Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Multiplatformní vývoj s využitím frameworku Flutter

Bakalářská práce

Autor: Štěpán Záliš
Studijní obor: Aplikovaná informatika (ai3-p)

Vedoucí práce: doc. Mgr. Tomáš Kozel, PhD.

Hradec Králové duben 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 29.4.2020 Štěpán Zališ

Poděkování:

Děkuji vedoucímu bakalářské práce doc. Mgr. Tomášovi Kozlovi PhD., za metodické vedení práce, odborné rady a vstřícnost při zpracování. Děkuji také své rodině a přítelkyni za trpělivost a podporu, bez níž by tato práce nemohla vzniknout.

Anotace

Cílem této bakalářské práce je popis jednoho z mnoha řešení pro vývoj multiplatformní mobilní aplikace. V úvodu práce je čtenář seznámen s problematikou vývoje aplikací jak z hlediska různorodosti platforem, tak i z globálního tržního podílu. Teoretická část práce se věnuje představení této technologie a stručnému popisu fungování. Dále je pak seznámen s možnými architekturami mobilní aplikace. Na základě poznatků z teoretické části bude výstupem mobilní aplikace, využívající tuto technologii pro podporu návštěvnosti událostí, konajících se na Univerzitě Hradec Králové. Jedná se zejména o podporu každoročních akcí, jako jsou „Den otevřených dveří", „Pedagogické dny", „International Day" a „Veletrh pracovních příležitostí". Aplikace by měla sloužit jak pro zájemce z veřejnosti, tak i pro stávající studenty zmiňované univerzity.

Klíčová slova: framework, Flutter, Dart, mobilní aplikace, plugin, widget, Firebase, Firestore

Annotation

Multiplatform development using Flutter framework

The aim of this thesis is to describe one of many solutions for the development of a cross-platform mobile application. In the introduction, the reader is acquainted with the problems of application development both in terms of platform diversity and global market share. The theoretical part is devoted to the introduction of this technology and a brief description how it works. Then he is acquainted with possible architectures of mobile apps. Based on the knowledge from the theoretical part, the output will be a mobile application using this technology to support attendance at events taking place at the University of Hradec Kralove. In particular, it supports annual events such as "Open Day", "Pedagogical Days", "International Day" and "Job Fair". The application should serve both the public and current students of the university.

Keywords: framework, Flutter, Dart, mobile application, plugin, widget, Firebase, Firestore

Obsah

1 Úvod ... 1

2 Cíl práce a metodika zpracování ... 2

3 Úvod do vývoje mobilních aplikací ... 4

3.1 Multiplatformní vývoj ... 6

3.2 Frameworky ... 6

3.2.1 Hybridní ... 7

3.2.2 Multiplatformní ... 7

3.2.3 Sdílení kódu ... 8

4 Popis použitých technologií ... 9

4.1 Dart ... 9

4.1.1 Historie ... 9

4.1.2 Syntaxe ... 10

4.1.3 Asynchronní programování ... 13

4.1.4 Knihovny a správa balíčků ... 16

4.2 Flutter ... 18

4.2.1 Historie ... 19

4.2.2 Deklarativní způsob ... 19

4.2.3 Widget ... 21

4.3 Návrhové modely řízení stavu aplikace ... 24

4.3.1 Lokální řízení stavu ... 24

4.3.2 Globální řízení ... 25

4.4 Webové služby, JSON serializace a deserializace ... 29

4.5 Firebase služby ... 29

4.5.1 Firestore ... 29

4.5.2 Authentication ... 30

4.5.3 Administrace ... 30

5 Návrh mobilní aplikace ... 31

5.1 Požadavky ... 31

5.1.1 Funkční požadavky ... 31

5.1.2 Non-funkční požadavky ... 33

5.2 Návrh obrazovek ... 33

5.3 Nástroje pro ulehčení vývoje ... 34

5.3.1 Volba návrhového vzoru ... 34

5.3.2 Inversion of Control ... 36

5.4 Návrh architektury ... 37

5.5 Struktura aplikace ... 39

5.5.1 Zobrazovací vrstva ... 39

5.5.2 Vrstva práce s daty ... 39

5.5.3 Společné jádro aplikace ... 40

5.6 Popis implementace ... 40

5.6.1 Spuštění aplikace a registrace tříd ... 40

5.6.2 Přihlašování pomocí Firebase Authentication ... 43

5.6.3 Komunikace s Firebase Firestore ... 44

5.6.4 Komunikace se REST API, deserializace JSON dat ... 45

5.6.5 Práce s repositáři ... 46

5.6.6 Bloc třídy – funkční logika ... 47

5.7 Uživatelské rozhraní ... 51

5.8 Výsledný vzhled aplikace ... 53

5.8.1 Splash screen ... 53

5.8.2 Průvodce aplikací ... 53

5.8.3 Domovská obrazovka ... 54

5.8.4 Konferenční událost ... 55
5.8.5 Program konferenční události ... 56
5.8.6 Detail konferenčního programu ... 56
5.8.7 Mapa ... 56
5.8.8 Další informace a nastavení ... 57
6 Testování ... 59
7 Výsledky práce ... 60
8 Závěr a doporučení ... 62
9 Seznam použité literatury ... 64
10 Seznam obrázků ... 67
11 Seznam tabulek ... 67
12 Přílohy ... 68
12.1 Zadání práce ... 68
12.2 Příloha č. 2 – drátěný model aplikace ... 69
12.3 Příloha č. 3 – vybrané obrazovky drátěného modelu ... 70
12.4 Příloha č. 4 – grafické podklady ... 71

1 Úvod

Mobilní aplikace tvoří důležitou roli v našich životech. Tato situace je dána zejména tím, že mobilní telefon či tablet vlastní téměř každý. Je cenově dostupný, jeho rozměry jsou malé a výkon některých zařízení dosáhl výkonu porovnatelným s laptopy. Další výhodou je snadná propojitelnost s dalšími zařízeními, jako jsou chytré hodinky, sluchátka apod. Tyto okolnosti z těchto relativně malých, ale výkonných zařízení, dělají společníka, bez kterého je těžké si dnes představit běžný život.

S rozvojem 3G a LTE sítí a snížením cen tarifů také narostl počet účastníků, využívajících tyto sítě nejen pro běžnou komunikaci, ale i jako pracovní nástroj. Za posledních pět se zvýšil přenos dat přes mobilní zařízení o 222 %. Předpokládá se, že tento trend bude stoupat a mobilní zařízení se stanou primárním zařízení k tomu, být online. [1]

Aby byl využit potenciál mobilních telefonů a tabletů, jsou nutnou součástí také aplikace v nich. Tyto aplikace nabízí uživateli mít svá data po ruce téměř kdykoli a kdekoli a bez těchto aplikací by se z těchto zařízení stala jen málo využitelná krabička. Nabízejí nám potenciál, který je ku prospěchu nás všech v moderní době využít.

Tyto okolnosti dali za vznik nápadu mobilní aplikace pro podporu událostí na Univerzitě Hradec Králové. Na této univerzitě se pořádá mnoho zajímavých akcí, pořádaných nejen zaměstnanci, ale i zástupci firem a odborné veřejnosti. Ačkoli se univerzita snaží propagovat tyto události na sociálních sítích a na webových stránkách, nabízí se využít i potenciál mobilní aplikace. Uživatelé, kteří budou aplikaci používat, budou operativně informováni o nových událostech, a to bez nutnosti sledování výše uvedených informačních kanálů. Důsledkem bude zlepšení obecného povědomí o těchto akcích, což by mělo vést i ke zvýšení jejich návštěvnosti.

2 Cíl práce a metodika zpracování

Cílem teoretické části práce je popsat základní charakteristiky jednoho z možných řešení pro tvorbu multiplatformní aplikace – frameworku Flutter od společnosti Google. Hlavním cílem práce je prozkoumat již zmíněný Flutter, jeho možnosti, výhody a nevýhody a také případná omezení. Nutnou dílčí součástí je prozkoumat programovací jazyk Dart, který je pro tento vývoj určený. Současně také navrhnout architekturu celé mobilní aplikace tak, aby vyhovovala standardům dnešní doby.

Na podkladě informací, získaných z teoretické části, bude výstupem praktické části mobilní aplikace pro Android a iOS, využívající tuto technologii pro podporu návštěvnosti událostí, konajících se na Univerzitě Hradec Králové (UHK). Jedná se zejména o podporu akcí, jako jsou „Den otevřených dveří", „Pedagogické dny", „International Day" a „Veletrh pracovních příležitostí". Aplikace by měla sloužit jak pro zájemce z řad veřejnosti, tak i pro stávající studenty. Lze ji chápat jako moderní náhradu plakátů a bannerů, umísťovaných jak v budovách patřících UHK, tak i jiných prostorách pro to určených. Měla by taktéž sloužit jako podpůrný prostředek propagace událostí na sociálních sítích.

Prvním krokem v teoretické části bude prozkoumat a popsat framework Flutter, důvody a okolnosti jeho vzniku, včetně popisu fungování, jako nutného aspektu pro jeho pochopení. Rovněž budou popsány základní konstrukce programovacího jazyka Dart a jeho výhody (případně omezení) pro užití při tvorbě dané aplikace, a to včetně zdůvodnění jeho volby. Teoretické znalosti budou získávány z internetových zdrojů a odborné literatury.

Praktickou částí bude vytvoření mobilní aplikace pro platformu Android a iOS. Prvním krokem pak určení funkčních a non funkčních požadavků. Nutné bude také stanovit prioritu jednotlivých funkcionalit v rámci vývoje. Následně proběhne vytvoření wireframů[1] pro pochopení interakce mezi jednotlivými obrazovkami.

[1] Wireframe: skica nebo také schéma rozložení prvků na obrazovce

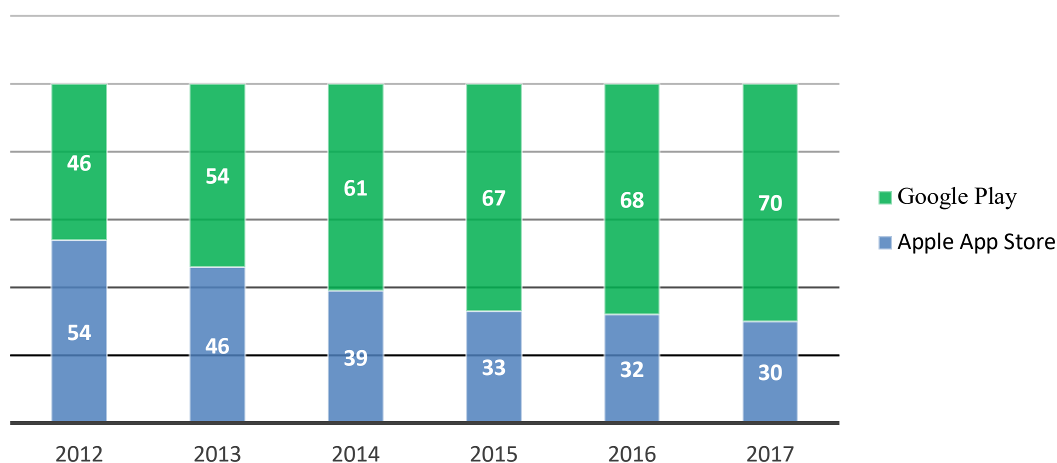
Poté následuje vytvoření grafických podkladů pro tvorbu aplikace. Dalším krokem pak bude popis jednotlivých částí a vytvářeného vzhledu s ukázkami. Zde bude nezbytný popis využití externích systémů jako zdroje dat.

Praktickou částí bude vytvoření mobilní aplikace pro platformu Android a iOS s následným postupem jednotlivých kroků:

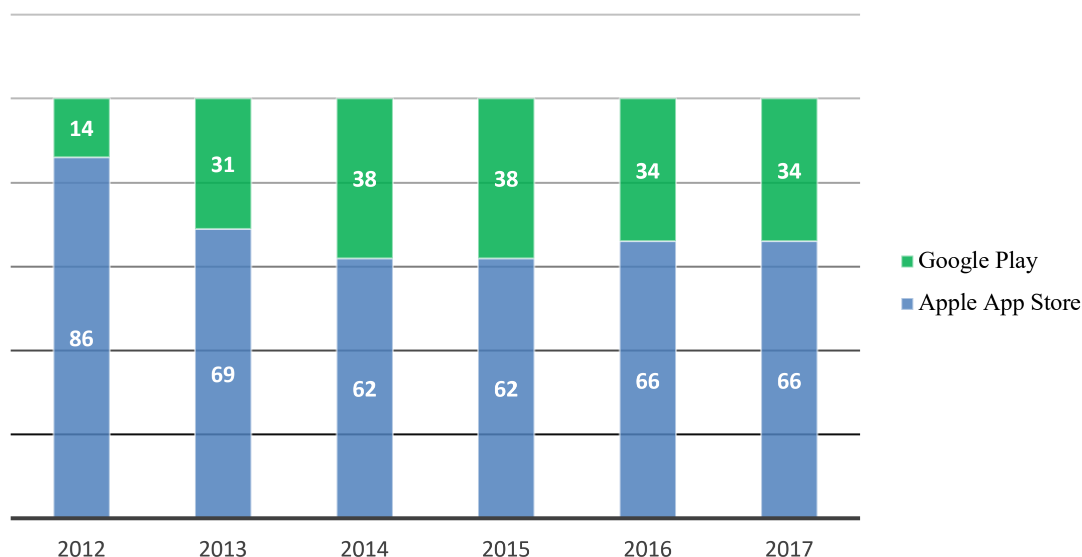
- určení funkčních a non-funkčních požadavků,
- stanovení priorit jednotlivých funkcionalit v rámci vývoje,
- vytvoření wireframů pro pochopení interakce mezi jednotlivými obrazovkami,
- vytvoření grafických podkladů pro tvorbu aplikace,
- popis jednotlivých částí a vytvářeného vzhledu s ukázkami.

3 Úvod do vývoje mobilních aplikací

Trh s mobilními zařízeními v současné době ovládají zejména dvě hlavní platformy: operační systém (OS) Android od společnosti Google a operační systém iOS společnosti Apple. OS Android zaujímá okolo 74 % celkového trhu a iOS má podíl na trhu téměř 25 %. [2] Je patrné, že OS Android si vytvořil v průběhu let značný náskok před iOS. Důvodem je zejména rozmanitost a cenová dostupnost zařízení velkého množství výrobců, díky níž se vytvořila obrovská základna uživatelů. Počet zařízení, využívající tento OS, čítá dnes dohromady přes dvě miliardy. Tento fakt by mohl na první pohled vést k předpokladu, že tato platforma bude sloužit jako primární cíl pro vývoj. Přesto ale existuje důvod, proč se tomu prozatím nestalo a možná nějakou dobu ještě nestane. Tímto důvodem je zpeněžení aplikace. U uživatelů zařízení, používajících Apple produkty, je podle statistik mnohem pravděpodobnější, že utratí peníze za nákup aplikace nebo v průběhu jejího používání. Jak je vidět na následujících grafech (Graf 1. a Graf 2.), přestože počet stažení aplikací iOS z oficiálního distribučního kanálu App Store v důsledku zvyšujícího se počtu Android zařízení klesá, podíl výdajů za aplikaci roste. [3]



Graf 1: Podíl stažených aplikací dle platforem v % [3]



Graf 2: Podíl výdajů za nákup nebo útratu v aplikaci v % [3]

Existence dvou různých platforem klade samozřejmě zvýšené nároky na vývojáře a vývojářské firmy. Ti (a v končeném důsledku i samotní uživatelé) by ocenili řešení, které by urychlilo jak vývoj samotných aplikací, tak i jejich následnou udržitelnost. To vše při zachování konzistentnosti aplikací na obou platformách a rovněž stejného (případně ještě lepšího) výkonu aplikace.

3.1 Multiplatformní vývoj

Vývoj aplikací pro Android a iOS je složitý, zejména z důvodu odlišností obou platforem (např. jiná obecná pravidla, tzv. guidelines pro chování aplikace a její vzhled). V obou případech existují doporučení popisující chování a vzhled aplikace, které by měli vývojáři respektovat.

Zásadní rozdílnost obou platforem je ale také v jiných programovacích jazycích – Java (nebo modernější jazyk Kotlin) pro Android a Objective-C (nebo taktéž modernější Swift) pro iOS. Odlišné programovací jazyky s sebou přináší nároky na znalosti a zkušenosti vývojáře, které pak v důsledku zvyšují cenu vyvíjené aplikace. Současně zpravidla vyžadují i větší personální kapacity.

3.2 Frameworky

Pro rozdělení multiplatformních frameworků do jednotlivých kategorií je potřeba definovat, co vlastně framework je, a co to znamená ve světě mobilních aplikací.

Framework je předem naprogramovaná struktura, která obsahuje podpůrné programy a aplikační rámce, které mají vývojářovi zjednodušit vývoj. [4] Vývojář se tedy nemusí soustředit na to, jak se navigovat na jiné obrazovky, ale na to, jak naprogramovat cílenou funkcionalitu.

Jelikož je výsledná aplikace často závislá na použitém frameworku, nelze po jeho odstranění z projektu aplikaci jednoduše znovu spustit. Tato závislost může být velká nevýhoda všech frameworků, a to nejen v mobilním vývoji.

Frameworky lze rozdělit do několika skupin. První skupinou jsou frameworky tzv. hybridní, které umožňují napsání jak logiky aplikace, tak i celého uživatelského rozhraní, a to bez nutnosti znát všechny vlastnosti a chování dané platformy. Druhou kategorií jsou multiplatformní, které mají blíž k nativním aplikacím. Zvláštní kategorii tvoří možnost sdílení kódů mezi platformami.

3.2.1 Hybridní

Zástupci této kategorie jsou Ionic a PhoneGap. Oba tyto frameworky využívají pro psaní aplikace webové technologie jako je HTML, CSS a JavaScript (JS). V některých případech lze pro usnadnění vývoje využít webové frameworky (např. Vue.js). V podstatě se jedná o klasickou webovou stránku, která je ale doplněna o speciální elementy. Celá tato stránka se následně vykreslí v nativní aplikaci v komponentě zvané WebView[2]. Při kompilaci je tato „webová stránka" nahrána do WebView, a tudíž lze výslednou aplikaci standardně nahrát na Google Play a App Store.

Vývoj aplikací s využitím hybridních frameworků je většinou rychlý a nevyžaduje hluboké znalosti vývoje mobilních aplikací. Omezením však může být přístup k API[3] platformám, jako je např. bluetooth nebo k fotoaparátu. Přístup k těmto API jsou většinou zprostředkovány přes různé pluginy. Tyto pluginy odstíní závislost na platformě a vývojář využívá jen předem definované třídy a metody konkrétního pluginu, čímž se často vývoj zjednoduší. Problém ovšem může nastat při specifické potřebě použití. Plugin totiž nemusí existovat nebo existuje v nedostatečné kvalitě. Právě kvalita a dostupnost těchto pluginů jsou důležitým faktorem pro hodnocení jednotlivých frameworků.
Negativní stránkou věci je také samotná komponenta WebView, která má vliv na rychlost aplikace – většina těchto aplikací bohužel nedosahuje výkonu srovnatelného s nativními nebo multiplatformními aplikacemi. [5] Tento typ vývoje se nehodí na větší a komplexní aplikace.

3.2.2 Multiplatformní

Mezi nejznámější multiplatformní frameworky patří React Native, Xamarin.Forms a v této práci popisovaný Flutter. Společná vlastnost těchto technologií je, na rozdíl od výše popsaných hybridních aplikací, nevyužití komponenty WebView. To přináší významný pozitivní vliv na výkon celé aplikace. Ačkoli je pro přístup

---

[2] WebView – komponenta zodpovědná za vykreslení webových stránek
[3] API – Application Programming Interface – množina tříd, metod a protokolů knihovny jež může programátor využívat

k platformní API někdy nutná znalost SDK platformy, je velká většina kódu sdílena. Avšak stejně jako u hybridních technologií, existuje široká škála pluginů, které tuto problematiku řeší.

Zdrojový kód je při kompilaci přeložen do nativního kódu platformy. Výsledkem jsou tedy spustitelné soubory, které lze také nahrát do obchodu s aplikacemi.

Oproti nativními přístupu (tedy psát kód pro každou platformu zvlášť) je multiplatformní přístup snazší a rychlejší na vývoj. Negativní stránkou může být pomalá reakce nově představené funkce platformy.

3.2.3 Sdílení kódu

Třetí možností je tzv. sdílení kódu, kdy se oddělí uživatelské rozhraní (UI) aplikace dle platformy a společná je pouze „business“ logika (chování aplikace). Příkladem tohoto přístupu je Kotlin Multiplatform, využívající jazyk Kotlin Native. Tento přístup umožňuje napsání jádra aplikace (například modelové třídy, komunikace s API), které je pro obě platformy stejné, avšak zachovat nativní přístup k UI komponentám dle platformy.

Pokud je kód závislý na platformě, a tedy i použití konkrétní API, třídu nebo metodu „přepíšeme“. Pokud se jedná např. o modelové třídy, není toto předepsání nutné. V tomto případě je možná přímá kompilace Kotlin kódu a vložení do projektu pro obě platformy.

Kotlin Multiplatform je ale stále v experimentální fázi a nedoporučuje se ho používat v produkci. [6]

4 Popis použitých technologií

V této kapitole jsou představeny použité technologie nutné pro vývoj aplikace ve fremeworku Flutter.

4.1 Dart

Dart je objektově-orientovaný, multiparadigmatický programovací jazyk, ovlivněný C-jazyky (jako je Java, C# nebo JS). Z každého z těchto jmenovaných jazyků si odvozuje některé vlastnosti a přidává k tomu některé další, které jsou naznačeny v následujících kapitolách. Dart má také volitelné typování, takže podporuje statické i dynamické, a to podle preferencí daného uživatele. Tento jazyk lze použít v prohlížeči, na serveru, v příkazové řádce nebo právě v mobilním vývoji. [7]

Tento jazyk je vyvíjen jako open-source pod licencí New BSD (Berkeley Software Distribution), tudíž lze zdrojový kód prohlížet, upravovat nebo šířit pouze s omezeními, plynoucími od autora. Podobně jako v Javě je pro běh programu nutná metoda *main().* [8]

4.1.1 Historie

Programovací jazyk Dart byl vytvořen v roce 2011 zaměstnanci Googlu Larsem Bakem a Kasperem Lundem. Původním záměrem bylo vytvořit takový jazyk, který by nahradil prohlížeči podporovaný JS, avšak s možností dynamického typování. Z důvodu využití na webových stránkách se tvůrci rozhodli vytvořit vlastní verzi prohlížeče, vycházející z Chromu, takzvaný „Dartium“, kde se Dart mohl spustit nativně. Záměr nahradit široce využívaný JS se však tvůrcům nepodařil a celý tým od této myšlenky postupně opustil. Řešením a částečnou náhradou bylo vytvoření kompilátoru, který umožňuje převést kód z tohoto jazyka do JS pomocí dart2js a následně tento kód spustit v jakémkoli prohlížeči. Až do roku 2015, kdy se poprvé začalo mluvit o frameworku Flutter, se zdálo, že Dart je téměř „mrtvý jazyk“ s nejistými vyhlídkami do budoucna. [9]

4.1.2 Syntaxe

Jak bylo napsáno v úvodu, Dart umožňuje psát objektově orientované aplikace, tudíž lze kód rozdělit na jednotlivé části, které lze následně znovu využít. Tento přístup šetří čas a také do určité míry snižuje míru chybovosti, protože chyba se často opraví pouze jednou. Atributy a metody, které jsou používány ve třídách, lze spojit do částí, které do sebe v projektu postupně zapadají a tvoří výsledný celek. [7]

4.1.2.1 Proměnné a datové typy

Proměnné lze vytvářet dvojím způsobem. Dart umožňuje statické nebo dynamické typování. Oba tyto přístupy s sebou přináší určité výhody i nevýhody.
Statické typování je obecně považováno za doporučované, protože umožňuje větší přehled programátora, snadnější úpravu kódu a hledání chyb. Jelikož byl Dart inspirován u C jazyků, datové typy jsou podobné a standardní jako v ostatních jazycích. [7]

Podobně jako v jazyce Java lze vytvořit celočíselnou proměnnou takto:

```
int cislo = 42;
```

Druhý způsob využívá dynamické typování nebo tzv. „type inference“, tedy analyzátor umí poznat datový typ pro lokální proměnné nebo i pro návratové hodnoty funkcí. Pokud analyzátor nepozná datový typ (nemá dostatek informací pro získání datového typu) využije dynamický datový typ. [7]

Předchozí příklad lze zapsat takto:

```
dynamic cislo = 42;
```

Proměnné lze však zapsat i bez specifikování datového atributu, podobně jako v JS jen s klíčovým slovem *var*[4]:

```
var cislo = 42;
```

Změna, například:

```
cislo = 1.001; // Chyba: nelze provést změnu datového typu
```

Statická analýza, která běží před samotnou kompilací, ohlásí chybu a program nejde zkompilovat. Pokud bychom chtěli za běhu změnit celočíselný datový typ na reálný (nebo naopak), můžeme v tomto případě zaměnit var a použít klíčové slovo num. Num je datový typ, který umožňuje jak celočíselnou, tak i reprezentaci v reálných číslech.

```
num cislo = 3;
cislo = 4.0;
```

Pokud nějakou proměnnou chceme specifikovat jako nemodifikovatelnou, lze před jméno proměnné zapsat klíčové slovo final, která znemožní následnou změnu hodnoty proměnné.

```
final cislo = 42;
cislo = 23 // Chyba: nelze změnit hodnotu final
```

4.1.2.2 Viditelnost

Řešení viditelnosti se od ostatních jazyků neliší, existují tedy dva hlavní typy – public a private. Rozdíl oproti zmíněným jazykům v úvodu je ale ve způsobu zápisu. V Dartu je standardní viditelnost atributů nastavena jako public a nijak se tedy nespecifikuje. Pokud ale chceme omezit přístup k nějaké proměnné nebo metodě, musíme přidat podtržítko před první písmeno ve jménu proměnné či metody. [7]

---

[4] var – variable, neboli proměnná

Příklad na proměnné:

```
String text  // otevřená viditelnost
String _text // privátní viditelnost
```

4.1.2.3 Třídy

Jelikož je Dart objektově orientovaný jazyk, je založený na třídách. Každá třída obecně dědí z třídy *Object*. Dart umožňuje pouze jednonásobnou dědičnost. Pokud již použijeme dědičnost a je potřeba využít metody nebo atributy z dalších tříd, je možné použít takzvané *mixins*. Na mixins lze nahlížet jako na rozhraní. V implementaci se využívá dědičnost, avšak tato dědičnost neznamená generalizaci, ale jde jen o přidání funkcionality. [7]

Konstruktory v tomto jazyce jsou dvojího typu. Výchozí konstruktor má název dle názvu třídy. Pokud je potřeba vytvořit další konstruktor, používá se takzvaný jmenný konstruktor. Pokud by se třída např. jmenovala *Student* proměnou *int vek*, konstruktor by byl *Student()* a jmenný konstruktor například *Student.sVekem(23).*
Jelikož ne vždy je nutné zadávat všechny parametry v konstruktoru, umožňuje Dart volitelné parametry nebo parametry s předem danou hodnou. Pokud by byl v předchozím případě standardní věk 23 let, konstruktor by byl *Student([vek = 23])*. [10]

Dart také umožňuje vytvářet různé instance pomocí klíčového slova *factory* (česky továrna)*. Factory* je implementace obecného návrhového vzoru. Tento návrhový vzor je jeden nejdůležitějších a často používaný, protože umožňuje vyšší abstrakci kódu, než klasický konstruktor. Další možností využití je při vícenásobném použití konstruktoru se stejnými parametry, ale jiným chováním. [10]

4.1.2.4 Kompilace

Velká výhoda Dartu je možnost duálního typu kompilace. Dart umožňuje "Ahead of Time" a "Just in Time" kompilaci. Ahead of Time (zkráceně AOT) je způsob kompilace zdrojového kódu, kde se kód převede do strojového kódu tak, aby mohl být zobrazen nativně na dané platformě.

Rozdíl oproti druhému způsobu JIT kompilace je v tom, že kód běží ve virtuálním stroji (VM) a kompilace je tedy velmi rychlá. V porovnání s AOT nedosahuje kód po kompilaci s JIT tak dobrou optimalizaci, a do jisté míry snižuje rychlost aplikace a zvyšuje náročnost na paměť. Výhodou je ale rychlost kompilace.

To, že Dart umožňuje oba způsoby kompilace a zároveň již dlouho podporuje pomocí *dart2js* kompilaci do JS, je pravděpodobně i hlavní důvod, proč si autoři Flutteru tento jazyk vybrali. Dalším důvodem je speciálně navržený garbage collector, který tento jazyk využívá. Je totiž optimalizovaný pro malé a krátkodobé objekty, tedy přesně zapadá do potřeb frameworku Flutter. [11]

4.1.3 Asynchronní programování

Dart podporuje asynchronní programování dvojího typu. Jedná se o *Future* a *Stream.* Každý typ je vhodný pro řešení jiného problému. Použití lze ilustrovat na následující tabulce, kde je současně pro srovnání ukázán i synchronní přístup.

Synchronní	int	Iterator<int>
Asynchronní	Future<int>	Stream<int>

Tabulka 1: Ilustrace synchronní a asynchronní přístupu

Pokud je potřeba hodnota pouze jednou a není nutné akci pravidelně opakovat, je vhodné použít *Future*. Naopak při provádění akce v pravidelných časových úsecích je na místě použít *Stream*. Konkrétní příklady si lze ukázat na dvou rozdílných implementacích.

4.1.3.1 Future

„Future reprezentují prostředky, které podávají hodnoty, které se vyskytnou v dalším chodu programu, ale v nynější době nejsou k dispozici. Když přijde daná funkce Future na řadu, v prvé řadě vrátí nedokončený objekt, který v budoucnu převezme dané hodnoty a dokončí se. Samozřejmě může nastat chyba, když se objektu nedostane hodnota, která je požadována. Tato chyba může být v rámci nedokončené input operace, chybou vstupu od uživatele, přerušení spojení apod. Pro získání dat a hodnot, které Future reprezentuje, se používá buď async nebo await. Tyto komponenty pomáhají programátorovi využívat výhody asynchronního programování tak, aby vypadalo jako synchronní.“ [8]

Použití Future je možné ilustrovat na následujícím kódu. Nejdříve se zavolá funkce *main*, ve které se následně zavolá asynchronní metoda *stahniObjednavku.* Tato funkce je asynchronní, neblokuje tedy hlavní vlákno programu. Proto se nejdříve do konzole vypíše *Stahovani objednávky* a následně až *Cerny caj*. Pokud by nebyla použita *async* funkce a „uspali“ bychom běh programu, pořadí výpisu by bylo opačné.

```
// Ilustracni funkce která vrati objednavku az za 4s
Future<void> stahniObjednavku () async {
  return Future.delayed(Duration(seconds: 4), () {
    return print("Cerny caj");
  });
}

void main () {
  stahniObjednavku();
  print("Stahovani objednávky");
}
```

4.1.3.2 Stream

Použití *Future,* popsané výše, není jediný způsob realizace asynchronního programování. Další možností je použití generické třídy *Stream.*

Stream je sekvence událostí. Lze si to představit jako tok událostí. Událost může být element námi zvoleného datového typu nebo chybová událost, která značí, že

se v proudu stala chyba. Pokud Streamem protekla všechna data, je „posluchač“ upozorněn na konec.

Existují dva typy Streamů. *Single-subscription* a *broadcast.* Tyto typy značí, kolik „posluchačů“ Stream může mít zaregistrováno v jeden čas. První ze zmíněných znamená, že tok dat může být poslouchán pouze jedním posluchačem. Pokud je zaregistrován další posluchač, program skončí výjimkou. Druhý typ značí, že může existovat víc posluchačů Streamu najednou.

Aby se Stream mohl kontrolovat, existuje třída StreamController. Jde o řídící prvek, který obsahuje jak Stream, tedy tok dat navenek, tak i tzv. Sink. Sink je třída, která slouží pouze ke vkládání dat do daného Streamu. [12]

Pomocí StreamControlleru lze s daty, která „protékají“, různě manipulovat a transformovat je. Lze například brát pouze každý sudý prvek apod. Po skončení toku dat je ovšem nutné zavřít Sink.

Celý koncept si lze představit na jednoduchém principu. Sink si lze představit jako hrdlo trubice. Pokud do hrdla nalijeme vodu, vznikne proud vody, se kterým lze dále pracovat.

Ilustrace použití funkce, vracející Stream, může být třeba čtení jednolivých řádků v souboru. Jelikož je Stream v podstatě proud dat, lze tyto jednotlivé prvky jednoduše transformovat, upravovat a validovat dle libosti. To je demonstrováno na ukázce níže.

```
void readFile() {
  File file = File("text_file.txt");
  file.openRead() // metoda vracejici Stream
    .transform(UTF8.decoder)
    .listen((String data) => print(data);
     onError: (error) => print(error);
     onDone: () => print("Hotovo"));
}
```

4.1.3.3 Shrnutí

Z pohledu vývojáře aplikací je podpora a snadná implementace asynchronních metod zásadní, protože většina uživatelských interakcí je závislá na zdrojích (baterie, CPU a GPU), které jsou limitované a není dobré s nimi plýtvat. Při synchronním volání má každá operace pevně daný časový úsek, ve kterém se může operace vykonat. Ve světě mobilních aplikací je tento časový úsek přibližně 16ms[5]. Pokud tato operace trvá déle, může se uživateli zdát, že aplikace není plynulá. Tomuto jevu se odborným slovem říká *jank*. Pokud nějaká operace trvá déle, než je výše uvedené časové okno, operační systém to umí rozpoznat a automaticky vynechá některé snímky z vykreslování. To pak uživatel může zpozorovat mírným poskočením nebo zamrznutím dané aplikace. [13]

Jako příklad asynchronní operace lze uvést zápis velkého souboru (například fotky nebo videa) do úložiště dat. Tato operace je závislá na mnoha faktorech (jako je velikost souboru, zda se jedná o lokální nebo vzdálené úložiště apod.). Pokud by se tato operace neprovedla asynchronně, nešlo by po určitý čas se zařízením pracovat a pro uživatele by aplikace byla uživatelsky nepoužitelná.

4.1.4 Knihovny a správa balíčků

Běžně se při vývoji softwaru stává, že je nějaká funkcionalita předem naprogramovaná a není nutné, aby ji programátor znovu vymýšlel a implementoval.

K tomu slouží v Dartu balíčkovací systém, nebo v originále „pub package manager“. [14]

Tyto závislosti se píší do speciálního souboru pubspec.yaml. Tento soubor má předem definovanou strukturu (viz ukázka níže), kterou je třeba dodržovat.

---

[5] Většina displejů mobilních zařízení pracuje s 60 FPS (60 snímků za vteřinu). Tento údaj je tedy vypočten podílem 1000ms a 60 FPS (1000 / 60 = 16,666)

V souboru je nutné uvést jméno aplikace, popis, a pokud jde knihovnu nebo plugin, také autora. Následně definovat závislosti do určené kategorie *dependencies*. Lze také ale zadat i další informace, jako je URL dokumentace, domovská stránka balíčku nebo také verze. [14]

Konkrétní soubor může vypadat například následovně:

```
name: UHK Events
author: Štěpán Záliš
version: 0.9.0
dependencies:
  knihovna: ^0.6.0
  dalsiKnihovna: 0.0.1
```

Pro zvýšení verze *knihovna* stačí v příkazové řádce zavolat flutter *pub upgrade.* Tímto příkazem se stáhnou a zvýší všechny definované závislosti dle nově zvolené verze.

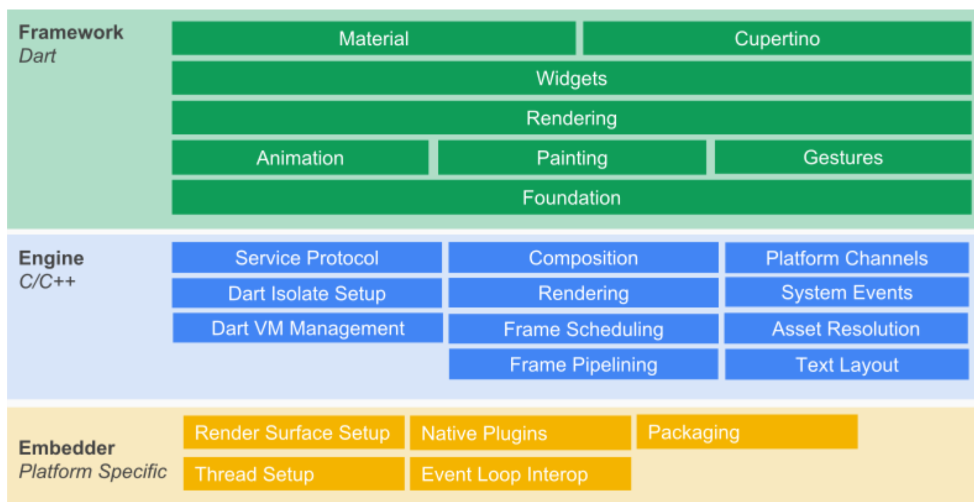
4.2 Flutter

Jak již bylo zmíněno v předchozí kapitole (3.2.2), Flutter patří mezi multiplatformní frameworky. To tedy umožňuje vývoj Android a iOS aplikace z jedné kódové základny. Je volně dostupný a open-source[6].

Byl vytvořený společností Google, která ho veřejnosti představila v roce 2017. Ačkoli bylo v této práci několikrát použito slovo framework, Flutter jako takový obsahuje dvě základná části.

První část tvoří SDK[7], což je kolekce nástrojů, které jsou nutnou součástí pro vytvoření aplikace. V tomto SDK je například kompilátor, který zkompiluje kódy do strojového kódu dle jednotlivých platforem. Nejdůležitější součástí je takzvaný *engine*, který je kompletně zodpovědný za celé vykreslování a komunikaci s platformní částí.

Druhou část tvoří právě framework, který obsahuje různé grafické elementy a komponenty, např. tlačítka, textová pole apod. Tyto takzvané widgety (popsány v kapitole 5.2.1) lze využít pro stavbu a konečnou vizualizaci aplikace.



Obrázek 1: Náhled struktury frameworku [15]

[6] open-source – volně dostupné zdrojové kódy
[7] SDK – software development kit

4.2.1 Historie

Je poměrně obtížné zjistit, jakým způsobem a proč Flutter vlastně vznikl. Jak řekl Eric Seidel, jeden z tvůrců Flutteru, v pravidelném rozhovoru na kanálu Google Developers, vznikl shodou náhod a nebyl ani plánován. Inženýři Googlu, kteří primárně pracovali na optimalizaci webového prohlížeče Chrome, kolem roku 2014 začali řešit problém s kompatibilitou a rychlosti renderování stránek webu. Jelikož projekt o velikosti tohoto webového prohlížeče má několik milionů řádků, začali z tohoto obrovského projektu odstraňovat některé části, o kterých si mysleli, že jsou příčinou problému, který se snažili vyřešit. Začali tedy odstraňovat a separovat různé části. Po nějaké době zjistili, že mají v ruce něco, co je extrémně rychlé v grafickém vykreslování. Začali přemýšlet o využití tohoto potenciálu. Jelikož měl tento tým (nebo jeho členové) zkušenosti s vývojem nativních aplikací a uvědomovali si, jaké problémy vývojáře trápí (např. pomalý build aplikace), rozhodli se vytvořit framework primárně pro mobilní aplikace. Postupem času, a také díky dobré modularitě (viz *Obrázek 1*), vzniklo velké množství realizací, kdy byl Flutter spuštěn i na ostatních platformách, jako jsou embedded[8] zařízení, desktop, nebo web. Právě tyto dvě posledně zmíněné jsou vytvořené týmem Google vývojářů a jsou oficiálně podporované. [16]

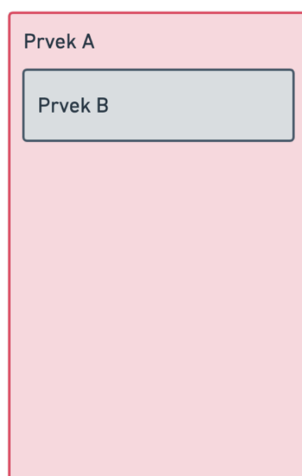
4.2.2 Deklarativní způsob

V programování existují dvě hlavní programovací paradigmata. Imperativní a deklarativní (které lze dále rozdělit na funkcionální a logické). Pro mnoho lidí je asi nejznámější právě imperativní paradigma, kde je pomocí předem definovaného algoritmu dosažen cíl. Program se tedy skládá z přesného návodu. Naopak při deklarativním přístupu je specifikován cíl (čeho se má dosáhnout) a algoritmus je ponechán programu, resp. překladači jazyka.

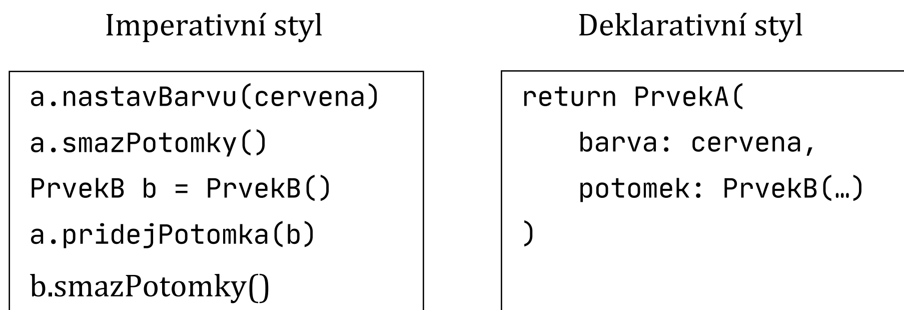
U vývoje nativních mobilních aplikací pro Android a iOS tomu není jinak. Pro ilustraci je uvedeno srovnání imperativního a deklarativního principu tvorby

[8] Embedded – jednoúčelové zařízení nebo mikropočítač

grafického rozhraní. Imperativní styl je založen na tzv. mutaci vnitřních stavů, kdy například pomocí volání metod s různými parametry měníme vnitřní stav daného objektu. Na druhou stranu, při deklarativním stylu je vnitřní reprezentace vždy stejná (a ani ji nelze měnit), ale celý objekt se při změně vytvoří znovu.



Obrázek 2: Simulace obrazovky aplikace (vlastní zpracování)



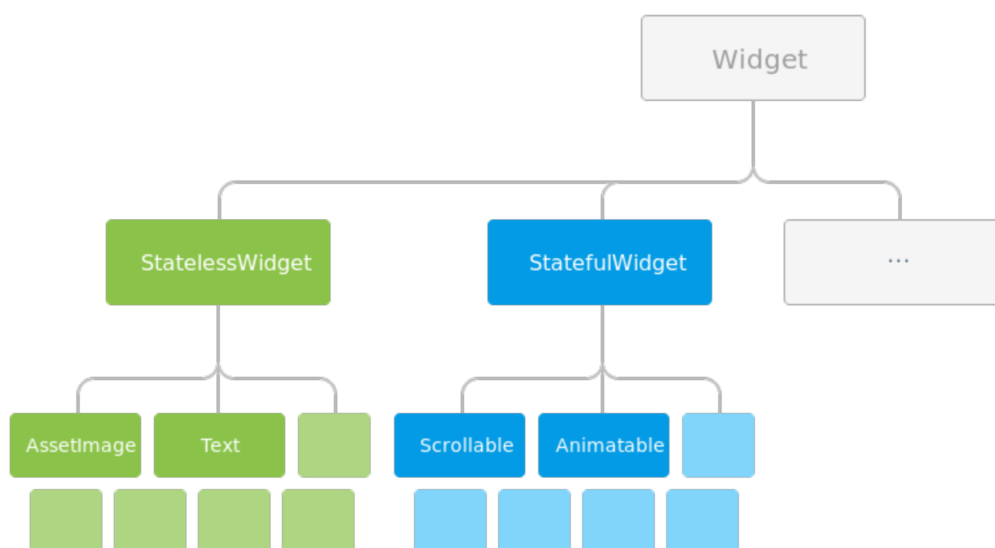
Obrázek 3: Srovnání imperativního a deklarativního stylu [17]

Jak bylo zmíněno výše, při změně barvy je u deklarativního stylu přegenerován celý strom, místo toho, aby byla zavolána metoda, která tuto barvu změní. Pokud se zamyslíme, napadne nás otázka, v čem je vlastně výhoda, protože takový přístup musí být náročnější, a to nejen na paměťové prostředky. Proč tomu tak ve Flutteru nutně není a jaké optimalizace jsou navrženy, je popsáno v následujících kapitolách.

4.2.3 Widget

Framework Flutter se skládá z takzvaných widgetů. Widget je základní stavební prvek uživatelského rozhraní. Každý widget tvoří neměnnou část tohoto rozhraní. Na rozdíl od ostatních frameworků, kde existují různá view, view controllery, rozložení stránek, nabízí Flutter sjednocení, a tedy v konečném důsledku i značné zjednodušení. Widget může být tlačítko, mezera, text nebo i celá aplikace. [16]

Největší síla frameworku spočívá v tom, že jednotlivé widgety se do sebe mohou vnořovat a tím tvořit struktury, které se dále mohou členit dále na další složité struktury. Tuto strukturu si lze představit jako strom, kde se jednotlivé uzly dále dělí na další uzly nebo celé podstromy. [18]



Obrázek 4: Struktura stromu widgetů [18]

Než bude popsán způsob a princip vykreslování UI ve Flutteru, je potřeba definovat princip skládání jednotlivých widgetů dohromady. Jak již bylo zmíněno výše, ve Flutteru je téměř vše widget. Díky tomu, že je například tlačítko widget, a přijímá další potomky typu widget, lze do tlačítka nejen vložit Text (také widget), ale také obrázek. Nebo třeba i pro horizontální a vertikální vycentrování

následujícího widgetu lze použít prvek Center (také widget). Takové zanořování může být v podstatě nekonečné. Problém tohoto přístupu ovšem spočívá v nepřehlednosti, a to, pokud programátor neextrahuje jednotlivé widgety do zvláštních souborů a nedělá dostatečnou abstrakci, např. použitím komponent, tedy znovu-využitelných částí. Pokud není na toto brán při vývoji zřetel, stává se kód velmi nepřehledným a v extrémním případě v podstatě nečitelným.

Aby byl widget vykreslen, je zapotřebí funkce, která ho bude vykreslovat. Tato funkce se ve Flutteru nazývá *build*. Ta definuje, kdy a jakým způsobem se sestaví uživatelské rozhraní. Četnost volání, a tedy i vykreslení, závisí na typu widgetu. Zvolením vhodného typu widgetu lze docílit menšího počtu překreslení. [18]

4.2.3.1 Stateless widget

Jak bylo popsáno výše, widget se vykresluje pomocí *build* funkce. Stateless widget je typ, který v průběhu svého životního cyklu nemění svůj stav. *Build* funkce je tedy volána pouze jednou, a to typicky při vložení do hierarchie. Pokud je potřeba widget znovu vykreslit, musí se celý widget vykreslit znovu, a je tedy zavolána zmíněná funkce znovu. Pokud je potřeba často změnit vnitřní stav widgetu, měl by se spíše zvolit druhý typ. [19]

4.2.3.2 Stateful widget

Než bude popsán tento typ widgetu, je potřeba definovat, co je State a proč je důležitý. *„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.“* [19]

Obrázek 5: State [20]

Stateful widget je na rozdíl od Stateless definován třídou *State* popsanou výše. Samotná třída je však neměnná. Proměnná hodnota je pouze třída *State*, kterou Stateful widget vytvoří pomocí funkce *createState.* Pokud je tento widget použit na více místech, vytvoří se i odpovídající počet *State* objektů. [21]

4.2.3.3 Inherited widget

Jelikož Flutter umožňuje zanořování podle výše popsaného principu, může být hierarchie velmi rozsáhlá. Problém ovšem může nastat, pokud je potřeba přenést data z jednoho widgetu na druhý. Jedním z možných řešení je použití konstruktoru. [22]

Toto řešení ale není vhodné, pokud je potřeba přenést data z widgetu, který je umístěn ve vrchní hierarchii do úplně spodního. V tomto případě by všechny widgety, které v této cestě stojí, měly mít definovaný parametr v konstruktoru. Je jasné, že toto řešení není vhodné, protože při změně struktury dat bychom museli upravit veškeré předávání mezi konstruktory. [22]

Tento problém můžeme vyřešit pomocí Inherited widgetu. Tento widget umožňuje přístup jakémukoli widgetu v hierarchii na data, která obsahuje. Jediná podmínka je, že se musí nacházet nad widgetem, který od něj požaduje přístup. [22]

4.3 Návrhové modely řízení stavu aplikace

Aplikace většinou obsahuje mnoho různých proměnných, aby se udržoval její vnitřní stav (popsáno v sekci 4.2.3.2). Jednat se může o parametry aktuálního výběru filtrace, zda se má zobrazit indikátor načítání (čeká se vykonání webových služeb apod.) V této podkapitole je představen koncept řízení stavu aplikace a možnosti výběru.

Existují dva základní koncepty řízení stavu – lokální a globální. V této sekci jsou stručně popsány rozdíly a doporučení, kterými by se měl vývojář řídit při výběru.

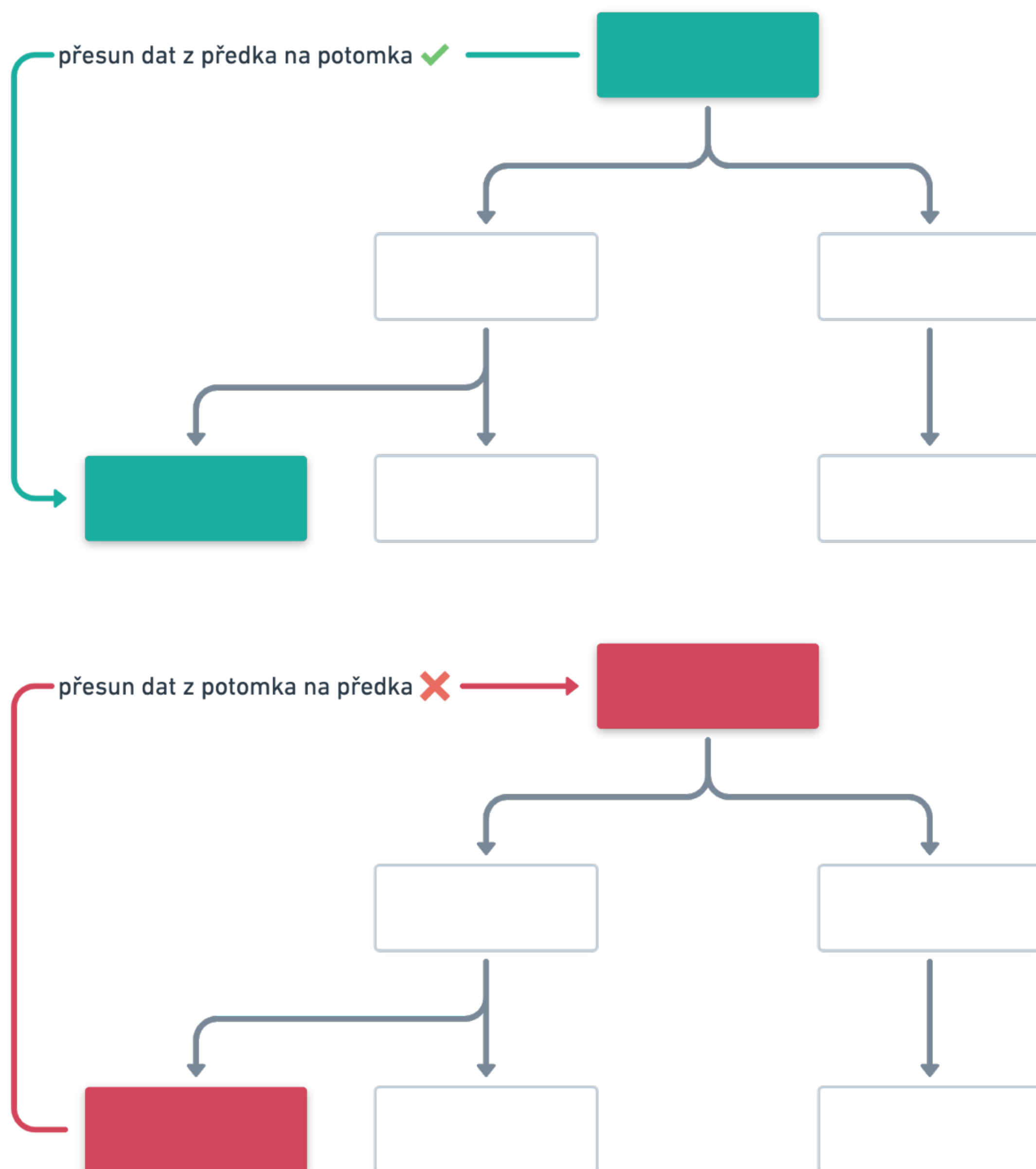
4.3.1 Lokální řízení stavu

V mnoha případech lze zapouzdřit všechna data do jednoho widgetu a lze k nim tedy jednoduše přistupovat. Toto řešení ovšem není vůbec vhodné z důvodu udržitelnosti pro větší aplikace, kde se předpokládá budoucí rozšíření.

Pokud je přesto toto řešení vybráno, lze jednoduše využít *Stateful Widget* a jeho metodu *setState* s callbackem, kde můžeme definovat libovolný počet proměnných, které se mají změnit.

```
var counter = 0;
void zvysHodnotu() {
  setState(() {
    counter++;
  });
}
```

Je vždy dobré oddělit logiku aplikace od samotného zobrazování. Pokud je ale potřeba separovat widgety do více souborů, nastává při použití tohoto řešení problém. Ve Flutteru v hierarchii widgetů data plynou shora dolů. Pokud bychom tedy z potomka chtěli notifikovat svůj rodičovský widget, např. že se nějaká hodnota změnila (např. byl přidán produkt do košíku), nastává problém. Vizualizace této problematiky je zobrazena níže.



Obrázek 6: Předávání dat mezi widgety (vlastní zpracování)

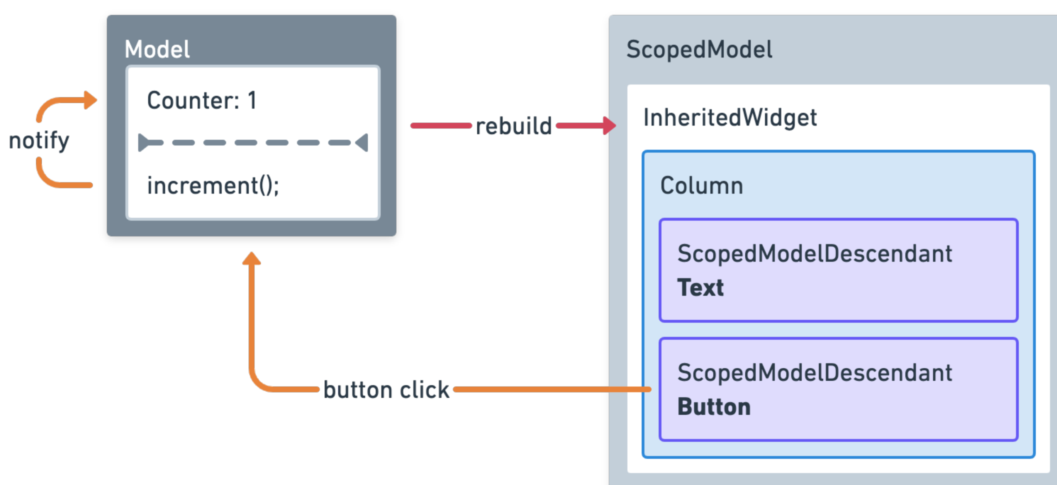
4.3.2 Globální řízení

Většina aplikací bude z výše zmíněných důvodů vyžadovat lepší a udržitelnější řízení stavu. Flutter je velmi flexibilní a nenabízí pouze jednu možnost. Lze tedy implementovat téměř jakýkoli návrhový vzor, od MVP (Model – View – Presenter), po MVVM (Model-View-ViewModel), nebo úplně jiné, méně známé varianty, či zcela vlastní řešení.

Google na svých stránkách zmiňuje některé možné varianty, z nichž lze doporučit ScopedModel, MobX a Business Logic Component (BLoC). [23]

4.3.2.1 Scoped Model

ScopedModel je minimalistický balíček nástrojů, který spíše využívá jen základní funkcionality frameworku Flutter. Hlavním cílem je umožnění předávání dat mezi rodičovským widgetem a jeho potomky (nebo naopak). *„Dělí se na tři hlavní části, z nichž první je Model, což je třída, která drží data a 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.“ [19]*



Obrázek 7: Příklad ScopedModel, vlastní zpracování dle [19]

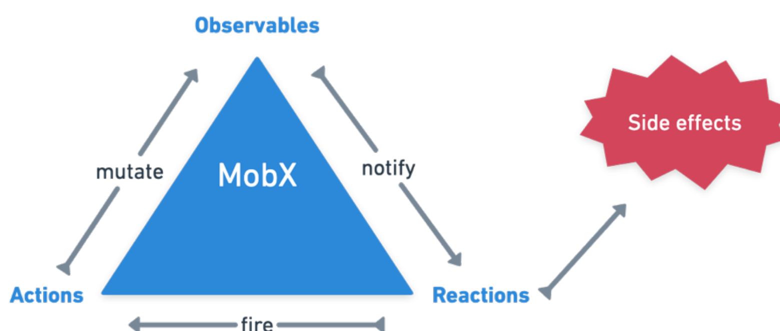
Nevýhodou tohoto přístupu je, že ačkoli se mění jen jediný atribut v objektu specifikovaný v Modelu, je widget poslouchající změny v modelu notifikován, i když se ho změna nemusí přímo týkat. Je tedy sestaven celý widget.

Na druhou stranu je to vhodná volba pro začátečníky a menší aplikace, protože je velmi snadno pochopitelný a nevyžaduje předchozí znalosti návrhových vzorů.

4.3.2.2 MobX

MobX je známé reaktivní řešení pro udržování stavu aplikace. Původně vznikl hlavně jako řešení pro webový vývoj, princip použití je ale na platformě nezávislý. Základním stavebním kamenem je *Store*, který představuje Model (drží data). V této třídě se pak nacházejí *Observables* (tedy jinak Stream) a akční metody, které modifikují hodnoty *Observables*. Dalším důležitým prvkem jsou tzv. *Computed observables*, což je taktéž stream, který ale odkazuje na odvozený stav z jednotlivých *Observables*. Pokud se tedy jedna hodnota z *Observables* změní, zareaguje na to tedy i tento prvek.

Aby se změny patřičně projevily v UI vrstvě, je nutné zaregistrovat tzv. Observer. Ve Flutter je to Widget, kterému je potřeba předat referenci na *Store*. Jakmile se hodnota, kterou sledujeme ve *Store* změní, Observer je notifikován a celý widget a jeho potomci jsou znovu vytvořeny. Flutter MobX je založen na generování souborů pomocí anotací. Tedy definovaná třída slouží pouze jako šablona a samotný kód je schovaný v definované třídě s podtržítkem na začátku. [24]



Obrázek 8: Architektura MobX, vlastní zpracování dle [24]

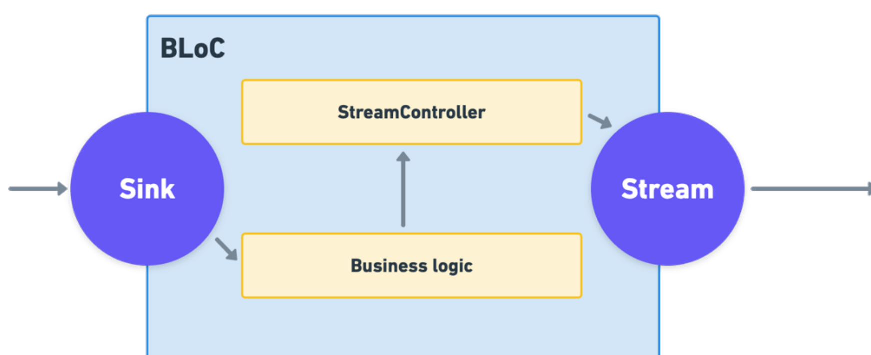
Výhodou MobX je její rozšíření ve světě vývoje webových aplikací. Oproti ScopedModelu dochází také efektivnějšímu znovu-vytváření stromu widgetů.

Možnou nevýhodou je nutnost vygenerování nového souboru *Store* po změně, např. ve jménech proměnných. Pokud by se na toto zapomnělo, změna není reflektována.

4.3.2.3 Business Logic Component (BLoC)

Business Logic Component nebo BLoC je jeden nejdoporučovanějších nástrojů na řízení stavu aplikace. Jako jeden z mála nástrojů pro řízení stavu nevyžaduje žádné závislosti. Lze jej tedy používat hned, jak se vytvoří Flutter projekt. Občas se využívá balíčku RxDart, který přidává další funkcionalitu pro práci se Streamy.

Základem BLoC je již zmíněný StreamController, který pomocí Sinku vkládá data do streamu, který je následně odposloucháván. Velmi podstatným prvkem v celé architektuře je StreamBuilder. Tento widget vyžaduje jako parametr stream pomocí StreamSubscription. Jakmile je emitován prvek ve streamu, StreamBuilder podobně jako MobX Observer popsaný výše vytvoří dané potomky. Obvykle je celá Bloc třída obalena pomocí InheritedWidgetu tak, aby se daný Bloc dal případně jednoduše znovu využít na více místech v aplikaci.



Obrázek 9: Architektura BLoC (vlastní zpracování)

Díky BLoC komponentám lze jednoduše sledovat tok událostí a stavů. Tím se kód stává přehlednější a jednoduše testovatelný. Lze také jednoduše omezit překreslování UI na potřebné minimum. Oproti ostatním přístupům není teoreticky potřeba žádná další závislost na kódu třetích stran.

4.4 Webové služby, JSON serializace a deserializace

Pro přenos dat mezi více zařízeními v síti se používají dva základní protokoly. SOAP a REST. V dnešní době je druhý zmíněný díky jeho flexibilitě oblíbenější a častěji využívaný. REST je protokol, umožňující zasílání dat ve formátu XML, HTTML i v podobě JSON. To je javaskriptová objektová notace, založená ve formátu *klíč:hodnota*. Flutter framework již obsahuje HTTP klienta, který dokáže odeslat požadavek na server a také jej přijímat.

Pro vytvoření Dart třídy z tohoto JSON objektu je možné zvolit několik řešení. Jedním z nich je manuální mapování, které bylo zvoleno i v řešené aplikaci. Druhým možným způsobem je například použití knihovny json_serializable[9], která dokáže mapování vytvořit pomocí generátoru. Generátor vytvoří nový soubor s příponou *.g.dart,* což značí vygenerovaný soubor.

4.5 Firebase služby

Firebase[10] je produkt vlastněný společností Google, který poskytuje mnoho služeb pro ulehčení vývoje jak mobilních, tak i webových aplikací. Tyto služby se dají rozřadit do tří kategorií – vývojové, analytické a podpůrné. V této sekci budou rozebrány pouze ty, které jsou popsány v následujících kapitolách.

4.5.1 Firestore

Mezi vývojové služby lze zařadit Firestore, což je BaaS[11] služba poskytující databázi ve formě dokumentů, které se pomocí kolekcí zanořují a částečně tak mohou připomínat NoSQL přístup. Oproti jiným databázím může tato služba nabídnout promítání změn v databázi v reálném čase. To v praxi znamená, že jakmile se v jakémkoli dokumentu uvnitř libovolné kolekce stane změna (například přidání nové hodnoty nebo změna již stávající), okamžitě je tato změna

---

[9] Json_serializable – dostupný z https://pub.dev/packages/json_serializable
[10] Firebase – dostupný z https://www.firebase.com
[11] BaaS – backend as a service

promítnuta do všech klientů, které tyto databázové změny poslouchají. Tím je zajištěna konzistence mezi databází a klientem.

Výhodou je i řešení stavu, kdy zařízení nemá na kratší časový úsek k dispozici internetové připojení. Pokud tento stav nastane, je zajištěno cachování[12]. [25]

Tato služba je pro určité množství databázových dotazů zdarma (jedná se padesát tisíc dotazů na čtení a dvacet tisíc zápisů na den)[13]. Pokud je tento limit překročen, je nutné zvolit placenou variantu. Lze buď platit pevný měsíční poplatek nebo zvolit variantu *„pay as you go“,* tedy variantu, kdy se platí podle počtu požadavků za konkrétní měsíc.

4.5.2 Authentication

Další služba, která je využita v aplikaci, je Firebase Authentication. Ta poskytuje jednoduché rozhraní pro implementaci přihlašování jak pomocí emailu a hesla, tak i pomocí sociálních sítí. Velkou výhodou je možnost anonymního přihlášení. To uživateli vygeneruje jedinečný identifikátor, který lze pak použít pro jeho identifikaci. Výhodou této služby je pak následné propojení anonymního uživatele s přihlášeným. Tedy jednotlivé identifikátory se po přihlášení propojí a nevznikne duplicita stejných uživatelů. Služba je zdarma do deseti tisíc ověření za měsíc. [26]

4.5.3 Administrace

Firebase také umožňuje správu všech služeb v administraci. Lze tedy přidávat nebo upravovat data, přidávat nebo odebírat uživatele, posílat notifikace na vybraná zařízení, to vše bez pokročilé znalosti IT.

---

[12] cachování – načtení výsledků databázového dotazu do mezipaměti zařízení
[13] Firestore cena – více informací zde: https://firebase.google.com/pricing

5 Návrh mobilní aplikace

V této kapitole jsou popsány požadavky, struktura, návrh architektury aplikace a jednotlivé popisy obrazovek. Mimo to jsou zde také představeny služby pro komunikaci s datovou vrstvou, které byly uvedeny v předchozích kapitolách. V závěru jsou popsány problémy, které při vývoji nastaly, a jejich následné řešení.

5.1 Požadavky

Při vytváření jakékoliv aplikace je nutné mít představu o tom, jaké funkce by měla mít a jaké požadavky splňovat. Tímto procesem si vývojář shrne hlavní myšlenky a samotný návrh obrazovek je poté snadněji uchopitelný.

5.1.1 Funkční požadavky

5.1.1.1 Průvodce aplikací

Při prvním spuštění aplikace se uživateli zobrazí stručný průvodce aplikací, který mu vysvětlí kroky a akce, které mu jsou v aplikaci umožněny. Uživatel bude moci povolit nebo případně zakázat zasílání notifikací.

5.1.1.2 Přihlášení do aplikace

Jedna z podmínek při vývoji aplikace byla umožnit její užívání jak přihlášenému, tak i nepřihlášenému uživateli. Aby však šly jednotlivé uložené události přiřadit k danému uživateli, bude nutné vytvořit univerzální unikátní identifikátor (UUID), který anonymně identifikuje uživatele.

5.1.1.3 Výpis všech událostí

Po spuštění by se uživateli měly zobrazit všechny události s vypnutým filtrem fakult.

5.1.1.4 Filtrace dle fakult

Aplikace by měla umět filtrovat výpis všech událostí dle fakult tak, že uživatel by měl mít možnost filtrovat z více než jedné fakulty najednou.

5.1.1.5 Detail události

Aplikace by měla zobrazit detail události dle typu. Pokud se jedná o konferenční událost, aplikace by měla zobrazit novou obrazovku s popisem a programem akce. Pokud se jedná o akci bez detailu, měly by se zobrazit jen popis a datum události.

5.1.1.6 Přijetí notifikace o nově přidané události

Aplikaci bude umožněno přijímat notifikace. Po přijetí této notifikace by se měl aktualizovat seznam všech událostí a zobrazit detail.

5.1.1.7 Přidání konkrétní události do svého programu

U konferenčního typu události by měl uživatel mít možnost uložit konkrétní událost do svého programu.

5.1.1.8 Načtení mapy s budovami UHK

Uživatel by měl vidět důležité budovy UHK na mapě. Po kliknutí na konkrétní pin by se měla otevřít nabídka aplikací, která ho k budově dokáže navigovat.

5.1.1.9 Zobrazení detailu programu

Po kliknutí na položku programu by se měla zobrazit nová obrazovka s detailnějšími informacemi o události. Událost by měla jít přidat do uživatelova programu.

5.1.2 Non-funkční požadavky

Non-funkční požadavky byly stanoveny takto:

- Aplikace bude na iOS zařízeních dostupná od verze 12.0 a na Androidu od verze 5.1.
- Aplikace bude primárně určena pro mobilní telefony.
- Aplikace bude podporovat český i anglický jazyk dle nastaveného jazyka v zařízení.
- Aplikace bude ukládat výpis všech událostí do NoSQL databáze.
- Aplikace bude čerpat data o událostech z webu UHK a data z konferenčních akcí bude čerpat z Firebase Firestore databáze.
- Struktura aplikace bude škálovatelná pro další rozšiřování.

5.2 Návrh obrazovek

Po setřídění myšlenek o funkčnosti aplikace je důležité tyto informace správně zobrazit, aby byly pro uživatele dobře pochopitelné. K tomuto účelu se používá drátěný model (wireframes[14]). K návrhu byl použit program Whimsical[15]. Tento program nabízí jednoduché rozhraní pro tvorbu wireframes a je pro studenty zdarma k dispozici.

Nejdříve bylo nutné vytvořit tzv. domovskou obrazovku, od které se odvíjí další funkcionalita. Poté bylo nutné rozmyslet, jak budou rozlišeny dva typy událostí – konferenční typ událostí (tedy ta, která obsahuje další informace o programu) a následně jak „klasický typ“ (událost bez dalšího programu). Po konzultaci se zadavateli z PR oddělení, kteří aplikaci pro zobrazování událostí vymysleli, byl tento problém vyřešen pomocí zobrazení další obrazovky s programem a dalšími

[14] Wireframe – označení pro černobílý návrh uživatelského rozhraní webových stránek nebo mobilní aplikace za pomocí definovaných prvků a komponent

[15] Whimsical – Produkt pro vytváření myšlenkových map a wifreframů. Dostupný na https://whimsical.com/

informacemi. V druhém případě pak pomocí obyčejného dialogu. Tím byla zajištěna přehlednost pro uživatele.

Jelikož jeden z požadavků byla snadno přístupná filtrace dle fakult, byla vložena hned na hlavní obrazovku pomocí log jednotlivých fakult.
Důležitá část drátěného modelu pak byla navigace mezi obrazovkami tak, aby se uživatel mohl snadno orientovat. K tomuto účelu byl využit dolní navigační bar, který slouží k přepínání mezi jednotlivými obrazovkami konferenční části. Drátěný model i s ukázkami obrazovek je zahrnut v kapitole 12.

Po drátěném modelu bylo nutné tento černobílý návrh validovat a vytvořit z něj grafické podklady. Při tvorbě této aplikace byl kladen důraz na jednoduchý, ale svěží vizuální styl, který splňuje požadavky na moderní mobilní aplikace. Graficky se inspiruje univerzitním webem tak, aby splňoval normy grafického manuálu vytvořeného pro UHK, což byla i jedna z podmínek při vytváření aplikace. Pro vytvoření grafických podkladů byl využit program Adobe XD[16]. Tento program je primárně určen pro tvorbu grafiky pro webové a mobilní projekty a je zdarma. Ukázky vytvořených grafických podkladů jsou uvedeny v kapitole 12.

5.3 Nástroje pro ulehčení vývoje

Před samotným vývojem bylo nutné zvolit patřičné knihovny a nástroje, které by urychlily vývoj. Také bylo potřeba vybrat vhodný nástroj pro udržování stavu aplikace.

5.3.1 Volba návrhového vzoru

Pro vývoj aplikace byl v této práci vybrán návrhový vzor BLoC. Díky tomuto návrhovému vzoru lze jednotlivé komponenty snadno rozdělit na menší části a spravovat tak dílčí stavy v rámci malých komponent.

---

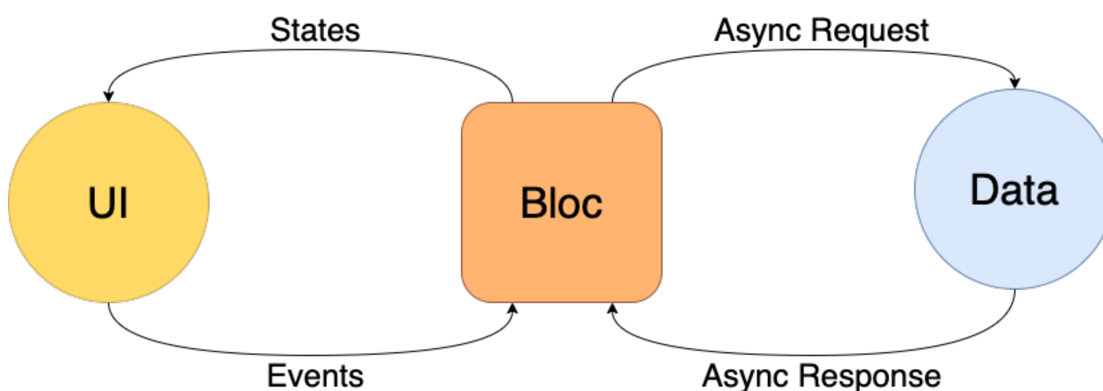
[16] Adobe XD – dostupný z https://www.adobe.com/products/xd.html

Pro ulehčení vývoje byla zvolena knihovna *flutter_bloc*, která abstrahuje nutnou část kódu a práce se Streamy. Využití této knihovny ulehčí vývoj a ušetří velké množství práce. Dále byl použit Inversion of Control kontejner, který se stará o udržování servisních a jiných tříd. Většina ukázkových aplikací kombinuje uvedené přístupy dohromady.

5.3.1.1 flutter_bloc

Knihovna čerpá z výše uvedených pravidel návrhového vzoru Bloc. Je částečně postavena nad reaktivní knihovnou RxDart[17], která přidává funkcionalitu pro práci se Streamy. Aby byl vývojář ušetřen starostí nad implementací Streamů a jejich kontrolerů, je knihovna omezena na události (Events), stavy (States). Dále je nutné aplikovat přechody mezi událostmi a stavy, tzv. přechody (Transitions) a samotný Bloc, který tyto prvky propojuje.

V praxi je to poměrně jednoduché. Pokud uživatel v UI stiskne tlačítko, vyvolá se událost, která popisuje daný stisk tlačítka. Tuto událost Bloc zaregistruje a vykoná patřičné operace. Ve chvíli, kdy je operace vykonána, Bloc vrátí nový stav. Tím vznikne přechod mezi starým a novým stavem, na který může aplikace patřičně zareagovat. [27]



Obrázek 10: Náhled na flutter_bloc architekturu [27]

---

[17] RxDart – dostupný z https://www.pub.dev/packages/rxdart

Jak je prezentováno na obrázku výše, v Bloc třídě vše probíhá asynchronně. Lze tedy i v průběhu zpracování dat v třetí vrstvě (Data) měnit stav aplikace. Pokud např. uživatel klikne na tlačítko, okamžitě se zobrazí indikátor načítání. Jakmile jsou data k dispozici, je možné vrátit načtená data.

5.3.2 Inversion of Control

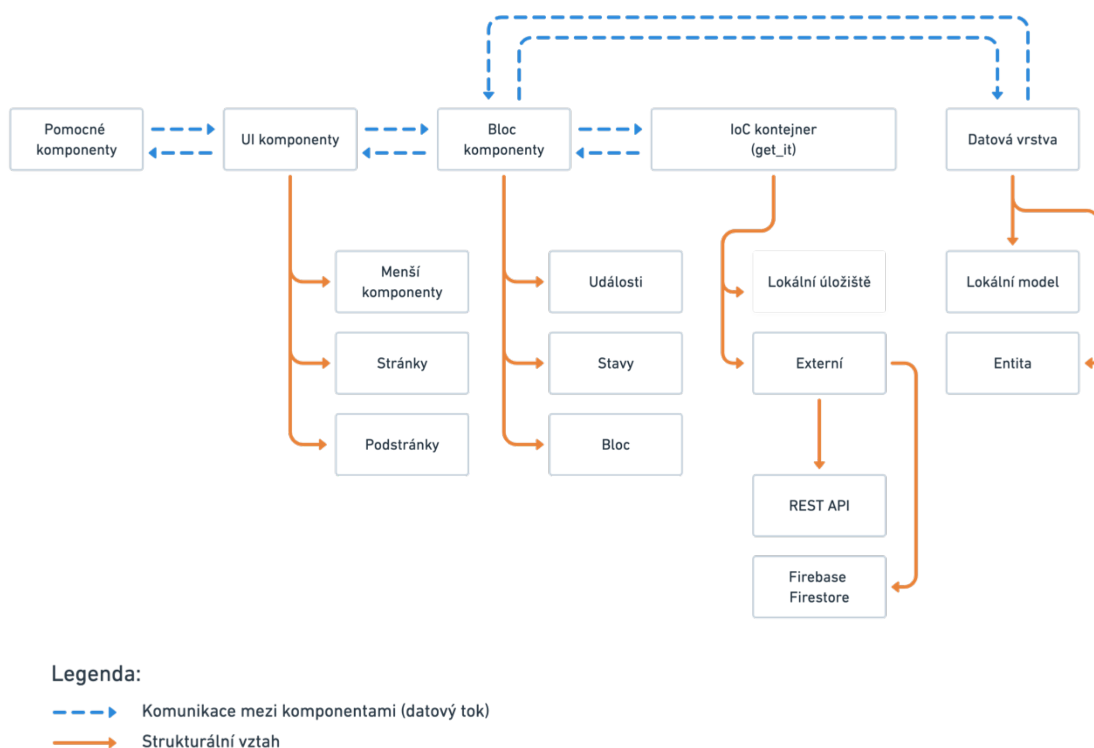
Dalším využitým konceptem je Inversion of Control. *„Inversion of Control (IoC) je programátorský princip, kdy dochází k inverzi řídícího toku programu. V principu při vývoji dochází k tomu, že je oddělen interface (v případě Dartu abstraktní bázová třída slouží jako interface) od samotné implementující třídy a zároveň je umožněno odkudkoliv z aplikace přistupovat ke konkrétní implementaci prostřednictvím onoho definujícího interface.“ [19]*

Tento přístup umožňuje vytvořit servisní třídy na jednom místě (v kontejneru) a následně požádat o navrácení instance už s konkrétními závislostmi na jiné třídy.

Pro využití IoC byla zvolena knihovna *get_it*[18], kterou lze použít v jakékoli Dart aplikaci.

[18] get_it – dostupný z https://pub.dev/packages/get_it

5.4 Návrh architektury

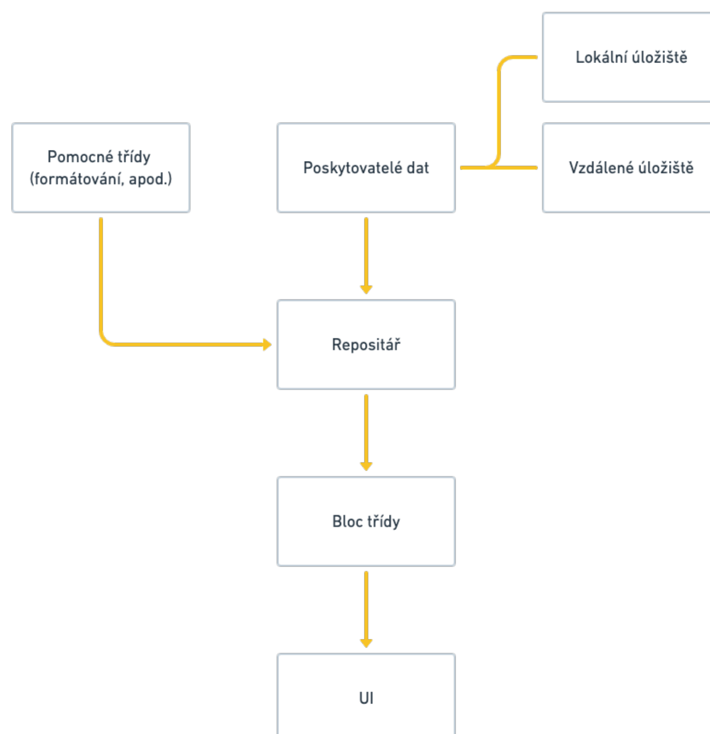


Obrázek 11: Náhled na architekturu aplikace, vlastní zpracování dle [19]

Jak vyplývá z BLoC architektury, kód lze v podstatě rozdělit na tři základní stavební kameny. Zobrazovací, řešení stavů a backend, což je ve smyslu mobilních aplikací například odesílání požadavků na server, databázové dotazy a mapování tříd.

Aplikace je z velké části závislá na zpracování dat, proto bylo vhodné oddělit datové třídy a entity. Entita je třída, která přesně koresponduje s vrácenými daty ze serveru. Datový model pak data z této (nebo více) entit transformuje do informací tak, aby byla snadno zobrazitelná v UI vrstvě. Pokud by vnikla potřeba změny v návratovém modelu, zobrazovací logika zůstane téměř stejná, pouze se změní patřičné klíče v entitě.

S *backend* vrstvou zásadně komunikují pouze Bloc třídy, které zajišťují a spravují veškerou aplikační logiku. Bloc komponenta dostane přidělenu závislost na daný repositář, spravující operace nad daty pomocí get_it kontejneru. Pokud je potřeba, lze pro Bloc komponentu vytvořit ještě další závislosti, například na třídy spravující formátování data, ukládání do lokální paměti apod.



Obrázek 12: Náhled architektury datového modulu (vlastní zpracování)

Bloc komponenty jsou rozděleny do logických celků tak, aby se nehromadilo zbytečně moc informací v komponentě, která daná data nepotřebuje. S Bloc komponentami komunikuje pouze UI vrstva. Ta do Bloc komponenty vyšle událost, která je zpracována, a zpět do zobrazovací vrstvy je vrácen stav s daty, která se mají zobrazit. Zobrazovací vrstva je vždy rozdělena na menší widgety, a to dle jejich zodpovědnosti a logického celku.

Architektura může na první pohled vypadat poměrně rozsáhle. Kvůli jedné Bloc třídě je třeba definovat všechny možné události, které mohou nastat (nejen z UI) a následně k nim vytvořit patřičné reakční stavy. Celek je pak nutné propojit v Bloc

třídě, která transformuje danou událost na stav. Na druhou stranu toto řešení umožňuje rychle se zorientovat v tom, co se v konkrétní chvíli v aplikaci děje, např. po stisknutí tlačítka. Pokud se rozhodneme navrátit k vývoji po delší době, lze se rovněž snadněji zorientovat, protože přesný výčet všech stavů je již znám.

5.5 Struktura aplikace

Cílem bylo navrhnout strukturu aplikace tak, aby jednotlivé části byly co nejvíce oddělené a byla do budoucna snadno rozšiřitelná. Aplikace je tedy rozdělena do tří balíčků na základě funkčnosti. Kód Flutter projektu lze najít v adresáři *lib*. Uvnitř tohoto adresáře jsou umístěny tři podadresáře. Jedná se o zobrazovací modul (*ui*), modul pro komunikaci a správu dat (*io*) a modul společného jádra (*common*).

5.5.1 Zobrazovací vrstva

Zobrazovací modul obsahuje pouze tu část aplikace, která je zodpovědná za vykreslování Widgetů na obrazovku. V *ui* složce se nachází čtyři další podsložky. *Main* složka, která se následně dělí dle typu události, *onboarding* (průvodce aplikací), *shared* (pro sdílené ui komponenty napříč aplikací) a *splashscreen*[19] obrazovka. Každá složka se dělí následně na složku se stránkou, použitými widgety a Bloc třídami s událostmi a stavy.

5.5.2 Vrstva práce s daty

Dalším modulem je *io*, který slouží pro komunikaci a správu dat. Obsahuje modelové a transformační třídy pro komunikaci pomocí HTTP požadavků (anglicky requestů) a pro získávání dat z Firebase Firestore. V aplikaci je využíván *repository pattern*[20], proto se zde nachází i balíček pro tento návrhový vzor. Modelové třídy jsou rozděleny do dvou kategorií – *entity* a *model.* Entitní třídy slouží pouze pro parsování JSON objektů. Z nich se pak dále vytváří modelové třídy, které se používají v zobrazovací vrstvě.

[19] Splashcreen – obrazovka viditelná ihned po spuštění aplikace
[20] Repository pattern – abstrakce mezi vrstvou pro přístup k datům a vrstvy logiky aplikace [28]

5.5.3 Společné jádro aplikace

Pro oddělení tříd, které jsou použity napříč celou aplikací, byl vytvořen balíček jádra (common). Tento balíček obsahuje třídy jako jsou konstanty, barvy a například pomocné metody pro formátování data. Pro zpřehlednění kódu je vložen balíček s *extensions*, a dále balíček *error* s třídami pro odchytávání chybových stavů a výjimek. Důležité je zmínit *managers*, což je balíček obsahující třídy pro správu notifikací a indikace stavu, kdy zařízení není připojené k internetu. Posledním balíčkem je *assets*, jež není součástí vygenerovaného projektu a bylo nutné ho manuálně vytvořit. V tomto balíčku se nachází v jednotlivých podsložkách ikony, fonty a lokalizační soubory. Aplikace v tuto chvíli obsahuje dvě lokalizace – český a anglický jazyk.

5.6 Popis implementace

V této sekci jsou popsány důležité třídy a soubory, které zajišťují chod aplikace. Dále jsou uvedeny jednotlivé implementační detaily, jež jsou nutné k pochopení architektury.

5.6.1 Spuštění aplikace a registrace tříd

Výchozím souborem pro spuštění aplikace je *main.dart.* V tomto souboru se nachází víc tříd a metod, které jsou třeba ke správnému spuštění aplikace.

```
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  _initServiceLocator();

  runApp(
    BlocProvider(
      create: (context) => injector<AuthenticationBloc>()
        ..add(AppStarted()),
      child: EventsApp(),
    ),
  );
}
```

Hned v úvodu je nutné zmínit zavolání *WidgetsFlutterBinding.ensureInitialized()*. Tato statická metoda zajišťuje to, aby se před spuštěním aplikace inicializovali všechny potřebné pluginy, které jsou definované v *pubspec.yaml* souboru. Pokud by se vynechala, aplikace by skončila chybou.

Důležitá je také inicializace kontejneru pro servisní třídy *_initServiceLocator().* V této metodě se inicializují třídy, které se pak používají napříč komponentami. V ukázce níže se inicializuje AuthenticationBloc, který má závislost na repositář, nutný pro práci s uživatelskými daty. Tento Bloc určuje to, jaká obrazovka se má při spuštění zobrazit (průvodce nebo už hlavní obrazovka). Tato logika je zajištěna díky uloženému UUID uživatele.

```
final injector = GetIt.instance;

Future<void> _initServiceLocator() async {

  injector.registerFactory<AuthenticationBloc>(
        () => AuthenticationBloc(userRepository: injector()),
  );
}
```

Úvodní *EventsApp* je StatefulWidget. Jak již bylo zmíněno v úvodních kapitolách, tento widget jen vytváří privátní třídu s příponou State.

```
class _EventsAppState extends State<EventsApp> {
  @override
  Widget build(BuildContext context) =>
      MaterialApp(
          title: 'UHK Events',
          debugShowCheckedModeBanner: isDebug,
          theme: theme,
          localizationsDelegates: [
            FlutterI18nDelegate(fallbackFile: 'cs', path:
            'assets/i18n'),
          ],
          home: // níže
      );
}
```

Prvním widgetem ve stromu je MaterialApp. Toto je widget, který vychází z Material designu. Tento designový jazyk je primárně určen pro Android aplikace, může se ale používat napříč platformami. Atributů, které má tento widget definován, je mnoho. Jedním z nich je *debugShowCheckedModeBanner.* Tomu můžeme předat informaci o tom, zda je aplikace v tzv. debugovacím módu. To je stav, kdy je aplikace vhodná pro ladění chyb. Pokud ano, je v pravém horním rohu

zobrazen štítek s touto informací. Tak lze jednoduše odlišit dvě stejné aplikace, které se například liší pouze jinou URL adresou pro příjem dat.

Dalším atributem je *theme*. Theme je tématické schéma v celé aplikaci. Lze zde specifikovat velikosti písem, jejich barvu, rozložení apod. Kromě písem lze vytvářet podobu dalších komponent, např. tlačítek apod. K tomu lze použít již přednastavený *Material (Android)* nebo Cupertino (iOS) styl, nebo vytvořit vlastní (jako u řešené aplikace).

Výše bylo zmíněno, že aplikace podporuje vícejazyčnost. Ta je zajištěna delegátem *FlutterI18nDelegate*, kterému je předána složka s jednotlivými i18n[21] soubory s texty. Lze zvolit formát JSON nebo YAML. V tomto případě byl vybrán jednodušší JSON. Podle klíče daného textu je pak v souboru nalezena jeho hodnota dle aktuální lokalizace zařízení.

Důležitým a klíčovým atributem je *home* datového typu Widget.

```
home: BlocBuilder<AuthenticationBloc, AuthenticationState>(
  builder: (context, state) {
    if (state is Uninitialized) {
      return OnboardingView();
    } else if (state is SplashScreen) {
      return SplashScreenView();
    } else {
      return BlocProvider(
        create: (context) => injector<EventFilteredBloc>(),
        child: HomeView(),
      );
    };
  },),);
```

V tomto případě jde o BlocBuilder. To je widget, který reaguje na změny v AuthenticationBloc komponentě. Zde se nachází rozhodovací logika o tom, která obrazovka se má zobrazit po spuštění aplikace. Informace o spuštění byla

[21] I18n – internalizace a lokalizace aplikace pro vícejazyčnost

zaregistrována výše v *main.dart* pomocí události *AppStarted()*. Může tedy dojít k zobrazení:

- Onbarding (průvodce aplikací) - pokud uživatel spustil aplikaci poprvé,
- SplashScreen (úvodní obrázek) - mezi tím, než uživatel čeká na stav výše,
- HomeView (hlavní obrazovka) - pokud již uživatel má vygenerovaný identifikátor.

5.6.2 Přihlašování pomocí Firebase Authentication

Pro komunikaci s knihovnou Firebase Authentication slouží stejně pojmenovaný plugin[22]. Ten zajišťuje totožné možnosti, jako při nativním vývoji nebo vývoji pro web. Pro abstrakci byla vytvořena třída *FirebaseAuthProvider*, což zajišťuje práci s třídou pluginu *FirebaseAuth*. Závislost na této třídě je taktéž definovaná v kontejneru *get_it* a byla poskytnuta této třídě. Po proběhnutí metody *signInAnonymously()* dojde k vytvoření účtu a vrácení uživatelského identifikátoru. Důležité je použití metody *isSignedIn(),* která zjišťuje, zda je již účet vytvořený, a která v konečném důsledku zajišťuje logiku zobrazení správné obrazovky.

[22] Firebase Authentication – dostupný z https://pub.dev/packages/firebase_auth

```
class FirebaseAuthProvider extends AuthProvider {

  final FirebaseAuth firebaseAuth;
  FirebaseAuthProvider({@required this.firebaseAuth});

  @override
  Future<void> signInAnonymously() async {
    return firebaseAuth.signInAnonymously();
  }

  @override
  Future<bool> isSignedIn() async {
    return getUserId() != null;
  }

  @override
  Future<String> getUserId() async {
    return firebaseAuth
        .currentUser()
        .uid;
  }
}
```

5.6.3 Komunikace s Firebase Firestore

Stejně jako pro práci s autentizací uživatele existuje balíček pro práci s databází Firestore[23]. Podobně jako při autentizaci uživatele byla vytvořena servisní třída FirestoreProvider.

Databáze Firestore slouží pouze pro konferenční typy událostí. Pro rozhodnutí, zda jde o konferenční událost, je nutné zjistit, jestli se identifikátor, poskytnutý z výpisu událostí, nachází i ve Firestore databázi. To je zajištěno pomocí metody *isConferenceType.* Prostřednictvím ní dochází ke kontrole existence dokumentu s daným identifikátorem. Výsledkem je navrácení hodnoty true nebo false.

[23] Firebase Firestore – dostupný z https://pub.dev/packages/cloud_firestore

```
class FirestoreProvider {
  final Firestore firestore;
  FirestoreProvider({@required this.firestore});

  Future<bool> isConferenceType(String id) async {
    final snapShot = await firestore
        .collection('events')
        .document(id)
        .get();
    return snapShot != null && snapShot.exists;
  }
}
```

5.6.4 Komunikace se REST API, deserializace JSON dat

Flutter nabízí velké množství balíčků třetích stran pro komunikaci s REST API. Jelikož se v řešené aplikaci používá pouze jeden požadavek typu HTTP Get, nebylo nutné takový externí balíček využít.

Dotaz na server zajišťuje třída *ApiProvider* a dědí ho z abstraktní třídy pro snadnější pozdější testování. Závislost na balíčku, zajišťujícím práci s http požadavky, je taktéž poskytnuta pomocí konstruktoru z kontejneru závislostí.

```
class ApiProvider extends Api {
  final String url = 'https://www.uhk.cz/cs/kalendar/…'
  final http.Client client;
  ApiProvider({@required this.client});

  @override
  Future<List<EventItemEntity>> getEventList() async {
    final Response response = await client.get('$url);
    final List<dynamic> list = jsonDecode(response.body);
    final eventEntities = list.map((item) =>
        EventItemEntity.fromJson(item)).toList();
    return eventEntities;
  }
}
```

Důležitou funkcí je *jsonDecode* – *ta* z textového řetězce JSON souboru vytvoří hash mapu, se kterou lze dále pracovat. Samotné „namapování“ na objekt entity probíhá o řádek níže, kdy se v každé iteraci pomocí factory metody *fromJson* vytvoří objekt.

```
factory EventItemEntity.fromJson(Map<String, dynamic> json) =>
    EventItemEntity(
      id: json['id'],
      publishDate: json['publishDate'],
      singleEventStart: json['singleEventStart'],
      facultyTitle: json['facultyTitle'],
      facultyType: json['facultyType'],
      eventTag: json['eventTag'],
      eventTitle: json['eventTitle'],
      eventUrl: json['eventUrl'],
    );
```

5.6.5 Práce s repositáři

Tyto poskytovatele dat, ať už ApiProvider nebo FirestoreProvider, je dobré spojit do jedné třídy tak, aby agregovala data a poskytovala je dál do Bloc tříd. Tím se zajistí lepší udržitelnost, pokud se jeden poskytovatel dat nahradí jiným. Jak pro práci s uživatelskými daty, tak i pro práci s událostmi byla vytvořena třída repositáře.

```
class EventRepositoryImpl extends EventRepository {
  final ApiProvider apiProvider;
  final FirestoreProvider firestoreProvider;
  final AppPreferences localDataSource;
  final NetworkInfo networkInfo;

  EventRepositoryImpl({@required this.apiProvider,
    @required this.firestoreProvider,
    @required this.localDataSource,
    @required this.networkInfo});

  @override
  Future<List<MainEventItem>> fetchScheduleFromEvent(String
eventId) =>
      firestoreProvider.fetchScheduleFromEvent(eventId);
}
```

V repositáři kromě jiného probíhá kontrola, zda je uživatel připojený k internetu. Na základě ní se vybere správný poskytovatel dat – buď lokální nebo vzdálené úložiště dat. Tato logika je „schovaná“ a vývojáři pouze stačí zavolat danou metodu z repositáře s definovanou návratovou hodnotou.

5.6.6 Bloc třídy – funkční logika

Po dokončení funkcionality repositáře je nutné okomentovat funkční logiku, kterou zajišťují právě Bloc třídy. Tyto třídy je dobré dělit buď dle funkčního celku nebo dle obrazovky. Jak bylo zmíněno, *flutter_bloc* se skládá ze 3 celků – stavů, událostí a tříd, které tyto stavy a události zpracovávají. Bloc třída je demonstrována na nejdůležitější části aplikace. To je načítání událostí, tedy *EventsFilteredBloc*.

5.6.6.1 Stavy

Stav je třída reprezentující všechny stavy, do kterých se aplikace v daném Blocu může dostat. Díky omezenému výčtu se musí aplikace vždy nacházet v jednom z definovaných stavů.

Pro snadnější předávání je dobré mít definovanou třídu jako předka všech stavů v daném Blocu. Tato třída by měla sloužit pouze jako předpis bez další definice logiky.

```
abstract class EventFilteredState {
  const EventFilteredState();
}

class FilteredEventsLoading extends EventFilteredState {}

class FilteredEventsError extends EventFilteredState {}

class FilteredEventsLoaded extends EventFilteredState {
  final List<EventItem> events;
  final List<Faculty> faculties;

  FilteredEventsLoaded(this.events, this.faculties);
}
```

První dva stavy neposkytují žádná další data. Nelze je ale nijak zaměnit, protože každá třída zajišťuje reprezentaci toho, co se má zobrazit (načítání nebo zobrazení chyby). Poslední stav předává pole všech načtených událostí a fakult. Tato data se pak následně přepošlou do jednotlivých widgetů k zobrazení.

5.6.6.2 Události

Události definují přechody mezi stavy. Pod událostí si lze představit zmáčknutí tlačítka nebo reakci na chybovou událost, která se může dále řetězit. Pro všechny události v definovaném Blocu platí definice stejného předka.

```
abstract class EventFilteredEvent {
  const EventFilteredEvent();
}

class UpdateFilter extends EventFilteredEvent {
  final Faculty faculty;
  UpdateFilter(this.faculty);
}

class UpdateEvents extends EventFilteredEvent {
  final List<EventItem> events;
  const UpdateEvents(this.events);
}
```

V tomto případě jde pouze o reakci na nějaký stav (stisk tlačítka nebo indikace načtení událostí). V události *UpdateFilter* se předá jako parametr aktuálně vybraná fakulta, která se má přidat nebo odebrat do filtru. Zda má jít o přidání nebo odebrání z filtru je pak realizováno v samotném Blocu.

5.6.6.3 Bloc

Pro propojení stavů a událostí slouží Bloc třídy, které využívají parametrizovanou Bloc třídu, kde jsou parametry událost a stav. Bloc má na starost udržení stavu. V reakci na událost pak vytvoří konkrétní stav. Bloc třídy mohou mít i závislost na dalším Blocu. V tomto případě jde o EventsBloc, který pouze načte události a předá je dále do FilteredEventsBlocu, který již zajišťuje práci s daty (např. filtrování).

```
class EventFilteredBloc extends Bloc<EventFilteredEvent,
EventFilteredState> {

  final EventsBloc eventsBloc;
  StreamSubscription eventsSubscription;
}
```

K propojení události a stavu slouží metoda, kterou je nutné implementovat. Ta se nazývá *mapEventToState* a vrací Stream stavů. To je velmi výhodné, protože díky asynchronnímu přístupu lze měnit jednotlivé stavy i opakovaně.

```
@override
Stream<EventFilteredState> mapEventToState(
    EventFilteredEvent event) async* {
  if (event is UpdateFilter) {
    yield* _mapUpdateFilterToState(event);
  } else if (event is UpdateEvents) {
    yield* _mapEventsUpdatedToState(event);
  } else {
    yield FilteredEventsError();
  }
}
```

V této ukázce je demonstrováno, že po reakci na událost se volá metoda, ve které je ukryta logika, což zpřehlední kód. Logika vrácení stavu po kliknutí na filtr fakulty je ukázána níže.

```
Stream<EventFilteredState> _mapUpdateFilterToState(
    UpdateFilter event) async* {
  final events = (eventsBloc.state as EventsLoaded).events;

  final faculties = state is FilteredEventsLoaded
      ? (state as FilteredEventsLoaded).faculties
      : const <Faculty>[];
  final filter = _addOrRemoveFilterFaculty(faculties,
event.faculty);

  yield FilteredEventsLoaded(
      _mapEventsToFilteredEvents(events, filter), filter);
}
```

Nejdříve se z *EventBlocu* do proměnné uloží všechny události. Poté se zkontroluje, jaké filtry (fakulty) má uživatel aktuálně zapnuté. Následněse dle toho, zda se daná fakulta již ve filtru nachází nebo ne, vrátí nové pole. Za toto rozhodnutí je zodpovědná prostá funkce *_addOrRemoveFilterFaculty,* která buď položku odebere, nebo přidá. V dalším kroku je dle filtru zavolána metoda *_mapEventsToFilteredList.*

```
List<EventItem> _mapEventsToFilteredEvents(
     List<EventItem> events,
     List<Faculty> filter) {

  if (filter.isEmpty) return events;
  return events.where((e) {
    return filter.contains(e.faculty);
  }).toList();
}
```

Ta provede samotné vyfiltrování. Poté je již vrácen *FilteredEventsLoaded* stav s vyfiltrovanými událostmi a aktuálně zapnutým filtrem.

5.7 Uživatelské rozhraní

V této podkapitole je popsán průběh vývoje pro uživatele nejdůležitější části, tj. obrazovky. V návaznosti na předchozí podkapitoly byla pro ilustraci zvolena hlavní obrazovka s výpisem událostí.

V rámci návrhového vzoru Bloc byl vybrán StatelessWidget, protože stav je dodáván zvenčí pomocí Blocu a není nutné měnit vnitřní stav zevnitř Widgetu. StatefulWidget není v aplikaci téměř využitý.

Důležitým prvkem je také *BlocBuilder*, který je zodpovědný za zobrazení správného widgetu dle stavu – načítání, chybový stav (prázdný seznam) a konečně seznam všech událostí. To je demonstrováno v ukázce níže.

```
class HomeView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: EventAppBar(context.translate("appTitle"),
          actions: [_FacultyFilterButtons()]),
      body: BlocBuilder<EventFilteredBloc,EventFilteredState>(
        builder: (context, state) {
          if (state is FilteredEventsLoading) {
            return _LoadingList();
          } else if (state is FilteredEventsLoaded) {
            return _EventListView(items: state.events);
          } else {
            return _EmptyEventList();
          }
        },
      ),
    );
  }
}
```

Horní lišta (*EventppBar*), kde je zobrazen název dané obrazovky, je v celé aplikaci téměř shodná. Z důvodu znovupoužitelnosti byla vyextrahována do samostatného widgetu. V tomto případě obsahuje pole dalších widgetů, které tvoří řádek s filtry fakult. Ukázka je v rámci zjednodušení upravena a zkrácena.

```
class _FacultyFilterButtons extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<EventFilteredBloc, EventFilteredState>(
        condition: (_, state) => state is FilteredEventsLoaded,
        builder: (context, state) {
          if (state is FilteredEventsLoaded) {
            return Row(
                children: faculties
                    .map((faculty) =>
                    GestureDetector(
                      onTap: () =>
                          context
                              .bloc<EventFilteredBloc>()
                              .add(UpdateFilter(faculty)),
                      child: FilterFacultyButton(
                          faculty: faculty,
                          isActive:
                          isFilterActive(state.faculties, faculty)),
                    ))
                    .toList());
          }
        },
    );
  }
```

Po kliknutí na widget vznikne událost UpdateFilter s parametrem dané fakulty. Ta notifikuje EventFilteredBloc a informuje o tom, jaké tlačítko filtru uživatel stiskl. Následně se provede proces, popsaný v předchozí podkapitole.

5.8 Výsledný vzhled aplikace

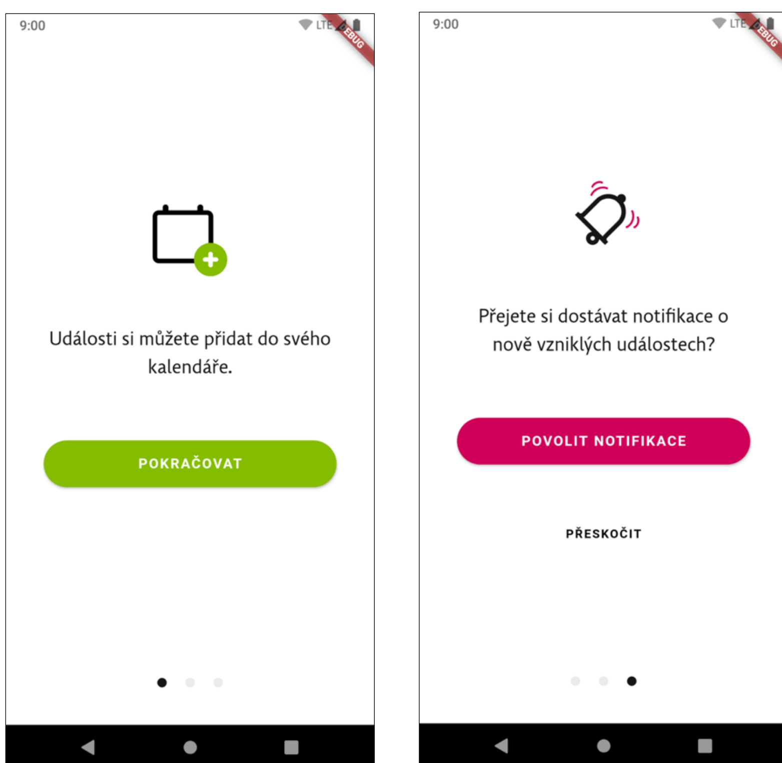
V této podkapitole jsou ukázány dokončené obrazovky v aplikaci. U každé obrazovky je také stručný popis fungování.

5.8.1 Splash screen

Splash screen je první obrazovka, která se zobrazí hned po spuštění aplikace. Je zobrazena po nezbytně nutnou dobu, než se inicializuje všechny potřebné třídy pro běh aplikace.

5.8.2 Průvodce aplikací

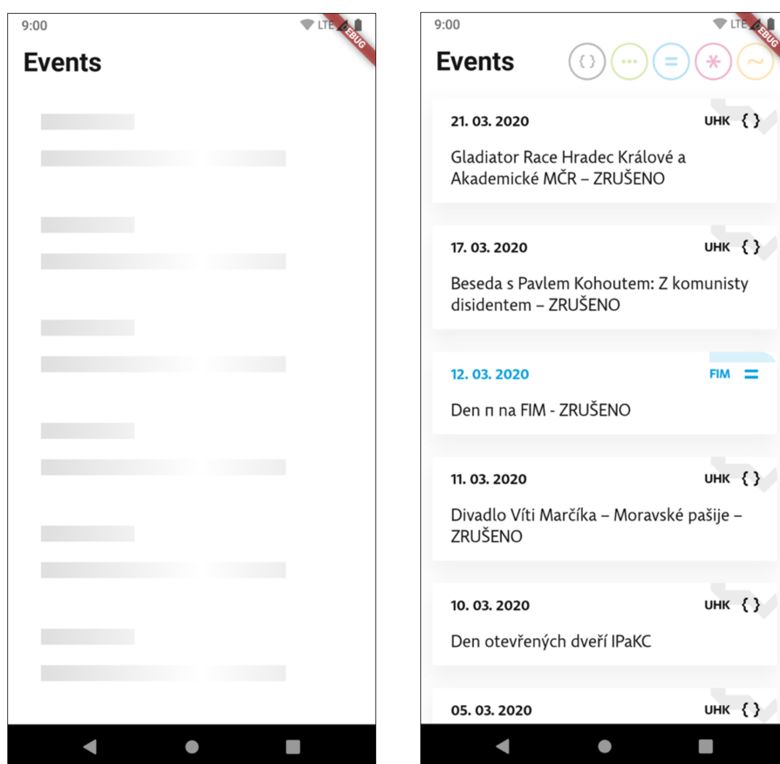
Při prvním spuštění se uživateli zobrazí krátký průvodce aplikací, který by měl seznámit uživatele s funkcemi, které mu nabízí. Na poslední stránce průvodce lze povolit zasílání push notifikací. Průvodce může být přeskočen.



Obrázek 13: Průvodce aplikací

5.8.3 Domovská obrazovka

Po dokončení průvodce je zobrazen výpis všech událostí s možností filtrace fakult. Při prvním spuštění jsou filtry vypnuté a zobrazeny jsou všechny události. Po kliknutí na políčko události se podle typu zobrazí dialog s možností přidání do kalendáře nebo detail konferenční části.



Obrázek 14: Hlavní obrazovka

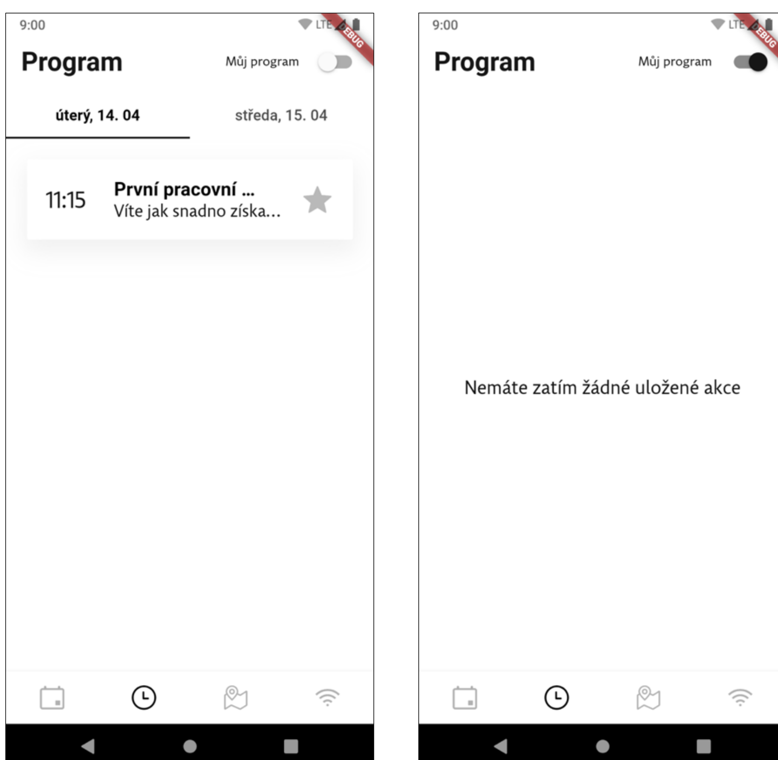
5.8.4 Konferenční událost

V konferenční části událostí se zobrazuje úvodní fotografie, název události a popis možností rozšíření (v základu se zobrazuje pouze zkrácený popis). Pokud uživatel vybral danou událost z programu jako oblíbenou, vidí zde zbývající počet minut do konání.

Obrázek 15: Konferenční událost

5.8.5 Program konferenční události

V druhé záložce konferenční části se nachází program. V konferenčním typu události může být program rozdělen do více dní, proto bylo navrženo snadné přepínání mezi těmito dny. Aby uživatel mohl filtrovat vybrané události, o které má zájem, lze mezi uloženými a neuloženými událostmi přepínat tlačítkem v horním panelu.



Obrázek 16: Program konferenční události

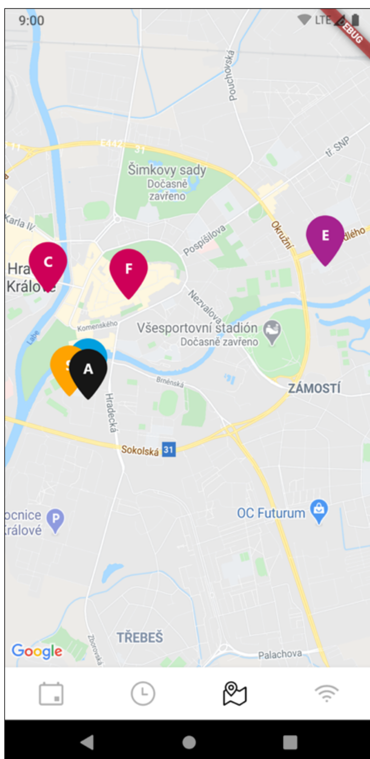
5.8.6 Detail konferenčního programu

V detailu programu může uživatel vidět fotku (pokud je k dispozici), čas konání a informace o události. Lze ji také uložit do svého programu.

5.8.7 Mapa

Na mapě se zobrazují piny označující budovy. Každý pin má barvu dle fakulty a písmeno, které označuje název budovy. V aplikaci byly využity Google Mapy pro

svoji jednoduchou implementaci na straně Flutteru. Výhodou pluginu poskytující mapy je vývoj a přímá podpora pracovníky společnosti Google. Tím je do jisté míry zajištěno, že plugin se bude v dalších verzích vylepšovat a budou opravovány chyby.



Obrázek 17: Mapa s piny budov

5.8.8 Další informace a nastavení

Poslední obrazovkou aplikace jsou další informace o události a jednoduché nastavení. Po konzultaci s PR oddělením bylo zjištěno, že účastníci akcí požadují informace o WiFi připojení. Proto při návrhu bylo rozhodnuto o zakomponování této informace do této obrazovky. Tyto údaje jsou plně konfigurovatelné ze strany Firebase Firestore. Na úplném konci obrazovky je odkaz pro kontaktování mé osoby v případě chybového reportu nebo nápadu na vylepšení. Pro identifikaci slouží i číslo verze, aby uživatel mohl uvést, v jaké verzi se chyba vyskytuje.

V nastavení lze dodatečně povolit nebo zakázat zasílání push notifikací. V dnešní době je také standardem tzv. temný režim, kdy se aplikace přepne do tmavých (nebo úplně černých) barev.

6 Testování

Testování aplikace probíhalo nejdříve v průběhu samotného vývoje aplikace. Jelikož takto nelze zjistit všechny nedostatky, bylo uskutečněno testování v uzavřené skupině pěti studentů. Těm byla aplikace nahrána na jejich zařízení bez dalšího vysvětlení o funkcionalitě. Jednalo se o tři zařízení s OS Android (ve verzích Android 7, 8 a 10) a dvě zařízení s iOS (s verzí 13). Cílem bylo získat informace o tom, zda jsou uživatelské prvky tam, kde je uživatelé čekají. Ze získané zpětné vazby vyplývá doporučení pro následující verze, a to u výpisu událostí rozlišovat, zda jde o konferenční část či nikoliv. Dále z uživatelského testování byla zjištěna nekonzistence s velikostí grafických prvků na základě odlišných poměrů stran a rozlišení obrazovek. Tento nedostatek byl eliminován pomocí dynamických velikostí jednotlivých komponent.

7 Výsledky práce

Práce navazuje na dlouhodobý požadavek PR oddělení UHK na přilákání nejen studentů univerzity, ale i široké veřejnosti, na akce pořádané na půdě univerzity. To mohou být události pořádané přímo univerzitou nebo i jinými subjekty. Cílem práce bylo vytvoření aplikace pro mobilní telefony pro operativní informování uživatelů o těchto akcích. Výsledkem práce je vytvoření návrhu uživatelského rozhraní aplikace (a jeho převedení do grafické podoby) a architektury vlastní aplikace. Hlavní a praktická část se věnuje implementaci navrhovaného řešení a popisu návrhového vzoru Bloc. V teoretické části jsou popsány použité prostředky, tj. framework Flutter a programovací jazyk Dart.

Aplikace byla naprogramována na základě stanovených požadavků na moderní architektuře. Struktura tříd byla implementována dle návrhu v kapitole 5.5. Vytvořená aplikace umožňuje uživatelům jednoduše zobrazovat data o událostech jak z webu univerzity, tak i ze služby *Firebase Firestore*. Tato data jsou získávána ve formátu JSON, který je následně v aplikaci parsován. Z důvodu Dokončeny nebyly všechny funkcionality z funkčních požadavků. K dokončení zbývá především příjem notifikací. Důvod nesplnění požadavku je hlavně ovlivněno chybějící funkcionalitou na straně serveru, který by měl o nově přidaných událostech informovat klienty pomocí zaslaného tokenu. Tato funkcionalita však pro fungování aplikace není klíčová a může být dokončena v budoucnu.

Již v prvotní fázi příprav bylo po konzultacích se zadavatelem rozhodnuto o vydání aplikace pouze v omezeném provedení. Tj. se zobrazováním průvodce a výpisu všech událostí s jednoduchým detailem dané události, a to pouze pro platformu Android. Po uživatelském testování bude následně rozhodnuto o případném pokračování a vydání stejné verze pro uživatele s operačním systémem iOS. Dalším krokem bude případná aktualizace s otevřenou konferenční částí a vydání verze pro mobilní telefony Apple.

Z důvodu nutné dohody nad vlastnictvím profilu, jeho vytvořením a zaplacením poplatků nutných k distribuci nebyla aplikace zatím publikována na Obchod Play ani na App Store.

8 Závěr a doporučení

Ačkoli je Flutter poměrně mladá technologie, dokázal se na trhu velmi rychle prosadit. Je to díky jednoduchosti a rychlosti tohoto frameworku a také díky komunitě lidí (nejen ze světa vývoje mobilních aplikací), která se kolem něj vytvořila.

Výsledná mobilní aplikace představuje relativně dobrou demonstraci funkcí užitého frameworku. Kromě představení těchto funkcí je demonstrován i využitý návrhový vzor Bloc, jež poskytuje velmi dobrou kontrolu nad tím, v jakém stavu je momentálně zobrazená obrazovka. To ve výsledku vede ke snadnější orientaci v kódu a rychlejšímu ladění chyb.

Právě testování a ladění chyb je nikdy nekončící úloha, na kterou je třeba se po dokončení vývoje zaměřit. Chybám se dá částečně zamezit tvorbou automatických testů, nelze je však zcela eliminovat. Aplikace bude v budoucnu poskytnuta uživatelům, kteří na nedostatky v aplikaci přijdou a bude nutné jejich odstranění. Tvorba automatických testů je proces, na který bych se rád v budoucnu zaměřil.

Dalším možných rozšířením projektu je podpora běhu Flutter aplikace na webu, což by umožnilo například zobrazovat události na obrazovkách, umístěných na chodbách univerzity.

Avšak nejdůležitějším a klíčovým prvkem celé aplikace je správa obsahu. To je nyní u konferenčních částí umožněno přes administrativní sekci ve Firebase Firestore. Pro pořadatele akcí by však bylo pohodlnější vyplnit formulář na webové stránce nebo přímo nahrát soubor (např. ve formátu JSON nebo CSV), který by se poté rozparsoval. Tato data by se pak nahrála do databáze a případně se jen manuálně upravila.

Pokud bylo výše napsáno, že testování a ladění chyb je nikdy nekončící proces, totéž platí i o vývoji celých aplikací a softwaru obecně. V základu je to dáno

zejména vývojem nových technologií, změnami situací, jež mobilní aplikace řeší nebo popisují a v neposlední řadě také zpětnou odezvou od uživatelů – reagováním na jejich požadavky, nároky a podněty. Proto výše uvedené možnosti rozšíření jsou v této chvíli jen základem a hlavními směry, jimiž se může využití aplikace dále ubírat. Kromě nich nepochybně z užívání vyplyne mnoho návrhů na další rozšíření, změny či odstranění chyb, jež nebyly odhaleny testováním. To může být v budoucnu podnětem pro další práci na aplikaci a předmětem vydání nových verzí.

9 Seznam použité literatury

[1] Mobile Vs. Desktop Usage (Latest Data For 2020). *BroadbandSearch.net* [online]. [cit. 2020-01-28]. Dostupné z: https://cdn.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics

[2] Mobile Operating System Market Share Worldwide. *StatCounter Global Stats* [online]. [cit 2020-01-28]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide

[3] Infographic: Apple Users More Willing to Pay for Apps. *Statista Infographics* [online]. [cit. 2020-01-28]. Dostupné z: https://www.statista.com/chart/14590/app-downloads-and-consumer-spend-by-platform/

[4] *Framework* [online]. 2019 [cit. 2020-03-10]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Framework&oldid=17557750

[5] PATRO, N. C. Choose the best — Native App vs Hybrid App. *Medium* [online]. 29. březen 2018 [cit. 2020-01-28]. Dostupné z: https://codeburst.io/native-app-or-hybrid-app-ca08e460df9

[6] Shared Library in Kotlin Multiplatform. *Karumi Blog* [online]. 2. září 2019 [cit. 2020-01-28]. Dostupné z: https://blog.karumi.com/shared-library-in-kotlin-multiplatform/

[7] *A tour of the Dart language* [online]. [cit. 2020-01-29]. Dostupné z: https://dart.dev/guides/language/language-tour

[8] OTMANOVÁ, Jaroslava. *Programovací jazyk Dart*. Hradec Králové, 2015. Bakalářská práce. Univerzita Hradec Králové.

[9] LADD, Seth a Kathy WALRATH. *What is Dart*. B.m.: O'Reilly Media; 1 edition (March 7, 2012), 2012. ISBN 978-1-4493-3316-4.

[10] VALKOVIČ, Patrik. *Factory (tovární metoda)* [online]. [cit. 2020-01-28]. Dostupné z: https://www.itnetwork.cz/factory

[11] *Why Flutter uses Dart?* [online]. [cit. 2020-01-28]. Dostupné z: https://flutter.dev/docs/resources/faq

[12] *Asynchronous programming: streams* [online]. [cit. 2020-01-31]. Dostupné

z: https://dart.dev/tutorials/language/streams
[13] *Slow rendering* [online]. [cit. 2020-01-28]. Dostupné z: https://developer.android.com/topic/performance/vitals/render
[14] *How to use packages in Flutter* [online]. [cit. 2020-01-28]. Dostupné z: https://dart.dev/guides/packages
[15] *flutter/engine* [online]. C++. B.m.: Flutter, 2020 [cit. 2020-01-29]. Dostupné z: https://github.com/flutter/engine
[16] *What is Flutter?* [online]. [cit. 2020-03-10]. Dostupné z: https://www.youtube.com/watch?v=h7HOt3Jb1Ts&list=PLOU2XLYxmsIJP13VD _Cg8qS5g2bKWTaYx&index=8&t=317s
[17] *Introduction to declarative UI* [online]. [cit. 2020-01-29]. Dostupné z: https://flutter.dev/docs/get-started/flutter-for/declarative
[18] *Technical overview* [online]. [cit. 2020-01-31]. Dostupné z: https://flutter.dev/docs/resources/technical-overview
[19] POKORNÝ, Martin. *Multiplatformní mobilní aplikace pomocí technologie Flutter*. Zlín, 2019. Diplomová práce. Univerzita Tomáše Bati ve Zlíně.
[20] *Start thinking declaratively* [online]. [cit. 2020-01-31]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative
[21] *StatefulWidget class - widgets library - Dart API* [online]. [cit. 2020-01-31]. Dostupné z: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html
[22] *InheritedWidget class - widgets library - Dart API* [online]. [cit. 2020-01-31]. Dostupné z: https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html
[23] *List of state management approaches* [online]. [cit. 2020-03-15]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/state-mgmt/options
[24] mobx | Dart Package. *Dart packages* [online]. [cit. 2020-03-15]. Dostupné z: https://pub.dev/packages/mobx
[25] *Cloud Firestore | Firebase* [online]. [cit. 2020-03-14]. Dostupné z: https://firebase.google.com/docs/firestore?hl=cs
[26] *Firebase Authentication | Simple, free multi-platform sign-in* [online]. [cit. 2020-03-15]. Dostupné z: https://firebase.google.com/products/auth?hl=cs
[27] bloc | Dart Package. *Dart packages* [online]. [cit. 2020-03-15]. Dostupné

z: https://pub.dev/packages/bloc

[28] NISHANIL. *Designing the infrastructure persistence layer* [online]. [cit. 2020-03-13]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

10 Seznam obrázků

Graf 1: Podíl stažených aplikací dle platforem v % [3] 4

Graf 2: Podíl výdajů za nákup nebo útratu v aplikaci v % [3] 5

Obrázek 1: Náhled struktury [15] 18

Obrázek 2: Simulace obrazovky aplikace (vlastní zpracování) 20

Obrázek 3: Srovnání imperativního a deklarativního stylu [17] 20

Obrázek 4: Struktura stromu widgetů [18] 21

Obrázek 5: State [20] 22

Obrázek 6: Předávání dat mezi widgety (vlastní zpracování) 25

Obrázek 7: Příklad ScopedModel, vlastní zpracování dle [19] 26

Obrázek 8: Architektura MobX, vlastní zpracování dle [24] 27

Obrázek 9: Architektura BLoC (vlastní zpracování) 28

Obrázek 10: Náhled na flutter_bloc architekturu [27] 35

Obrázek 11: Náhled na architekturu aplikace, vlastní zpracování dle [19] 37

Obrázek 12: Náhled architektury datového modulu (vlastní zpracování) 38

Obrázek 13: Průvodce aplikací 53

Obrázek 14: Hlavní obrazovka 54

Obrázek 15: Konferenční událost 55

Obrázek 16: Program konferenční události 56

Obrázek 17: Mapa s piny budov 57

11 Seznam tabulek

Tabulka 1: Ilustrace synchronní a asynchronní přístupu 13

12 Přílohy

12.1 Zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Štěpán Záliš**
Studium:	I1600638
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Multiplatformní vývoj s využitím frameworku Flutter**
Název bakalářské práce AJ:	Multi-platform Development using Flutter Framework

Cíl, metody, literatura, předpoklady:

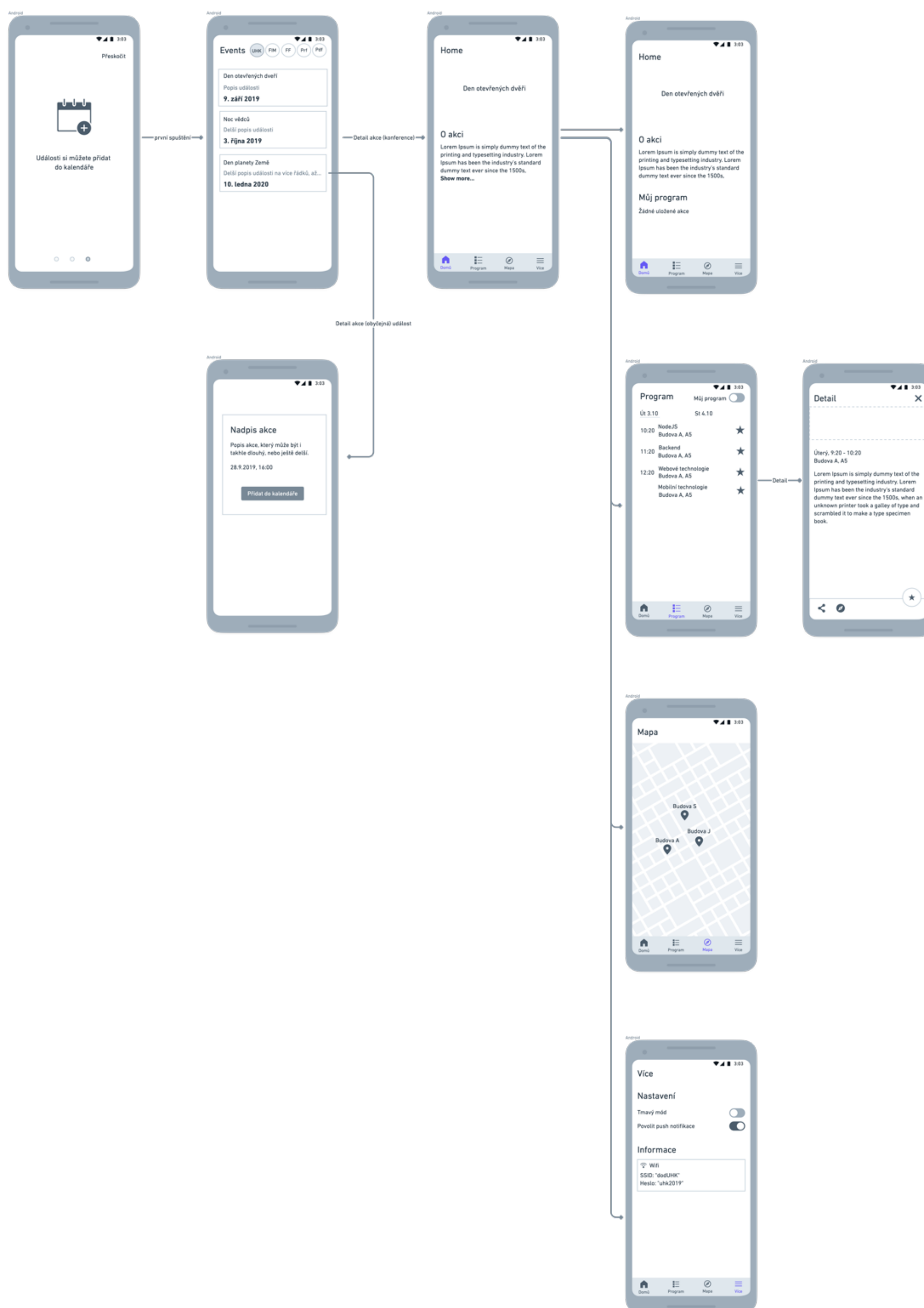
Cíl: Seznámit se s možnostmi frameworku Flutter a využít jej při tvorbě vybrané mobilní aplikace.
Osnova:

1. Úvod
2. Cíl práce
3. Popis technologií pro vývoj
4. Návrh a implementace aplikace
5. Výsledky testování
6. Závěr a doporučení
7. Seznam použité literatury

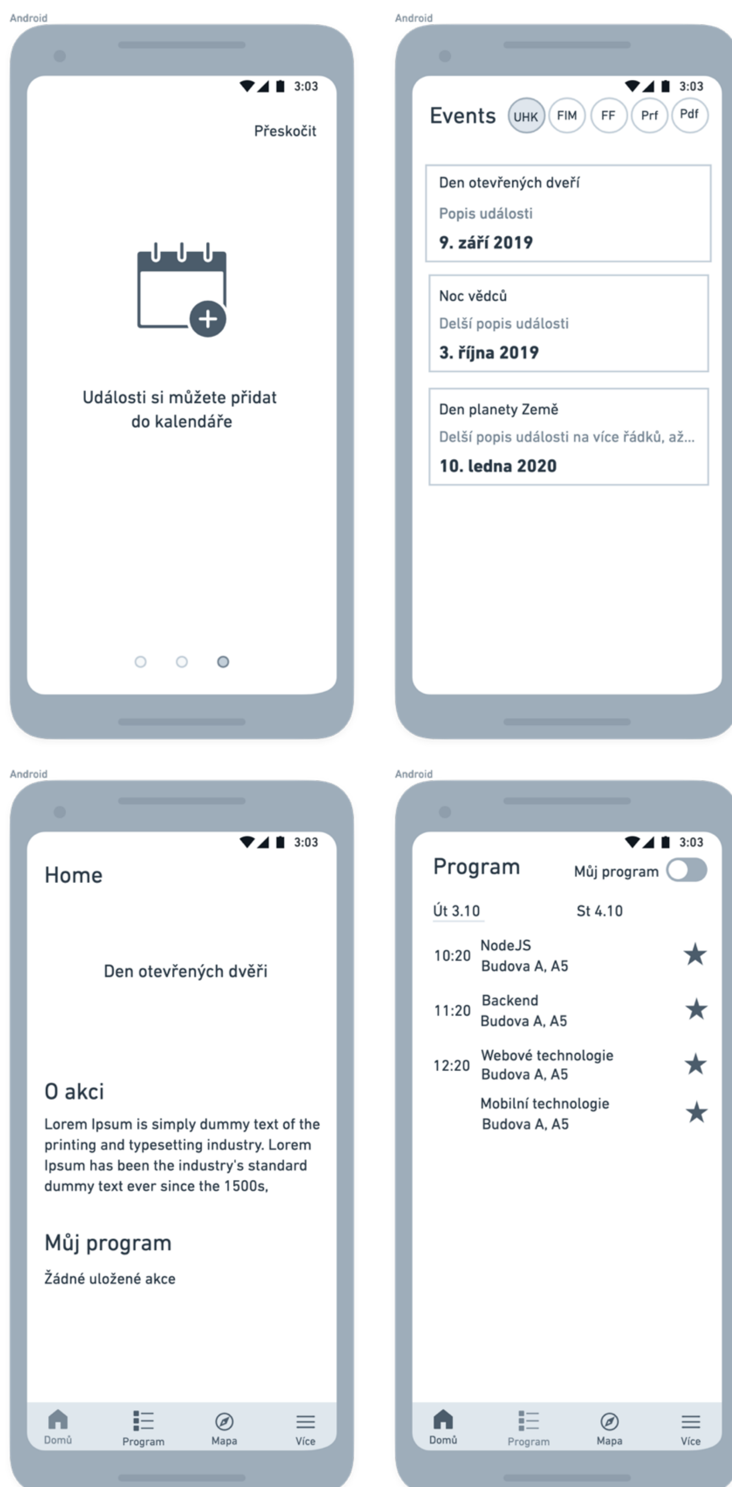
Bucket, Chriss. *Dart in Action*. Shelter Island, NY: Manning, [2013]. ISBN 9781617290862

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.11.2019

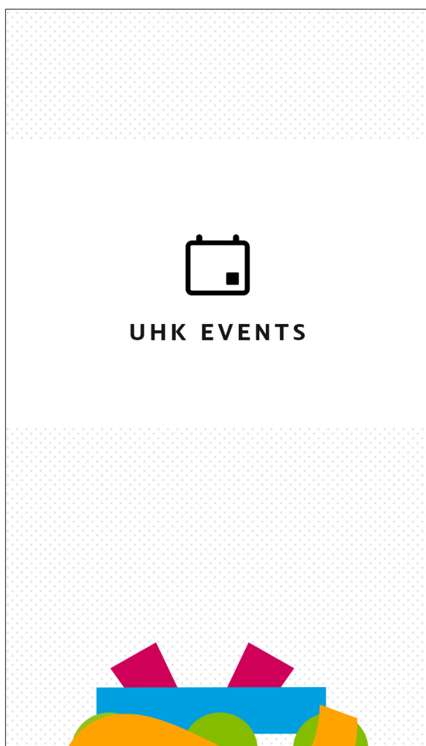
12.2 Příloha č. 2 – drátěný model aplikace

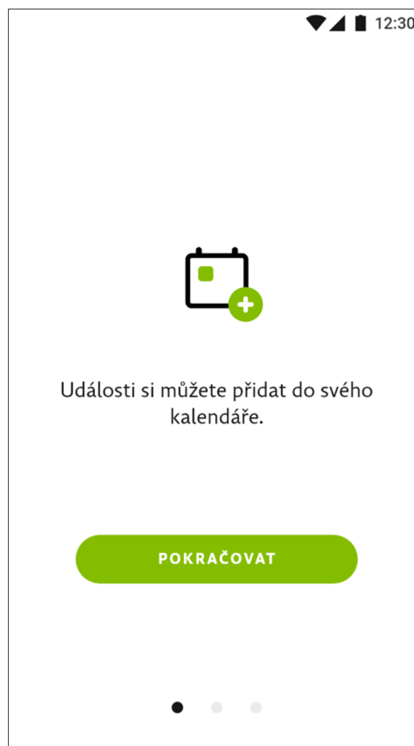


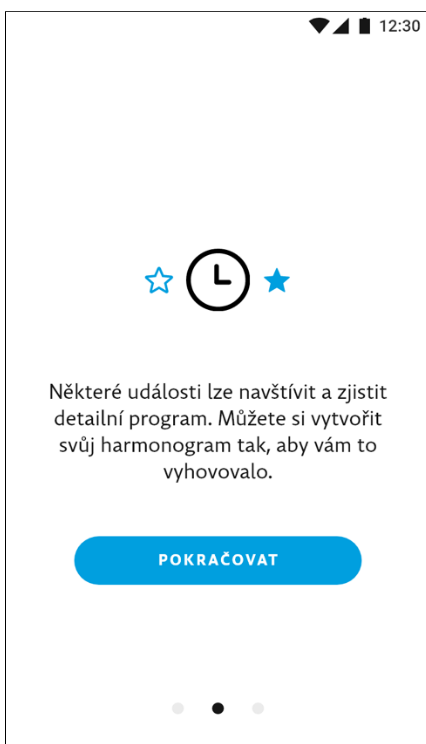
12.3 Příloha č. 3 – vybrané obrazovky drátěného modelu

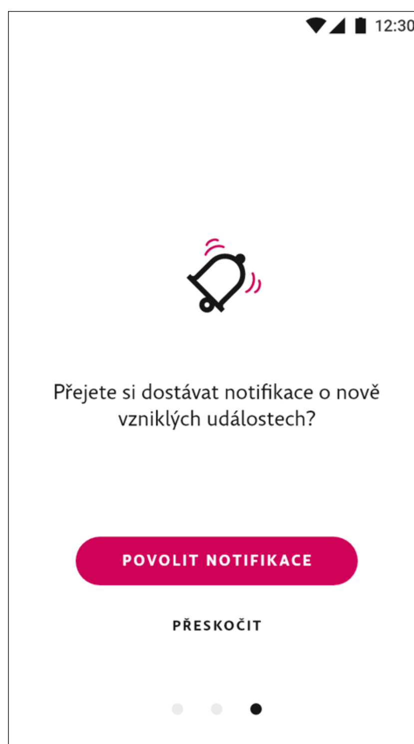


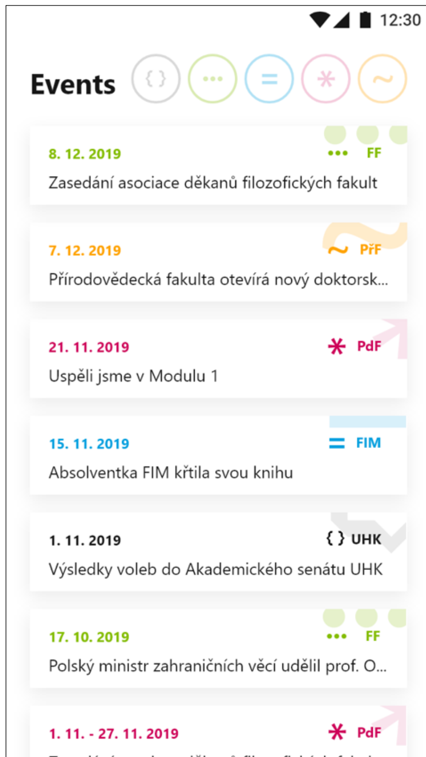
12.4 Příloha č. 4 – grafické podklady

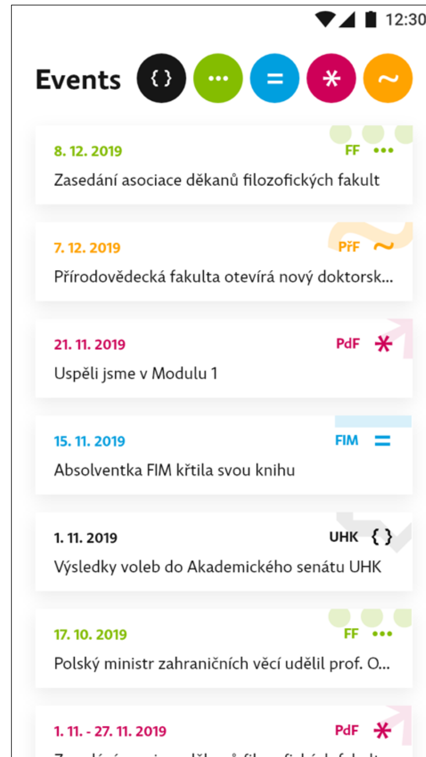


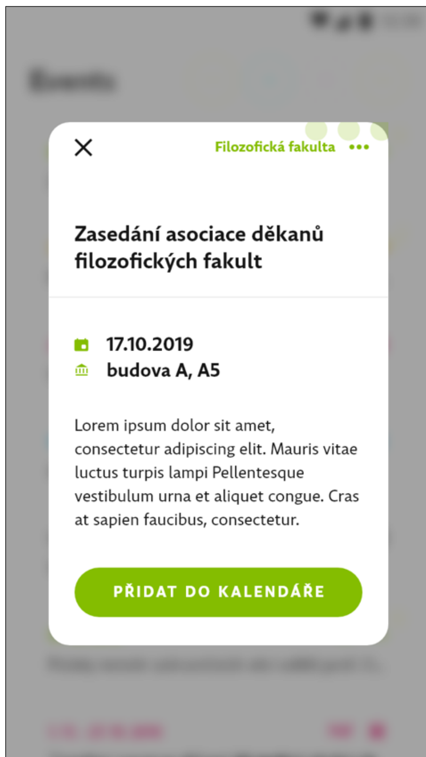


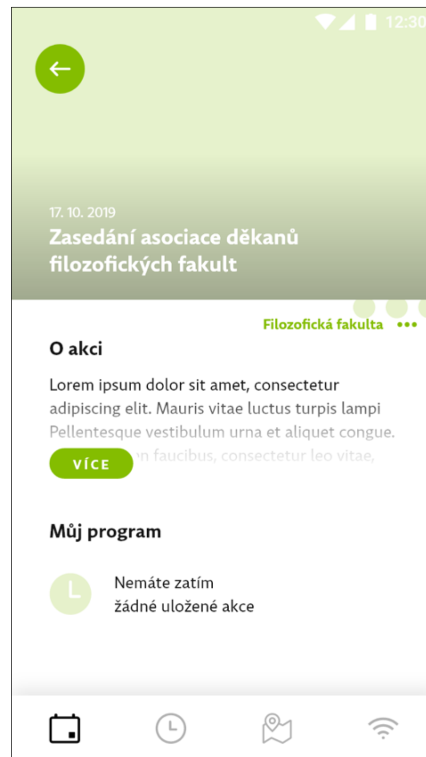


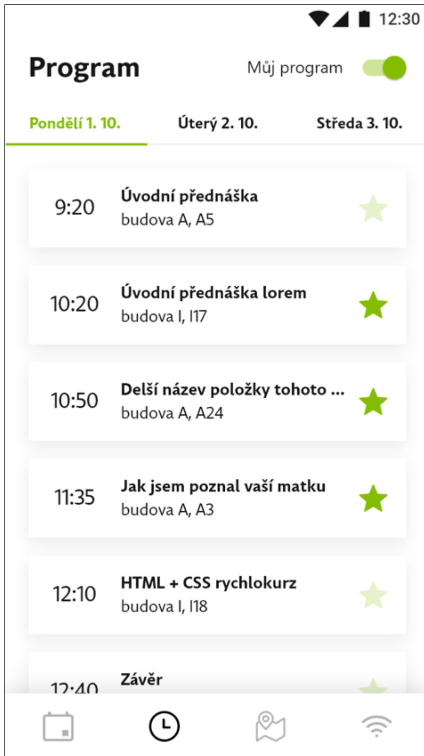


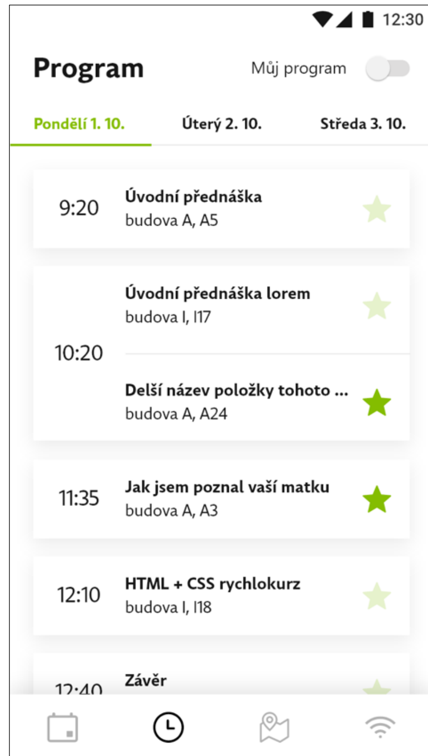

12:30
Events
8. 12. 2019
FF
Zasedání asociace děkanů filozofických fakult
7. 12. 2019
PřF
Přírodovědecká fakulta otevírá nový doktorsk...
21. 11. 2019
PdF
Uspěli jsme v Modulu 1
15. 11. 2019
FIM
Absolventka FIM křtila svou knihu
1. 11. 2019
UHK
Výsledky voleb do Akademického senátu UHK
17. 10. 2019
FF
Polský ministr zahraničních věcí udělil prof. O...
1. 11. - 27. 11. 2019
PdF

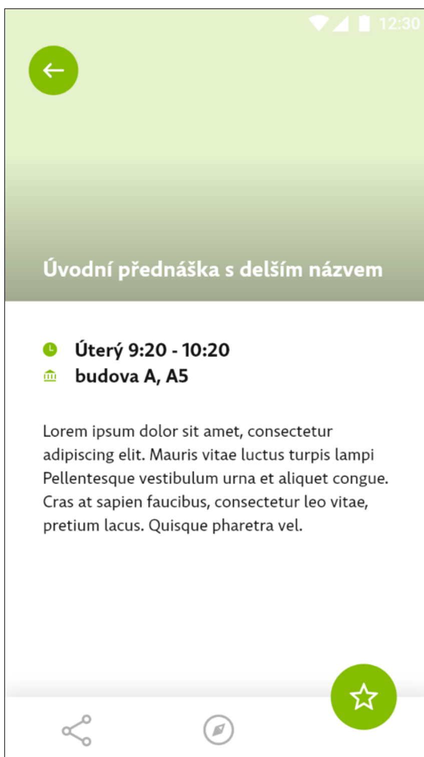

12:30
Events
8. 12. 2019
FF
Zasedání asociace děkanů filozofických fakult
7. 12. 2019
PřF
Přírodovědecká fakulta otevírá nový doktorsk...
21. 11. 2019
PdF
Uspěli jsme v Modulu 1
15. 11. 2019
FIM
Absolventka FIM křtila svou knihu
1. 11. 2019
UHK
Výsledky voleb do Akademického senátu UHK
17. 10. 2019
FF
Polský ministr zahraničních věcí udělil prof. O...
1. 11. - 27. 11. 2019
PdF

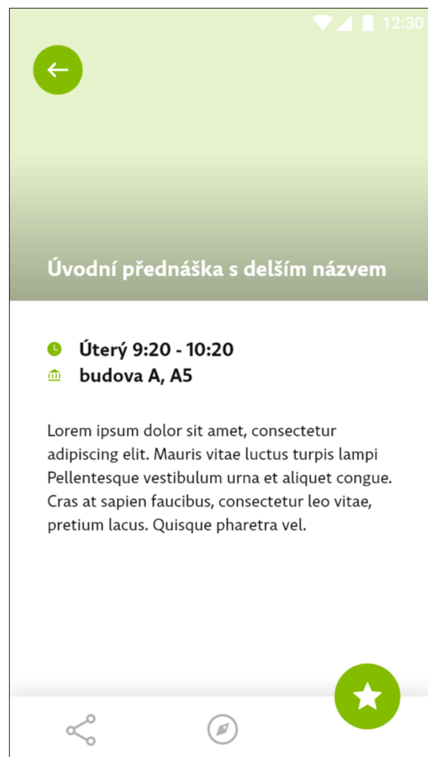

Filozofická fakulta
Zasedání asociace děkanů filozofických fakult
17.10.2019
budova A, A5
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris vitae luctus turpis lampi Pellentesque vestibulum urna et aliquet congue. Cras at sapien faucibus, consectetur.
PŘIDAT DO KALENDÁŘE

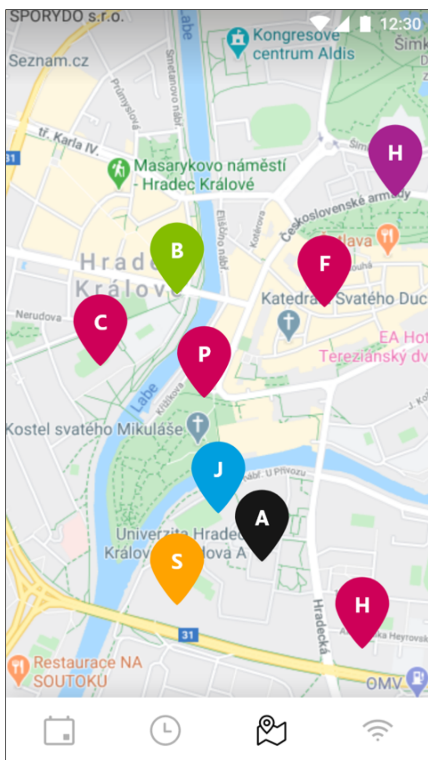
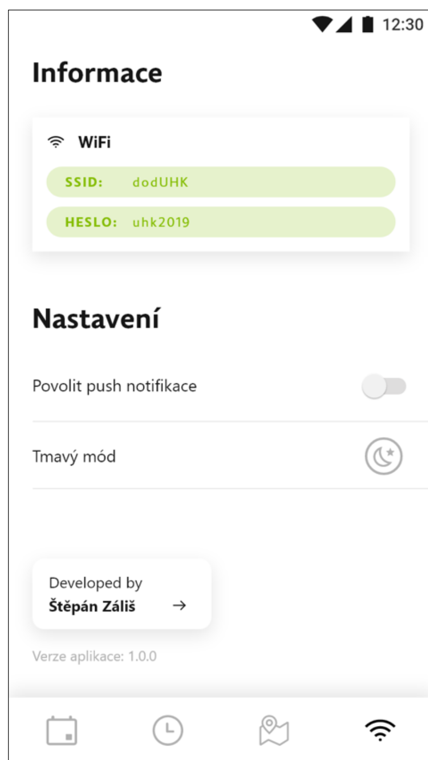

12:30
17. 10. 2019
Zasedání asociace děkanů filozofických fakult
Filozofická fakulta
O akci
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris vitae luctus turpis lampi Pellentesque vestibulum urna et aliquet congue.
VÍCE
Můj program
Nemáte zatím žádné uložené akce


12:30
Program
Můj program
Pondělí 1. 10.
Úterý 2. 10.
Středa 3. 10.
9:20
Úvodní přednáška
budova A, A5
10:20
Úvodní přednáška lorem
budova I, I17
10:50
Delší název položky tohoto ...
budova A, A24
11:35
Jak jsem poznal vaší matku
budova A, A3
12:10
HTML + CSS rychlokurz
budova I, I18
Závěr


12:30
Program
Můj program
Pondělí 1. 10.
Úterý 2. 10.
Středa 3. 10.
9:20
Úvodní přednáška
budova A, A5
Úvodní přednáška lorem
budova I, I17
10:20
Delší název položky tohoto ...
budova A, A24
11:35
Jak jsem poznal vaší matku
budova A, A3
12:10
HTML + CSS rychlokurz
budova I, I18
Závěr


12:30
Úvodní přednáška s delším názvem
Úterý 9:20 - 10:20
budova A, A5
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris vitae luctus turpis lampi Pellentesque vestibulum urna et aliquet congue. Cras at sapien faucibus, consectetur leo vitae, pretium lacus. Quisque pharetra vel.


12:30
Úvodní přednáška s delším názvem
Úterý 9:20 - 10:20
budova A, A5
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris vitae luctus turpis lampi Pellentesque vestibulum urna et aliquet congue. Cras at sapien faucibus, consectetur leo vitae, pretium lacus. Quisque pharetra vel.


SPORYDO s.r.o.
12:30
Kongresové centrum Aldis
Seznam.cz
tř. Karla IV.
Masarykovo náměstí - Hradec Králové
Hradec Králové
Katedrála Svatého Ducha
EA Hotel Tereziánský dvůr
Kostel svatého Mikuláše
Labe
Univerzita Hradec Králové
Restaurace NA SOUTOKU
OMV
Hradecká
H
B
F
C
P
J
A
S
H

12:30
Informace
WiFi
SSID: dodUHK
HESLO: uhk2019
Nastavení
Povolit push notifikace
Tmavý mód
Developed by
Štěpán Záliš
Verze aplikace: 1.0.0