS Software, 2018 [cit. 2019-04-27]. Dostupné z: https://www.codemag.com/article/1703031/Xamarin-versus-Cordova

[15] RAMOS, Javier. Ionic 4: All you need to know. *Medium* [online]. Irsko: Medium, 2019 [cit. 2019-04-27]. Dostupné z: https://medium.com/@javier.ramos1/ionic-4-all-you-need-to-know-d2b9627aaf03

[16] MASA. My Experience building an app for Android and iOS with Qt. *Medium* [online]. San Francisco: Medium, 2016 [cit. 2019-04-27]. Dostupné z: https://medium.com/@shigmas/my-experience-building-an-app-for-android-and-ios-with-qt-5d4a63ab7d6

[17] *Qt for Mobile App Development* [online]. Helsinki: The Qt Company, 2019 [cit. 2019-04-27]. Dostupné z: https://www.qt.io/mobile-app-development/

[18] *Flutter - Beautiful native apps in record time* [online]. Mountain View: Google, 2019 [cit. 2019-04-29]. Dostupné z: https://flutter.dev

[19] Flutter: Pros and Cons for Seamless Cross Platform Development. *Hacke Noon* [online]. Belmont: SteelKiwi, 2018 [cit. 2019-04-29]. Dostupné z: https://hackernoon.com/flutter-pros-and-cons-for-seamless-cross-platform-development-c81bde5a4083

[20] *Dart programming language* [online]. Mountain View: Google, 2019 [cit. 2019-04-29]. Dostupné z: https://www.dartlang.org

[21] LELER, Wm. Why Flutter Uses Dart. *Hacker Noon* [online]. Mountain View: Google, 2018 [cit. 2019-04-29]. Dostupné z: https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf

[22] SULLIVAN, Matt. Flutter: Don't Fear the Garbage Collector. *Medium* [online]. Mountain View: Medium, 2019 [cit. 2019-04-29]. Dostupné z: https://medium.com/flutter-io/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30

[23] Technical Overview - Flutter. *Flutter* [online]. Mountain View: Google, 2019 [cit. 2019-05-02]. Dostupné z: https://flutter.dev/docs/resources/technical-overview

[24] BIRCH, Joe. Stateful or Stateless widgets?. *FlutterDoc* [online]. Mountain View: Google, 2018 [cit. 2019-05-02]. Dostupné z: https://flutterdoc.com/stateful-or-stateless-widgets-42a132e529ed

[25] HRÁČEK, Filip. Inherited Widgets Explained - Flutter Widgets 101 Ep. 3. *YouTube* [online]. 2019: Google, MountainView [cit. 2019-05-02]. Dostupné z: https://www.youtube.com/watch?v=Zbm3hjPjQMk

[26] Asynchronous Programming: Futures. *Dart* [online]. Mountain View: Google, 2019 [cit. 2019-05-04]. Dostupné z: https://www.dartlang.org/tutorials/language/futures

[27] SHARMA, Atul. *How to develop a platform channel in Flutter between Dart and Native Code* [online]. San Francisco: Medium, 2019 [cit. 2019-05-04].

Dostupné z: https://medium.com/@atul.sharma_94062/creating-a-bridge-in-flutter-between-dart-and-native-code-in-java-or-objectivec-5f80fd0cd713

[28] *JSON and serialization* [online]. Mountain View: Google, 2019 [cit. 2019-05-04]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/json

[29] BERTRAND, Chris. Coding Concepts - Reflection. *DEV Community* [online]. New York City: DEV Community, 2019 [cit. 2019-05-04]. Dostupné z: https://dev.to/designpuddle/coding-concepts---reflection-4d2c

[30] BROGDON, Andrew. *Some Options for Deserializing JSON with Flutter* [online]. Mountain View: Google, 2018 [cit. 2019-05-04]. Dostupné z: https://medium.com/flutter-io/some-options-for-deserializing-json-with-flutter-7481325a4450

[31] BABENKO, Vladimir. *Flutter App Lifecycle* [online]. Las Vegas: Pharos Production, 2019 [cit. 2019-05-04]. Dostupné z: https://medium.com/pharos-production/flutter-app-lifecycle-4b0ab4a4211a

[32] BOELENS, Didier. BLoC - ScopedModel - Redux - Comparison. *Flutter - Didier Boelens* [online]. Brusel: Didier Boelens, 2019 [cit. 2019-05-04]. Dostupné z: https://www.didierboelens.com/2019/04/bloc---scopedmodel---redux---comparison/

[33] TERPIL, Jenya. Redux. From twitter hype to production. *Slides* [online]. Kyjev: Slides, 2016 [cit. 2019-05-04]. Dostupné z: http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production#/24

[34] SAVJOLOVS, Vadims. Flutter app architecture 101: Vanilla, Scoped Model, BLoC. *Medium* [online]. San Francisco: Medium, 2019 [cit. 2019-05-04]. Dostupné z: https://medium.com/flutter-community/flutter-app-architecture-101-vanilla-scoped-model-bloc-7eff7b2baf7e

[35] ANGELOV, Felix. Bloc, a predictable state management library for Dart. *Bloc* [online]. Chicago: Github, 2019 [cit. 2019-05-04]. Dostupné z: https://felangel.github.io/bloc/

[36] ABRAHAMSSON, Joel. Inversion of Control – An Introduction with Examples in .NET. *Joel Abrahamsson* [online]. Stockholm: Joel Abrahamsson, 2010 [cit. 2019-05-04]. Dostupné z: http://joelabrahamsson.com/inversion-of-control-an-introduction-with-examples-in-net/

[37] BURKHART, Thomas. One to find them all: How to use Service Locators with Flutter. *GuruMeditation* [online]. Kolín nad Rýnem: Guru Meditation, 2018 [cit. 2019-05-04]. Dostupné z: https://www.burkharts.net/apps/blog/one-to-find-them-all-how-to-use-service-locators-with-flutter/

[38] WINDMILL, Eric. *Flutter in Action*. 2018. Portland: Manning, 2018. ISBN 9781617296147. Dostupné také z: https://www.manning.com/books/flutter-in-action

[39] FRANCESCHI, Hervé. *Android app development*. Burlington, MA: Jones & Bartlett Learning, [2018]. ISBN 978-1284092127.

[40] GAUCHAT, J.D. *IOS Apps for Masterminds 4th Edition: How to take advantage of Swift 4.2, iOS 12, and Xcode 10 to create insanely great apps for*

iPhones and iPads. 4. 4. vyd. Toronto: Amazong Digital Services, 2018. ISBN 978-1724466440.

[41] MEIER, Reto. *Professional Android 4e*. 4. vyd. Indianapolis, IN: John Wiley, 2018. ISBN 978-1118949528.

[42] PANHALE, Mahesh. *Beginning hybrid mobile application development*. New York, NY: Apress, [2016]. ISBN 978-148-4213-148.

[43] *Mobile App Development for iOS and Android, Edition 2.0*. 2. vyd. Burlington, VT: Prospect Press, 2017. ISBN 978-1943153282.

[44] INTERNATIONALIZATION. *W3C* [online]. Cambridge, Massachusetts: W3C, 2016, 2016 [cit. 2019-05-12]. Dostupné z: https://www.w3.org/standards/webdesign/i18n

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

GPS	Global Positioning System
XML	Extensible Markup Language
HTML	Hypertext Markup Language
CSS	Cascade Style Sheets
JS	JavaScript
UI	User Interface
API	Application Programming Interface
Obj-C	Objective-C
WPF	Windows Presentation Foundation
MVVM	Model-View-ViewModel
QML	Qt Modeling Language
AOT	Ahead-of-Time
JIT	Just-in-Time
X86	x86 kompatibilní procesory
ARM	Advanced RISC (Reduced Instruction Set Computer) Machines
JSON	JavaScript Object Notation
BLoC	Business Logic Component
MVC	Model-View-Controller
IoC	Inversion of Control
REST	Representational State Transfer
SDK	Software Development Kit
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
XAML	Extensible Application Markup Language

SEZNAM OBRÁZKŮ

Obrázek 1 Rozdělení trhu z pohledu mobilních systémů z kraje roku 2019 [1] 9
Obrázek 2 Flow z JavaScript vrstvy na vrstvu nativního iOS v Reactu [9] 14
Obrázek 3 Xamarin.Forms a znázornění komunikace s nativní vrstvou [12] 15
Obrázek 4 Základní struktura frameworku Flutter [5]...18
Obrázek 5 Popis algoritmu za Young Space Scavenger generací garbage collectoru [22] 21
Obrázek 6 - struktura widgetů [23]..23
Obrázek 7 Struktura komunikace mezi nativní a sdílenou vrstvou [18].....................26
Obrázek 8 Struktura fungování návrhového vzoru Redux [33].................................29
Obrázek 9 Struktura fungování návrhového vzoru ScopedModel [32]......................31
Obrázek 10 Struktura fungování návrhového vzoru BLoC [32] 32
Obrázek 11 Screenshoty z bočního menu, vyhledávání v úkolech a vykazování úkolu 34
Obrázek 12 Zjednodušená struktura návrhového vzoru BLoC v implementaci flutter_bloc knihovny [35]...35
Obrázek 13 Navržená struktura aplikace realizované v praktické části práce............37
Obrázek 14 Screenshoty z přehledu projektu, statistik a reportu uživatele 73
Obrázek 15 Screenshoty z přihlašovací obrazovky, vyskakovacího dialogu a menu překryvného tlačítka 74

SEZNAM TABULEK

Tabulka 1 Obecné shrnutí metod vývoje [3] [4] [5] [43] 10
Tabulka 2 Shrnutí srovnání frameworků [7]...[15]..17

SEZNAM PŘÍLOH

P I CD disk s diplomovou prací a soubory zdrojového kódu