ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA PEDAGOGICKÁ
KATEDRA VÝPOČETNÍ A DIDAKTICKÉ TECHNIKY

Tvorba multiplatformních mobilních aplikací pomocí Frameworku Flutter

BAKALÁŘSKÁ PRÁCE

Petr Vomáčka
Informatika se zaměřením na vzdělávání

Vedoucí práce: PhDr. Tomáš Jakeš, Ph.D.

Plzeň 2020

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně s použitím uvedené literatury a zdrojů informací.

V Plzni, 30. června 2020

.....

vlastnoruční podpis

Rád bych poděkoval panu PhDr. Tomáši Jakešovi, Ph.D. za odbornou pomoc a cenné rady v průběhu zpracování této bakalářské práce.

ZDE SE NACHÁZÍ ORIGINÁL ZADÁNÍ KVALIFIKAČNÍ PRÁCE.

OBSAH

SEZNAM ZKRATEK

1.	SDK	Software development kit (sada vývojářských nástrojů)
2.	GPU	Graphics processing unit (grafický procesor)
3.	Tzv.	Takzvaný
4.	UI	User Interface (uživatelské rozhraní)
5.	UX	User Experience (uživatelská zkušenost)
6.	HTML	HyperText Markup Language
7.	CSS	Cascading Style Sheets
8.	HTTP	Hypertext Transfer Protocol
9.	Aj.	a jiné
10.	URL	Uniform Resource Locator
11.	IDE	Integrated Development Environment
12.	GUI	Graphical User Interface (Grafické uživatelské rozhraní)
13.	VS code	Visual Studio code
14.	SQL	Structured Query Language (Strukturovaný dotazovací jazyk)
15.	KVD	Katedra výpočetní a didaktické techniky

ÚVOD

S příchodem chytrých mobilních telefonů se otevřely nové možnosti vývoje aplikací. V počátcích však tento vývoj mohl být finančně, a hlavně časově náročný, protože každý operační systém vyžadoval svou specifickou technologii. V současné době existují technologie, které nám umožňují vytvoření jediné aplikace, která bude kompatibilní s mobilními systémy iOS i Android, které jsou v dnešní době nejrozšířenější. Mezi takové technologie patří Framework Flutter od společnosti Google.

Framework Flutter umožňuje vytvářet přirozené (nativní) prvky pro oba výše zmíněné mobilní systémy. Díky této vlastnosti můžeme tvořit uživatelská rozhraní, na která jsou uživatelé daných systémů zvyklí, což vede k intuitivnímu používání aplikace.

Cílem této práce je čtenáři představit tuto novou technologii a pomocí praktických ukázek na vytvářené ukázkové aplikaci ho seznámit se základními funkcemi a vývojem ve Frameworku Flutter.

Bakalářská práce je rozdělena do dvou bloků. V první části čtenáře teoreticky seznamuji s Frameworkem Flutter tak, aby pochopil, jak tato technologie funguje a mohl porozumět praktickým ukázkám. Následně seznamuji s programovacím jazykem Dart, na kterém celý Framework běží. Podrobně se věnuji widgetům, které jsou základním stavebním prvkem Frameworku. Slouží jako programové bloky, ze kterých se skládá celá Flutter aplikace. Dále popisuji práci s balíčky a jejich využitím. Na závěr se věnuji samotné struktuře aplikace Flutter, vývojovým nástrojům a rozdílům ve vývoji pro systémy iOS a Android.

Ve druhé části práce popisuji vývoj ukázkové aplikace pro katedru výpočetní a didaktické techniky. Jedná se o systém výpůjček zařízení a uživatelů, kteří je spravují. V této části práce jsem vybral několik příkladových funkcí, kterými by finální aplikace měla disponovat. Celý proces jsem se snažil popisovat názornými ukázkami, tak aby danou problematiku bylo možno pochopit co nejlépe. Mezi ukázky například spadá vytvoření projektu Flutter, práce s dílčími soubory, deklarace grafického tématu, práce s widgety apod.

1 FRAMEWORK FLUTTER

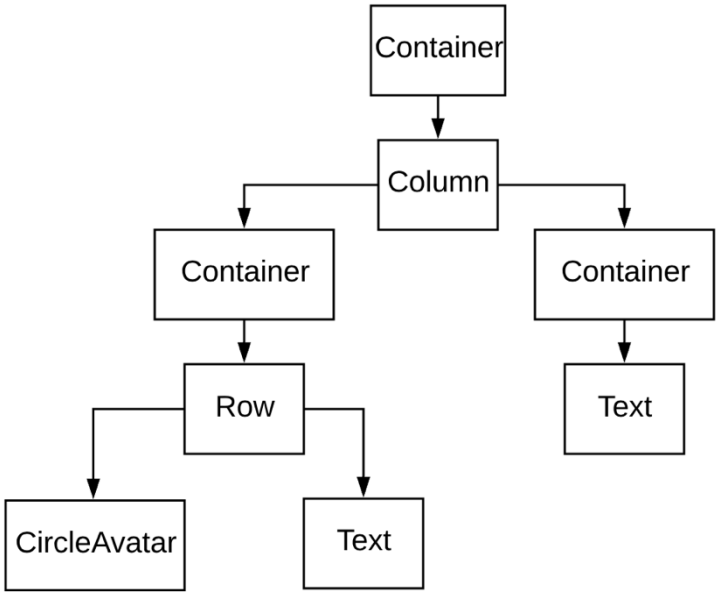
Flutter je poměrně nový[1] Framework[2] od společnosti Google, který je open-source. Skládá se ze dvou částí, z SDK a samotného Frameworku. SDK slouží uživateli jako balíček nástrojů, které mu pomáhají s vývojem, optimalizací, testováním a vytvářením multiplatformních mobilních aplikací. Samotný Framework pak využíváme při psaní kódu aplikace. Můžeme tak říci, že se jedná o programovací rozhraní, které má určitá pravidla a výchozí souborovou strukturu. Flutter je postaven na programovacím jazyce Dart a nabízí širokou škálu užitečných funkcí a vestavěných widgetů. (SCHWARZMÜLLER, 2019)

Flutter kompiluje námi napsaný kód v Dartu přímo do nativního (strojového) kódu zařízení. Myšlenka společnosti Google je tedy taková, že díky takové kompilaci kódu je velmi rychlý nejen samotný chod aplikace, ale i její vývoj. Programátor píše jeden kód a Flutter mu ho umožní vydat pro dvě různá zařízení, např. pro Android a pro iOS. Vývojář se může rozhodnout, jestli pro oba systémy zvolí jednotný design, nebo zda použije knihovny s nativními prvky, které jsou v podobě rozšíření součástí samotného Frameworku.

Celé uživatelské rozhraní je tvořeno widgety, na kterých je Flutter postaven. Framework používá vlastní vykreslovací jádro k vykreslování těchto widgetů. Pojem widget je zde z mého pohledu chápán trochu jinak, než je tomu například u vývoje webových aplikací, kde bych ho definoval jako malou část grafického prostředí, kterou uživatel využívá jako

[1] Dle dokumentace ze stránek flutter.dev je uváděna první stabilní verze, tedy verze 1.0.0, k datu **4. 12. 2018**.
[2] **Framework** je v podstatě soubor knihoven, nástrojů, funkcí, API aj., které nám pomáhají a usnadňují vývoj softwaru.

ovládací prvek stránky. Flutter však má widgety jako základní jednotky, ze kterých se tvoří tzv. Widget tree[3] (viz Obrázek 1). (NAPOLI, 2019)



Obrázek 1 - Widget tree (Zdroj: vlastní)

1.1 PŘÍSTUP K ZAČÍNAJÍCÍM VÝVOJÁŘŮM

Pro vývojáře, kteří přicházejí z různých platforem a doposud se s Flutterem nesetkali, připravil tým vývojářů ve své webové dokumentaci samostatnou sekci. Můžeme zde vidět porovnání základních syntaxí a rozdílů Flutteru oproti vývoji na React Native, iOS, Android, web, či Xamarin. Vždy je zde uveden konkrétní příklad kódu, či určité problematiky, kterou by měl vývojář dané platformy dobře znát, k tomu je uvedeno možné řešení dané problematiky pomocí Flutteru.

1.1.1 FLUTTER PRO WEBOVÉ VÝVOJÁŘE

Část *Flutter for web devs*[4] je určena pro vývojáře, kteří jsou obeznámeni s problematikou tvoření uživatelského rozhraní pomocí HTML a CSS. Představuje například vycentrování objektu pomocí webových nástrojů a následné převedení pomocí Flutteru.

[3] Jedná se o stromovou strukturu.
[4] Odkaz na dokumentaci - https://flutter.dev/docs/get-started/flutter-for/web-devs

```
//HTML
<div class="greybox">
  Lorem ipsum
</div>

//CSS
.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
```

Obrázek 2 - Centrování objektu pomocí HTML a CSS (Flutter.dev, 2020)

Na Obrázku 2 vidíme jednoduchý příklad HTML kódu, který obsahuje kontejner greybox uzavírající obyčejný text. Následují CSS vlastnosti tohoto kontejneru.

```
var container = Container( // grey box
  child:  Center(
    child:  Text(
      "Lorem ipsum",
      style: bold24Roboto,
    ),
  ),
  width: 320,
  height: 240,
  color: Colors.grey[300],
);
```

Obrázek 3 - Centrování objektu pomocí Flutteru (Flutter.dev, 2020)

Na Obrázku 3 již vidíme řešení pomocí Frameworku Flutter. Jedná se o jeden rodičovský widget *Container*, který má nastavenou šířku, výšku, barvu pozadí a větví se do svých potomků. Widget *Center* slouží k vycentrování obsahu kontejneru jak horizontálně, tak vertikálně. Dále je zde *Text*, který obsahuje prostý text s nastavenými hodnotami (tučný, 24px, Roboto).

2 DART

V této kapitole stručně představím zásadní prvky programovacího jazyka Dart. Zmíním, na kterých platformách může jazyk běžet a jakým způsobem je tak učiněno. Dále se zde budu věnovat datovým typům, které jsou v Dartu specifickým tématem a jsou tak trochu odlišné od jiných programovacích jazyků.

2.1 PŘEDSTAVENÍ

Dart je tzv. general purpose[5] programovací jazyk, který byl vytvořen společností Google v roce 2011. Jedná se o objektově orientovaný programovací jazyk, který je založen na deklarování tříd. Současná stabilní verze je 2.7.2[6].

2.2 PLATFORMY

Dart byl optimalizován tak, aby mohl běžet na několika rozdílných platformách:

- Webový prohlížeč
- Interpretovaná aplikace
- Nativní aplikace

2.2.1 WEBOVÝ PROHLÍŽEČ

Pomocí SDK, který je součástí programovacího jazyka, se kód kompiluje do JavaScriptu. Vývoj samotného balíčku byl tak efektivní, že tento transponovaný kód je z hlediska běhu aplikace rychlejší, než kdybychom kód psali v samotném JavaScriptu. (CLOW, 2019)

2.2.2 INTERPRETOVANÁ APLIKACE

SDK Dartu obsahuje virtuální prostředí, které umožňuje kód spustit v příkazové řádce, a to pomocí příkazu „dart“. Je zde využíváno Just In Time[7] (JIT) kompilace. Tuto funkci vývojář využije při ladění své aplikace, nebo při tzv. Hot reload, kdy můžete psát kód a zároveň tak svou aplikaci testovat, zpětná vazba je okamžitá. (CLOW, 2019)

[5] Programovací jazyk je univerzální, není tedy vymezen pro určitý druh aplikací. Mimo jiné do této skupiny patří např. C++, C, Java, JavaScript a celá řada dalších.
[6] Uvedená verze je k datu 30. 4. 2020.
[7] Jako **Just In Time kompilace** se označuje program, jehož kód je při samotném spuštění kompilován do nativního kódu zařízení, které při spuštění používáme.

2.2.3 NATIVNÍ APLIKACE

Dart může být kompilován do nativního kódu zařízení pomocí ahead of time[8] (AOF) kompilace. Uvedená vlastnost tedy podporuje rychlost chodu těchto nativních aplikací, jelikož Flutter widgety jsou psány v Dartu a mohou se tak kompilovat do strojového kódu, každý widget se tak může stát nativním prvkem zařízení. (CLOW, 2019)

2.3 SYNTAX

Syntaxe jazyka Dart je velmi podobná jazykům jako jsou například C#, C++, Java či JavaScript.

2.3.1 HLAVNÍ METODA MAIN

Každá aplikace musí být spouštěna přes funkci nejvyšší úrovně *main*, která slouží jako přístupový bod aplikace. Tato funkce nevrací žádnou hodnotu, je tedy datového typu *void*. (NAPOLI, 2019)

```
//syntax arrow funkce
void main() => runApp (MyApp());

//Běžná syntax zápisu funkce
void main() {
  print("Hello World!");
}
```

Obrázek 4 - Ukázka možností, jak deklarovat funkci main () (Zdroj: vlastní)

2.3.2 DATOVÉ TYPY

Programovací jazyky bývají rozděleny podle dvou typových systémů na staticky typované a dynamicky typované jazyky.

Staticky typované jazyky, jako jsou například C, C++, Java, Objective C, mají specificky definovaný datový typ proměnné. Kontrola datových typů proměnných je kompilátorem překládána ještě před tím, než se samotný kód spustí tedy tzv. ahead of time kompilace. Výhodou tohoto systému je, že vás může kompilátor včas upozornit na případné chyby zapříčiněné chybnými datovými typy. Software vyvíjený některým ze staticky typovaných

[8] **Ahead Of Time kompilace,** je taková kompilace, při které dochází k překladu vyšších programovacích jazyků, jako je například Dart, do strojového kódu, který je pro zařízení nativním.

jazyků bývá časově náročnější k dokončení, avšak v komplexnějších systémech může být spolehlivější a lepší volbou. (CLOW, 2019)

Dynamicky typované jazyky, například JavaScript, PHP, Python, Ruby, nemají specificky definovaný datový typ proměnné. Ke kontrole datového typu u proměnných dochází až při samotném běhu programu. Vývoj softwaru pomocí jazyků tohoto typu může být rychlejší a v méně komplexnějších případech také výhodnější. Ve složitějších případech je však tento systém náchylnější k chybám. (CLOW, 2019)

Programovací jazyk **Dart** je v tomto trochu specifický. Může využívat obou těchto typových systémů. Můžeme si tedy vybrat dle potřeby, jestli chceme využít specifické deklarace proměnné, nebo ji ponechat nespecifikovanou.

Dart nabízí běžné datové typy, které můžeme znát i z jiných programovacích jazyků. Některé se však trochu liší. Těchto šest datových typů patří do staticky typovaného systému.

1. **Number** (číslo)

   Deklarování proměnné jako číslo omezuje její hodnotu pouze na čísla. Dart umožňuje tento typ rozdělit ještě na dva podtypy *int* a *double*. Pokud naše hodnota nepotřebuje desetinná místa, například 10, 200, -14 …, použijeme deklaraci *int.* Pokud však vyžadujeme hodnotu s přesností na desetinná místa, například 30.2 nebo 150.38786, použijeme deklaraci *double*. Obě deklarace umožňují jak kladná, tak záporná čísla. Číselná hodnota může být maximální možné velikosti 64 bitů. Příklad zápisu můžeme vidět na Obrázku 5. (NAPOLI, 2019)

```
int pocet = 10;
double cena = 120.50;
```

Obrázek 5 - Deklarace datového typu Number (Zdroj: vlastní)

2. **String** (Text)

   Deklarace proměnné klíčovým slovem *String* umožňuje zadat hodnoty jako posloupnost textových znaků. K přidání jednoho řádku znaků mohou být použity jednoduché nebo dvojité uvozovky, například ‘auto‘, “auto“. K přidání znaků na více řádcích se použijí trojité uvozovky, například ‘‘‘auto‘‘‘. Řetězce lze kombinovat

pomocí znaménka + nebo lze použít tuto interpretaci „${}“, která nám umožní to samé. Příklady zápisů vidíme na Obrázku 6. (NAPOLI, 2019)

```
//klasický String
String jmeno = "Petr";

//zřetězený String
String vek = 'Je mi ${12 + 12} let';
String pozdarv = 'ahoj' + ' ' + 'světe';

//víceřádkový String
String adresa = '''
Plzeň
Koterovská
''';
```

Obrázek 6 - Různé deklarace String (Zdroj: vlastní)

3. **Boolean** (logická hodnota)

   Pomocí klíčového slova *bool* deklarujeme proměnnou, která může nabývat pouze dvou logických hodnot, a sice *true* nebo *false*. Příklad zápisu je vyobrazený na Obrázku 7. (NAPOLI, 2019)

```
bool isMatch = false;
isMatch = true;
```

Obrázek 7 - Příklad deklarace logické proměnné (Zdroj: vlastní)

4. **List**

   Deklarace proměnné typu *List*, u některých programovacích jazyků známé také jako *array* (pole), nám umožňuje uchovávat více hodnot najednou. Tento datový typ je tedy uspořádaná skupina objektů, ve které je každý objekt přístupný podle pozice indexu, či klíče. Takto zavedené indexování začíná na pozici 0 a končí na hodnotě, která odpovídá délce Listu mínus 1. List může mít pevně danou velikost nebo může být flexibilní, základní interpretace *List()* je nastavena jako flexibilní seznam.

K pevně dané velikosti stačí do závorek přidat číselnou hodnotu, která zastupuje námi zvolenou velikost např. *List(7)*. Příklady vidíme na Obrázku 8. (NAPOLI, 2019)

```
// List flexibilní
List kontakty = List();
List kontakty = [];
List kontakty = ['Honza', 'Petra', 'Pavel'];

// List pevné velikosti
List kontakty = List(25);
```

Obrázek 8 - Příklady deklarace datového typu List (Zdroj: vlastní)

5. **Map**

Proměnná datového typu *Map*, kterou můžeme znát z jiných programovacích jazyků jako *Object,* má společné vlastnosti s datovým typem *List*. Liší se však způsobem uchovaní hodnot. Každá hodnota v proměnné má unikátní klíč, pod kterým je inicializována. Klíč a hodnota samotné mohou být jakýmkoliv typem objektu, například textovým řetězcem nebo číslem. Příklad této proměnné vidíme na Obrázku 9. (NAPOLI, 2019)

```
// Příklad deklarace objektu (Map), 'klíč': 'hodnota'
Map novyObjekt = {
  'id1': 'žena',
  'id2': 'Anna',
  'id3': 'Plzeň'
  };
// Změna položky, která se nachází po klíčem "id3"
novyObjekt['id3'] = 'Tachov';
```

Obrázek 9 - Příklad deklarace proměnné typu Map (Zdroj: vlastní)

6. **Runes**

Proměnné tohoto datového typu mohou obsahovat hodnoty Unicode[9] znaků. Dart pro takové hodnoty používá speciální zápis '\u{XXXX}', kde 'XXXX' je čtyřmístná hexadecimální hodnota. Takto zadaná hodnota se po vypsání konvertuje do decimální hodnoty, kterou je třeba převést do Unicode znaku pomocí metody *String.fromCharCodes()*. Příklad vidíme na Obrázku 10. (NAPOLI, 2019)

[9] **Unicode** je celosvětový standard, ve kterém je každý znak zastoupen unikátní číselnou hodnotou mezi U+0000 a U+10FFFF. Tento kód může být o velikosti 8, 16 nebo 32 bitů. Nejnovější verze standardu obsahuje více než 120 000 znaků.

```dart
// Unicode pro emotikon anděla je u+1f607
Runes mujAndel = Runes('\u{1f607}');

print(String.fromCharCodes(mujAndel));
// Výsledek
//😇
```

Obrázek 10 - Příklad proměnné typu Runes (Zdroj: vlastní)

Dart však nabízí i dynamicky typované proměnné, například **var**, u kterých se neudává fixní datový typ. Dart si ho tak určí sám, či ho ponechá dynamický. Další taková proměnná je **final**. Tu použijeme v případě, kdy víme, že se nám hodnota po inicializaci v proměnné nezmění. Téměř stejný typ, jako je final, je proměnná typu **const**, která se však liší v práci s objekty. Proměnná typu final nutně nemusí zamezit změně objektu, const však takovou změnu nedovolí. (dart.dev, 2020)

3 WIDGETY

Základní myšlenkou Flutteru je postavit celé uživatelské rozhraní pomocí widgetů. Každý z nich má definované vlastnosti například vzhled, velikost, stav a další. Když se některá z nich změní, dojde ke změně stavu (state) u modifikované vlastnosti. Ta je zaznamenána a widget se znovu vykreslí na pozici, kde byla změna vykonána, tím se šetří výkon zařízení. Za znovu vykreslení zodpovídá metoda *build().* (flutter.dev, 2020)

3.1 METODA BUILD

Framework Flutter využívá widgety jako soubor vlastností, který je specifický pro každý prvek vykreslený na obrazovce. Když je tedy widget potřeba vykreslit, zavolá se metoda *build*, která je také widget. Tato metoda vrací objekt, který obsahuje informace o konfiguraci a potomcích, které tento widget může obsahovat. Metoda *build* obsahuje důležitý argument *context*, který obsahuje informaci o pozici, na které se widget nachází ve widget tree. (CLOW, 2019)

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        title: 'Ukázka metody build()', home: Center(child: Text('Ahoj KVD')));
  }
}
```

Obrázek 11 - Ukázka metody build (Flutter.dev, 2020)

Na Obrázku 11 vidíme příklad, kdy metoda *build* vrací objekt s informacemi o datech, které mají být vykresleny. V tomto případě tedy *MaterialApp* widget, který obsahuje titulek a text, který je vycentrovaný.

3.2 STATELESS WIDGET

Tento typ widgetu je vhodný pro statický obsah. Jak již název napovídá, stateless widget nemá objekt *State*. Důsledkem toho takový widget nelze v reálném čase modifikovat. Funkčnost si můžeme uvést na jednoduchém příkladu. Vytvoříme widget, který obsahuje

kontejner. Uvnitř je nadpis a pod ním obrázek. Ani jeden z těchto widgetů se nebude při užívaní aplikace měnit, budou tedy obsahovat statické hodnoty (viz Obrázek 12).

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
        child: Center(
            child: Column(children: <Widget>[Text("Nějaký text"),
                Image.network("Odkaz na obrázek")]));}}
```

Obrázek 12 - Syntax Stateless widgetu (Flutter.dev, 2020)

Metoda *build* (pouze UI část) takového stateless widgetu může být volána v těchto třech případech:

1. Když je widget poprvé vytvořen
2. Při změně rodičovského widgetu
3. Při změně zděděného widgetu

3.3 STATEFUL WIDGET

Tento widget je užitečný v případě potřeby vykreslení dat, která se v reálném čase mohou dynamicky měnit. Flutter při deklaraci tohoto widgetu oproti Stateless widgetu navíc vytvoří objekt *State,* ve kterém uchovává veškerá dynamická data, která lze měnit v průběhu jeho životnosti. Pokud se stav některé vlastnosti má změnit, použijeme metodu *setState()*, která má jako povinný argument anonymní funkci, do které vložíme data určená k překreslení.

3.3.1 OBJEKT STATE

State objekty si Framework Flutter vytváří sám. Samotný *state* je informace, která je při vytváření widgetu synchronně načtena a může být v průběhu jeho životnosti změněna. V případě změny widgetu je třeba objekt *state* aktualizovat, k tomu slouží metoda *setState()*, která je dostupná pro Stateful widget. Funkce *setState()* nastaví aktualizované hodnoty a překreslí UI prvky, kterých se tato změna týká.

```
class JournalEdit extends StatefulWidget {
  @override
  _JournalEditState createState() => _JournalEditState();
}

class _JournalEditState extends State<JournalEdit> {
  String note = 'Trip A';

  void _onPressed() {
    setState(() {
      note = 'Trip B';
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Icon(),
        Text('$note'),
        FlatButton(
          onPressed: _onPressed,
        ),
      ],
    );
  }
}
```

Obrázek 13 - Stateful widget příklad, kód převzat z knihy (Napoli, 2019)

Na Obrázku 13 vidíme příklad takového Stateful widgetu, ve kterém byla vytvořena třída *JournalEdit*, která vytváří *state* pro třídu *_JournalEditState.* V této třídě se po kliknutí na tlačítko změní proměnná *note*, která je umístěna v privátní funkci *_onPressed()*. Tato proměnná je součástí Flutter funkce *setState()*, která má anonymní funkci jako argument a mění *state* widgetu změnou proměnné *note*. Framework tak ví, že má znovu zavolat metodu *build()* a obsah widgetu znovu vykreslit.

3.4 PŘÍKLADY ZÁKLADNÍCH WIDGETŮ

V následující části představím widgety, které vývojáři Frameworku Flutter ve své webové dokumentaci uvádějí jako nejvíce používané. Dalších widgetů je samozřejmě značné množství. Jsou uvedeny v katalogu[10] webové dokumentace.

3.4.1 TEXT WIDGET

Tento widget v aplikaci umožní vytvořit stylizovaný text. Může obsahovat řadu dalších widgetů, které lze pomocí argumentů široce modifikovat[11]. Například použití konstruktoru *Text.rich* umožní zobrazit odstavec textu s různě stylovanými textovými spany[12] viz Obrázek 14. (api.flutter.dev, 2020)

```
const Text.rich(
  TextSpan(
    text: 'Hello', // default text style
    children: <TextSpan>[
      TextSpan(text: ' beautiful ', style: TextStyle(fontStyle: FontStyle.italic)),
      TextSpan(text: 'world', style: TextStyle(fontWeight: FontWeight.bold)),
    ],
  ),
)
```

Obrázek 14 - Ukázka Text.rich konstruktoru (Flutter.dev, 2020)

3.4.2 ROW A COLUMN WIDGETY

Row a Column widgety v uživatelském rozhraní umožňují vytvářet flexibilní rozvržení komponent ve vodorovném (Row) i vertikálním (Column) směru. Tyto vlastnosti jsou převzaty z webové technologie Flexbox[13]. (flutter.dev, 2020)

Row widget[14] zobrazí každého svého potomka v horizontálním poli. Obrázek 15 ukazuje příklad zápisu takového widgetu. Definuje widget *children*, který je potomkem widgetu *Row*, vloží do něj pole objektů, v tomto případě logo (*FlutterLogo)*, prostý text (*Text*)a

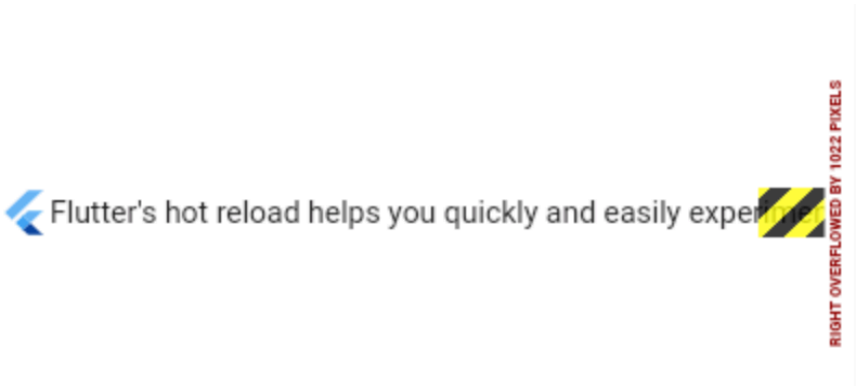
[10] **Katalog widgetů** - https://flutter.dev/docs/development/ui/widgets
[11] **Text widget** - https://api.flutter.dev/flutter/widgets/Text-class.html
[12] **Span** je využíván jako řádkový selektor pro určitý element.
[13] **Flexbox** je CSS modul, který slouží pro pohodlnější práci s rozvržením webových elementů.
[14] **Row widget** - https://api.flutter.dev/flutter/widgets/Row-class.html

emotikona (*Icon)*. Všechny tyto elementy budou zarovnány horizontálně. (api.flutter.dev, 2020)

```
Row(
  children: <Widget>[
    const FlutterLogo(),
    const Text('Flutter\'s hot reload helps you quickly and easily experiment'),
    const Icon(Icons.sentiment_very_satisfied),
  ],
)
```

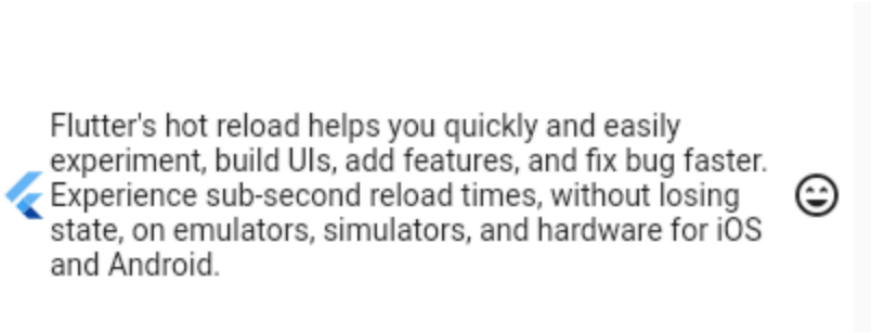
Obrázek 15 - Příklad Row widgetu (Flutter.dev, 2020)

Může se stát, že jeden z elementů bude příliš velký a došlo by tak k chybovému zobrazení obsahu viz Obrázek 16.



Obrázek 16 - Ukázka příliš velkého elementu v Row widgetu (Flutter.dev, 2020)

Pro tyto případy je zde widget *Expanded()*, do kterého zapouzdříme příliš velké potomky, například velmi dlouhý text či velký obrázek. Tento widget vypočítá místo, které je v horizontálním prostoru k dispozici a vyplní jím svůj obsah, avšak s ohledem na elementy, které jsou s ním v kontejneru. O místo se tzv. podělí. Příklad vidíme na Obrázku 17. (api.flutter.dev, 2020)



Obrázek 17 - Příklad využití Expanded widgetu (Flutter.dev.expanded, 2020)

Column widget[15] zobrazí každého svého potomka ve vertikálním poli. Tento widget dodržuje téměř stejná pravidla jako *Row*. Hlavní rozdíl je tedy ve směru řazení samotných elementů. (api.flutter.dev, 2020)

3.4.3 CONTAINER WIDGET

Tento widget slouží k modifikaci prostoru, který obklopuje každého potomka. Tomuto prostoru můžeme nastavit obsáhlou variaci vlastností, například výšku (height), šířku (width), pozici, ale i barvu (color) či transformaci, jako je zakřivení prostoru (transform) aj. Tyto vlastnosti můžeme znát z webových technologií, ve kterých se kontejnery hojně využívají při stylizaci HTML elementů pomocí CSS. (api.flutter.dev, 2020)

```
Container(
  child: Text("Less boring!"),
  color: Colors.blue,
  padding: EdgeInsets.all(20.0)
);
```

Obrázek 18 - Ukázka Container widgetu (Flutter.dev.container, 2020)

Na Obrázku 18 vidíme jednoduchý příklad *Container* widgetu, který obsahuje potomka *Text* a nastavuje barvu prostoru, který potomka obklopuje, na modrou. Poslední vlastností widgetu je *padding*, který nastavuje vnitřní odsazení od samotného textu.

[15] **Column widget** - https://api.flutter.dev/flutter/widgets/Column-class.html

4 BALÍČKY A PLUGINY

Flutter podporuje používání sdílených balíčků, kterými přispěli ostatní vývojáři do ekosystému Flutteru a Dartu. To umožňuje rychlejší vývoj aplikací, protože nemusíme vyvíjet vše od nuly.

4.1 ZÁKLADNÍ IMPLEMENTACE BALÍČKŮ

Framework Flutter, respektive Dart, podporuje systém knihoven, které se dělí na dva základní typy:

1. Core balíčky

- Tyto balíčky jsou již obsaženy v samotném Frameworku Flutter
- Pro jejich použití se nemusí nastavovat žádné závislosti třetích stran

```
import 'package:flutter/material.dart';
```

Obrázek 19 - Implementace Core balíčku (Zdroj: vlastní)

2. Externí balíčky

- Nejsou obsaženy ve Frameworku Flutter
- Balíčky jsou většinou tvořeny uživateli
- Pro použití je potřeba modifikovat soubor s názvem *pubspec.yaml*, který je obsažen v *root* adresáři naší aplikace. V tomto souboru přidáme balíček do sekce *dependencies*, kde uvedeme jeho název a verzi viz Obrázek 20. Poté v příkazovém řádku spustíme příkaz *flutter pub get* (CLOW, 2019)

```
import 'package:shared_preferences/shared_preferences.dart';
```

```
dependencies:
  flutter:
    sdk: flutter
  css_colors: ^1.0.0
```

Obrázek 20 - Implementace externího balíčku (Zdroj: vlastní)

Některé balíčky po instalaci vyžadují kompletní restart aplikace, například když obsahují kód specifické platformy jako jsou například Java, Swift, či Objective-C. Funkce *hot reload* a *hot restart* aktualizují pouze Dart kód. (flutter.dev, 2020)

4.2 HLEDÁNÍ BALÍČKŮ A JEJICH DOKUMENTACE

Balíčky lze vyhledávat na stránce https://pub.dev. U každého balíčku můžeme zjistit na jakých platformách dokáže pracovat, zda na webu, Android nebo iOS zařízeních. Po rozkliknutí se můžeme dozvědět další informace o balíčcích, např. implementaci, dokumentaci, či přehled verzí.

Balíčků zatím není mnoho, je však možné vytvořit si vlastní, který by nám, nebo i ostatním vývojářům mohl pomoci při vývoji.

4.3 PŘÍKLADY POUŽÍVANÝCH BALÍČKŮ

Uvedu zde jen některé, dle dokumentace Flutteru, nejvíce stahované balíčky.

4.3.1 URL_LAUNCHER

Tento balíček slouží ke spouštění URL na mobilních platformách Android a iOS. Jedná se o externí balíček, musíme ho tedy uvést do *pubspec.yaml* souboru. URL adresy mohou mít více formátů viz Tabulka 1.

Tabulka 1 - Ukázka čtyř možných schémat pro URL – převzato z (pub.dev, 2020)

Schéma	Akce
`http:<URL>` , `https:<URL>` , e.g. `http://flutter.dev`	Otevře URL v prohlížeči
`mailto:<email address>?subject=<subject>&body=<body>` , e.g. `mailto:smith@example.org?subject=News&body=New%20plugin`	Odešle mail na danou adresu
`tel:<phone number>` , e.g. `tel:+1 555 010 999`	Zavolá na dané číslo
`sms:<phone number>` , e.g. `sms:5550101234`	Pošle SMS na dané číslo

Tyto URL adresy se spouští přes metodu *launch()*, do které jako argument napíšeme námi zvolené schéma adresy. Takto zadaná adresa musí být datového typu *String.*

4.3.2 HTTP

Knihovna slouží pro vytváření http dotazů. Dále balíček obsahuje sadu funkcí a tříd, které usnadňují práci s http zdroji. Je nezávislý na platformách a lze jej použít jak v příkazovém řádku, tak i v prohlížeči. (pub.dev, 2020)

Knihovnu používáme přes její předdefinované funkce, které můžeme využívat po importu do našeho programu.

```
import 'package:http/http.dart' as http;

var url = 'https://example.com/whatsit/create';
var response = await http.post(url, body: {'name': 'doodle', 'color': 'blue'});
print('Response status: ${response.statusCode}');
print('Response body: ${response.body}');

print(await http.read('https://example.com/foobar.txt'));
```

Obrázek 21 - Ukázka využití http balíčku (pub.dev.http, 2020)

Na Obrázku 21 vidíme příklad práce s http balíčkem. Nejprve jej importujeme a cestu k jeho použití definujeme přes klíčové slovo *http*. Do proměnné *response* se ukládá asynchronní *post* dotaz, který má argumenty *url*[16] a *body*[17]. Poté co se dotaz vykoná, vypíše se výsledný stav[18] a tělo tohoto dotazu. Dále se například pomocí metody *http.read* přečte textový dokument z uvedené url adresy.

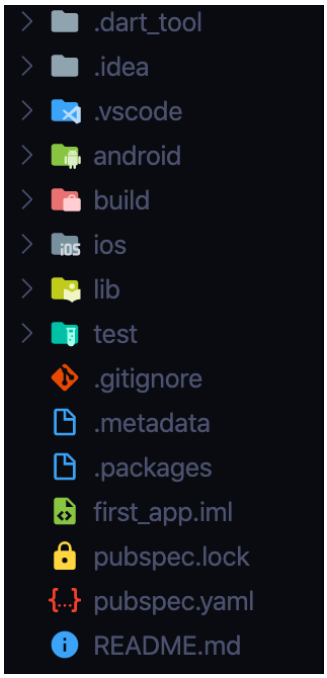
[16] Jedná se o adresu webové stránky v proměnné url.
[17] Jedná se tělo dotazu. Co tento dotaz vlastně obsahuje.
[18] http dotaz má určité status kódy, kterými udává jak dotaz, či odpověď serveru proběhla.

5 STRUKTURA FLUTTER PROJEKTU

Po vytvoření Flutter projektu se automaticky vygeneruje adresářová struktura, ve které je nutné porozumět jednotlivým složkám z důvodu efektivní práce s Frameworkem.

5.1 VÝZNAM JEDNOTLIVÝCH SLOŽEK

Na Obrázku 22 vidíme automaticky vygenerovanou souborovou strukturu, kterou si v této kapitole popíšeme.



Obrázek 22 - Souborová struktura projektu Flutter (Zdroj: vlastní)

.dart_tool

V této složce se dočasně uchovávají informace o momentálně používaných verzích balíčku, a to z důvodu potřeby zpětné kompatibility v případě, že dojde k aktualizaci právě používaného rozšíření, které třeba ještě není plně kompatibilní s ostatními verzemi systémů, či programů na nich závislých. (stacksecrets, 2019)

.idea

Tato složka obsahuje konfigurační soubory pro Android studio. (stacksecrets, 2019)

.vscode

Tato složka se vygeneruje v případě, že je používán Visual studio code editor. Jedná se opět o konfigurační složku tohoto editoru, do které je možné vkládat specifická nastavení přímo pro daný projekt. (stacksecrets, 2019)

android

Tato složka je velmi důležitá pro spuštění kódu Flutter na zařízení Android. Pomocí Flutter SDK se vytvořený kód při kompilaci spojí právě s touto složkou, která jej obohatí o funkce potřebné ke spuštění aplikace na zařízeních Android. Jedná se tedy o projekt Android, do kterého je vnořen kompilovaný Flutter kód. (stacksecrets, 2019)

build

Tato složka obsahuje výstupy Flutter aplikace, je automaticky upravována v průběhu vývoje a poslouží při konečném vydání aplikace pro systémy Android či iOS. (stacksecrets, 2019)

ios

Tato složka má stejný význam jako složka android, s tím rozdílem, že se jedná o systém iOS. Používá se, pokud chceme využít některé změny specifické k dané platformě. Uživatelům systému Windows se tato složka také vygeneruje, ale nebudou ji moci využívat. (stacksecrets, 2019)

lib

Jedná se o hlavní složku, do které se kód aplikace píše. Soubor *main.dart* slouží jako vstupní a celá aplikace se z něj bude spouštět. (stacksecrets, 2019)

test

Tato složka se využívá pro psaní automatizovaných testů, které chceme využít pro Flutter aplikaci. Využívá se tedy při fázi ladění. (stacksecrets, 2019)

.gitignore

Tento soubor slouží pro organizaci souborů, které nechceme ukládat pomocí verzovacího systému Git. Je možné zde označit celé složky nebo jen jednotlivé soubory. (stacksecrets, 2019)

.metadata

Tento soubor obsahuje metadata související s projektem Flutter. Soubor by neměl být ručně upravován. (stacksecrets, 2019)

.packages

Soubor .packages obsahuje cesty, k již nainstalovaným Flutter balíčkům, které jsou součástí samotného Frameworku a je tak generován automaticky. Opět by se neměl ručně upravovat. (stacksecrets, 2019)

first_app.iml

Tento soubor je opět specifický pro Android projekt a obsahuje informace o modulu JAVA. Je generován automaticky, takže zde není důvod k manuální modifikaci. (stacksecrets, 2019)

pubspec.lock

Tento soubor je automaticky generovaný ze souboru *pubspec.yaml*, kde se uvádí detailnější informace o každé závislosti samostatně. Soubor není třeba modifikovat. (stacksecrets, 2019)

pubspec.yaml

Do souboru *pubspec.yaml* se přidávají závislosti třetích stran, jako jsou například externí balíčky, fonty, obrázky. Můžeme zde také měnit verze aplikace. (stacksecrets, 2019)

README.md

Tento soubor slouží pro popsání projektu určitého git úložiště, uloženého například na GitHub. Uživatel, který se chce na projekt podívat nahlédne právě do tohoto souboru, kde by se měl objevit popis toho, jak aplikace funguje a k čemu slouží, případně jak celý projekt správně spustit. (stacksecrets, 2019)

6 ROZDÍLY VE VÝVOJI PRO SYSTÉMY ANDROID A IOS

Framework Flutter umožňuje výběr grafického rozhraní. Pro oba systémy můžeme použít jednotný design nebo jej odlišíme nativními widgety, které Framework poskytuje, *Cupertino* widgety určené pro iOS UI nebo *Material* widgety pro Android UI. To pro některé vývojáře může být výhodou a rádi využijí možnost vlastního a jedinečného designu. Například technologie React Native[19] má jednotné komponenty a podle zařízení si je sama modifikuje, není zde tak možnost volby. (FLUTTER, 2020)

6.1.1 ADAPTIVNÍ KONSTRUKTOR

Framework Flutter umožňuje k některým widgetům připojit[20] konstruktor *.adaptive.* Po přidání tohoto konstruktoru již nemusíme pomocí logické podmínky detekovat, na jaké platformě je kód spuštěn. Vykreslen je buď *Cupertino* widget či *Material* widget. Widgety tohoto typu se modelují samy podle typu zařízení. (FLUTTER, 2020)

Na Obrázku 23 vidíme příklad využití *.adaptive* konstruktoru na *Switch* widgetu. Na začátku

```
bool _showChart = false;

Switch.adaptive(
    value: _showChart,
    onChanged: (val) {
      setState(() {
        _showChart = val;
      });
    },
),
```

Obrázek 23 - Příklad konstruktoru *.adaptive* (Zdroj: vlastní)

si deklaruji proměnnou datového typu *boolean*, která má předdefinovanou hodnotu *false*. Samotný widget má povinné dva parametry, a sice *value* a *onChanged*. **Value** obsahuje hodnotu stavu přepínače. Druhý parametr **onChanged** obsahuje anonymní funkci, která vyžaduje parametr *val*, který stále sleduje momentální hodnotu. Tělo této funkce obsahuje Flutter metodu *setState()*[21], ve které se po kliknutí mění hodnota *_showChart* na

[19] Konkurenční Framework od firmy Facebook. Využívá se pro tvorbu multiplatformních mobilních aplikací a užitý jazyk je zde JavaScript. Pro tvorbu komponent se používá knihovna React.
[20] Zatím pouze k widgetům *Switch* a *SwitchListTile.*
[21] Po zavolání této funkce Flutter spustí metodu *build()* a tím znovu vykreslí obsah na displeji, v tomto případě přepnutí přepínače.

momentální hodnotu přepínače. Výsledný rozdíl na Android (vlevo) a iOS (vpravo) zařízení vidíme na Obrázku 24.

Obrázek 24 - Vizualizace *Switch.adaptive* widgetu na rozdílných zařízeních (Zdroj: vlastní)

6.1.2 POUŽITÍ CUPERTINO A MATERIAL WIDGETŮ

Nejjednodušší způsob, jak vkládat rozdílné widgety dle platformy, na které aplikace běží je pomocí ternárního operátoru[22] nebo jednoduché *if-else* podmínky. V některých případech je zde možnost vytvářet vlastní adaptivní třídy, které nám pomohou s celkovou přehledností kódu. Ne vždy je však lze vytvořit.

```
final PreferredSizeWidget appBar = Platform.isIOS
    ? CupertinoNavigationBar(
        middle: Text("Zkouška"),
        trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[
            GestureDetector(
              child: Icon(CupertinoIcons.add),
              onTap: () => _startAddNewTransaction(context),
            ) // GestureDetector
          ], // <Widget>[]
        ), // Row
      ) // CupertinoNavigationBar
    : AppBar(
        title: Text("Zkouška"),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.add),
            onPressed: () => _startAddNewTransaction(context),
          ), // IconButton
        ], // <Widget>[]
      ); // AppBar
```

Obrázek 25 - Vkládání rozdílných widgetů pomocí ternárního operátoru (Zdroj: vlastní)

[22] Nejčastěji se jedná o zkrácení jazykové konstrukce *if-else*. Ternární z důvodu, že má právě tři položky. Příklad by mohl vypadat takto: **podmínka ? výraz1 : výraz2;**

Na Obrázku 25 vidíme příklad widgetu, který podle zařízení v aplikaci zobrazí buď iOS *CupertinoNavigationBar*[23] nebo Android *AppBar* widget. Ternární operace začíná podmínkou *Platform.isOS*[24], která v případě, že nabývá hodnoty *true* zobrazí *CupertinoNavigationBar().* V opačném případě, kdy by podmínka nabývala hodnoty *false* se nám zobrazí widget *AppBar()*. Zjednodušený ternární zápis operace by tedy vypadal následovně: *Platform.isIOS* **?** *CupertinoNavigationBar()* **:** *AppBar();*. Vizuální rozdíly pro Android(vlevo) a iOS(vpravo) vidíme na Obrázku 26.

Obrázek 26 - Rozdílná vizualizace lišty pro Android a iOS zařízení (Zdroj: vlastní)

[23] Jedná se horní lištu, která je v aplikaci umístěna. To samé zobrazuje *AppBar*.
[24] Tato statická metoda je součástí Dart SDK. Pro použití stačí naimportovat rozšíření: *import "dart:io"*. Více k importování balíčků v kapitole 4.1. Její funkce nám jednoduše řekne, zdali je, nebo není naše aplikace spouštěna na zařízení iOS. Nabývá tedy hodnoty *true* nebo *false.*

7 VÝVOJOVÉ NÁSTROJE

Framework Flutter nabízí velkou řadu podpůrných nástrojů. Většina z nich se používá primárně pro ladění a testování. Z tohoto důvodu je zde budu popisovat právě v tomto kontextu. Pro jejich použití je třeba projít kompletní instalací Frameworku Flutter a jeho neoddělitelnými součástmi. Postup instalacemi viz Příloha 1 - Instalace potřebné k používání Frameworku Flutter. Mezi nástroje patří:

- Emulátory
- Fyzická zařízení
- Debugger
- Dart Devtools

7.1.1 EMULÁTORY

Jedná se o virtuální zařízení emulující mobilní telefon, na kterém naši aplikaci můžeme testovat, a to na několika typech mobilních telefonů bez toho, aniž bychom některý z nich fyzicky vlastnili. Tyto nástroje tak můžeme využít pro prvotní vizualizaci naší aplikace. Můžeme si např. ověřit uživatelské rozhraní, jeho soulad s grafickým návrhem či zda je aplikace funkční z logického pohledu.

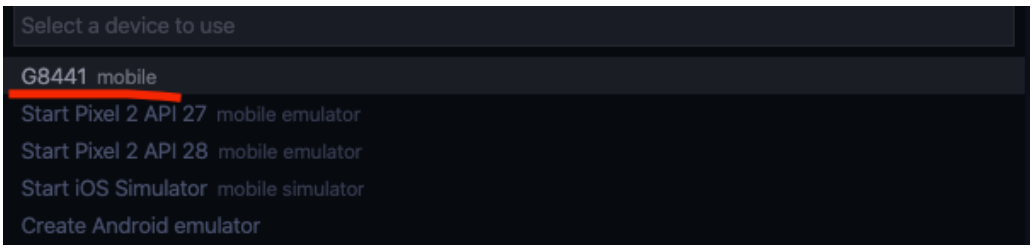
7.1.2 FYZICKÉ ZAŘÍZENÍ

Před samotným vydáním naší aplikace je velmi důležité ji vyzkoušet na co největším počtu reálných zařízení. Emulátory jsou sice velmi užitečné nástroje, ale nikdy nám nenahradí opravdová zařízení, která mohou na aplikaci reagovat jinak, než bychom očekávali.

Mobilní telefon se systémem **Android** musíme na takové testování připravit. V telefonu budeme následovat tuto cestu *Nastavení-> Systém-> Informace o telefonu*[25], pokud v této sekci sjedeme až na konec uvidíme ***číslo sestavení,*** na které musíme 7x kliknout, to nám zpřístupní sekci s názvem ***pro vývojáře***. V této sekci nalezneme přepínací položku **ladění USB** a povolíme ji.

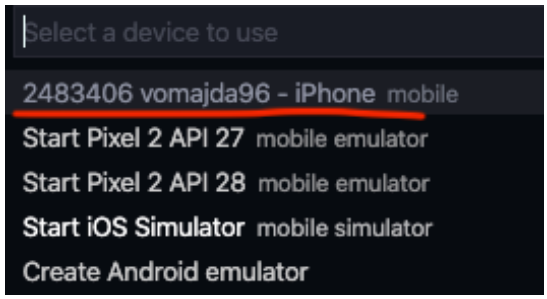
[25] Na různých verzích systému Android se tato cesta může trochu lišit.

V tuto chvíli můžeme připojit telefon k počítači a v našem editoru Visual studio code ho zvolit jako zařízení, přes které chceme aplikaci spustit (viz Obrázek 27).



Obrázek 27 - Zvolení reálného Android zařízení ve VS code. (Zdroj: vlastní)

Pro testování na reálném mobilním zařízení od firmy **Apple** je třeba doinstalovat do editoru Xcode rozšiřující balíček **cocoapods**. Další podmínkou je mít vývojářský učet[26], který je propojený s Xcode editorem. Kompletní návod, jak nainstalovat potřebná rozšíření a propojit náš účet s Xcode, je součástí dokumentace Flutter[27], sekce *Deploy to iOS devices*. Po splnění předchozích předpokladů stačí jen připojit náš iPhone k počítači. Měli bychom jej vidět v nabídce zařízení v našem VS code editoru (viz Obrázek 28).



Obrázek 28 - Zvolení reálného iPhonu ve VS code (Zdroj: vlastní)

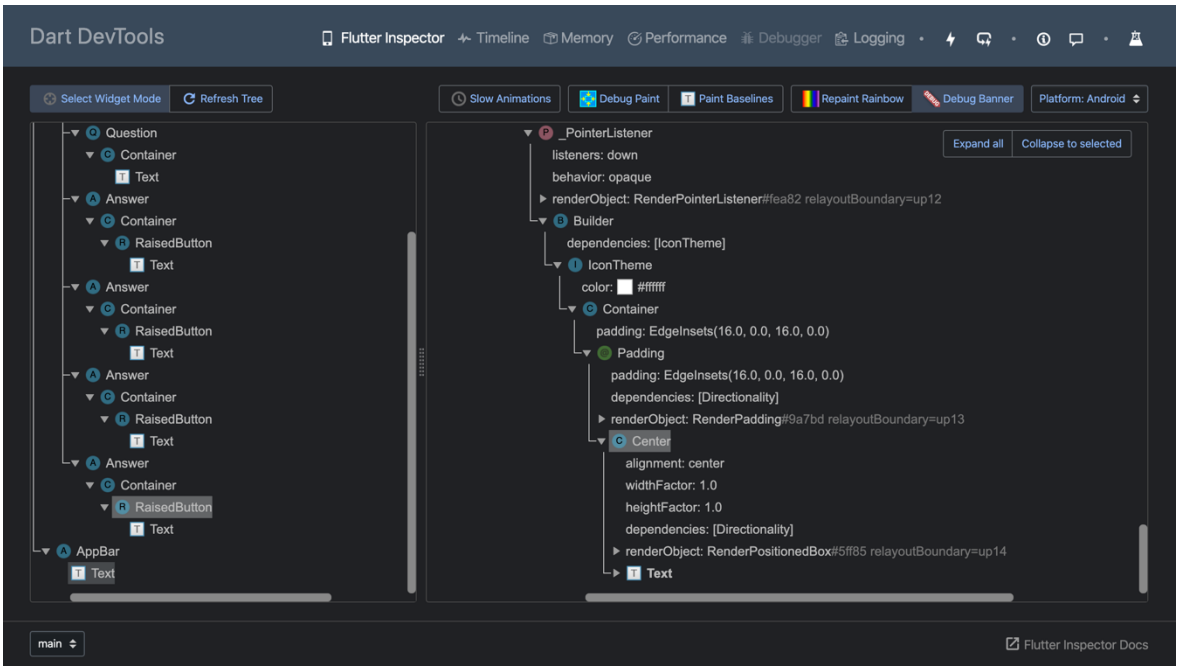
7.1.3 DEBUGGER

Visual studio code má vlastní Debugger, který nám nabízí poměrně komplexní soubor nástrojů. Používá se pro snadnější detekci chyb. Pro spuštění musíme ve VS code zvolit možnost Run-> Start Debugging. V tuto chvíli můžeme v našem kódu umístit tzv. Breakpointy, na nichž je provádění kódu přerušeno za účelem bližšího komplexního prozkoumání kódu v dané oblasti. Můžeme např. sledovat hodnoty proměnných. Tuto možnost ladění bych zvolil při opravdu rozsáhlé a komplexní aplikaci, kde nám podrobné možnosti procházení našeho kódu mohou výrazně ušetřit čas.

[26] Založení vývojářského účtu - https://developer.apple.com
[27] **Návod** na propojení vývojářského účtu s Xcode - https://flutter.dev/docs/get-started/install/macos.

7.1.4 DART DEVTOOLS

Dart DevTools jsou webovým nástrojem, který bych přirovnal k vývojářské konzoli ve webovém prohlížeči. Spustíme ho tím, že ve VS code stiskneme zkratku *cmd+shitf+p*[28], do vyhledávacího pole napíšeme *dart devtools* a vybereme nabízenou možnost *Dart: Open DevTools*. Otevře se nám nové okno prohlížeče (viz Obrázek 29), kde můžeme volit z nespočtu možností testování, například sledovat výkon naší aplikace v průběhu času, využití paměti, které části naší aplikace se při používání překreslují, komplexní podrobnosti o widgetech, či inspektora, který nám dovolí sledovat jednotlivé prvky uživatelského rozhraní a ukáže nám jejich specifikace.



Obrázek 29 - Dart DevTools (Zdroj: vlastní)

[28] Na systému Windows *ctrl+shift+p*

8 PROCES VÝVOJE UKÁZKOVÉ MOBILNÍ APLIKACE PRO KVD

V této kapitole popíši, jak jsem postupoval při vytváření multiplatformní mobilní aplikace pro katedru výpočetní a didaktické techniky. Z aplikace jsem vybral části, které jsou pro základní funkčnost aplikace stěžejní. Navrhl jsem jednoduchý design a posléze jsem ho zakomponoval do konečného řešení.

Součástí CD přílohy je vyexportovaná aplikace pro systémy Android s příponou *.apk*. Pro finální vydání aplikace je třeba vlastnit vývojářské účty na platformách Google Play pro Android aplikace a App Store pro aplikace iOS. Proces celého vydání aplikace popisuje kapitola Příloha 2 - Vydání aplikace, ve které se odkazuji na oficiální dokumentaci Flutter. Aplikaci lze také pustit na emulátorech nebo fyzickém zařízení v rámci samotného Frameworku. Pro tuto možnost je však třeba mít nainstalovaný Flutter a jeho součásti. Tuto tématiku popisuji v kapitole Příloha 1 - Instalace potřebné k používání Frameworku Flutter.

8.1 POŽADAVKY NA APLIKACI

Aplikace by měla sloužit jako evidenční a výpůjční systém pro zařízení na KVD. Měla by zpřehlednit manipulaci s jednotlivými položkami, které jsou na katedře k dispozici. Systém by měl obsahovat databázi uživatelů, kteří ho využívají. Dále pak databázi samotných položek, která je provázána s databází uživatelů. Tyto funkce jsem si vybral pro názorné ukázky, ve kterých objasním základy tvoření mobilních aplikací pomocí Frameworku Flutter.

8.2 DESIGN A PROTOTYP APLIKACE

Grafický model celé aplikace jsem rozdělil do dvou částí:

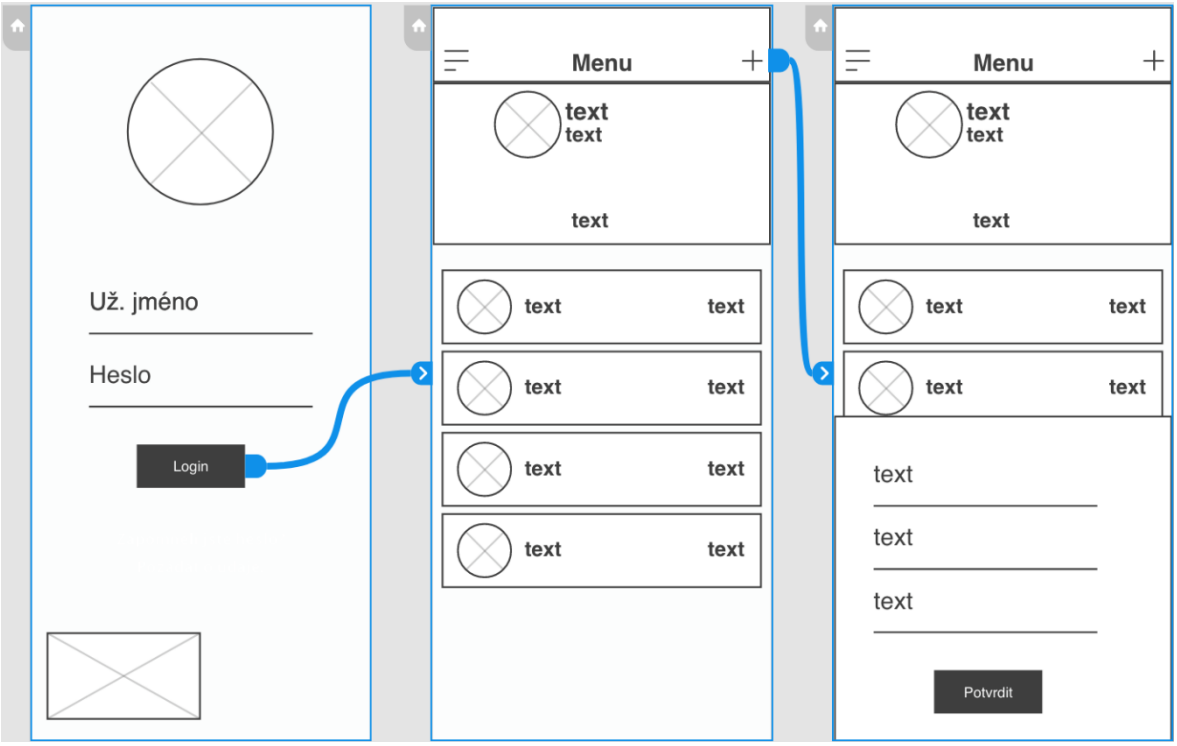
- Drátěný model
- Design uživatelského rozhraní

V drátěném modelu se budu věnovat funkčnosti ukázkové aplikace a rozvržení jednotlivých elementů aplikace. V grafickém designu se budu soustředit na vizuální stránku celé aplikace.

8.2.1 DRÁTĚNÝ MODEL

Vývoj samotné aplikace jsem začal vytvořením drátěného modelu, který mi pomohl rozložit elementy na jednotlivých obrazech aplikace. K tomu jsem použil prototypovací nástroj

Adobe XD[29]. Díky tomu jsem si mohl vyzkoušet požadované chování aplikace bez toho, aniž bych napsal jediný řádek kódu, například v podobě jednotlivých přechodů mezi obrazy nebo animací určitých bloků. Mezi další nástroje tohoto typu patří například Sketch, Figma, Affinity designer aj. Používání prototypovacích nástrojů a tvorbu drátěných modelů bych tak doporučil pro úsporu času, protože dojde k vytvoření vizualizace ještě před samotným vývojem. Příklad lze vidět na Obrázku 30.



Obrázek 30 - Ukázka drátěného modelu (Zdroj: Vlastní)

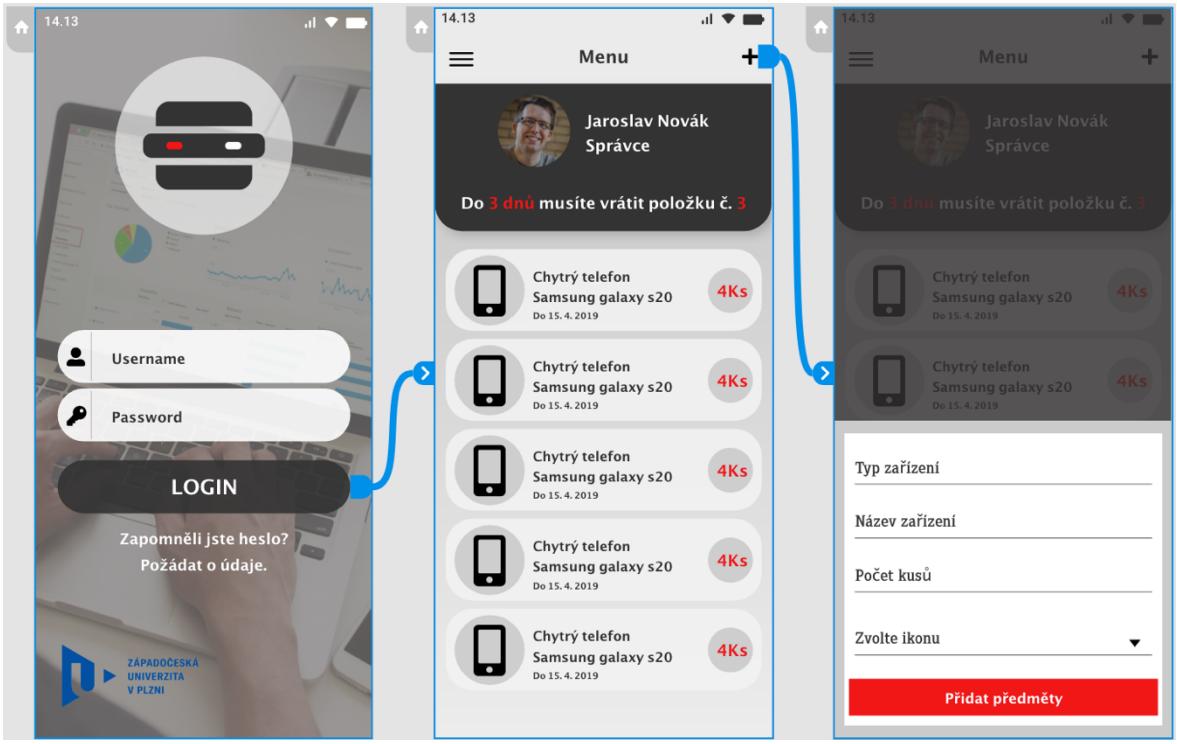
8.2.2 DESIGN UŽIVATELSKÉHO ROZHRANÍ

Po vytvoření drátěného modelu jsem se věnoval grafickému návrhu aplikace. V první řadě jsem přemýšlel nad logem, které jsem nakonec použil již z hotového projektu, který jsme tvořili v týmu na předmětu webových technologií. Pro celou aplikaci jsem zvolil tři základní barvy. Primární barvou je tmavě šedá (#333333), jako sekundární barvu jsem zvolil naopak světle šedou (#CECECE), odstínem spíše k bílé. Tu jsem zvolil tak, aby byla kontrastní a primární barva se v ní tak nemohla ztrácet, či s ní splývat. Třetí barvu jsem zvolil odstín červené (#ED1C24), tu používám jako doplňkovou, to znamená na elementy jako jsou například tlačítka, ikony, či doplňující texty.

[29] **Adobe XD -** https://www.adobe.com/products/xd.html

Uživatelské rozhraní jsem se snažil držet na jednoduché úrovni, aby zde nebyly redundantní prvky a na menších zařízeních tak nedocházelo k nepřehlednosti.

Část prototypu, který jsem v aplikaci Adobe XD vytvořil, vidíme na Obrázku 31. Přes tlačítka jsem propojil jednotlivé stránky aplikace a nasimuloval tak přechody mezi nimi. Z obrázku je patrné, že po přihlášení se dostaneme na úvodní obrazovku, na které vidíme jednoduchou uživatelskou kartu, seznam vypůjčených položek a interaktivní tlačítka v podobě dvou ikon, které se nacházejí v horní liště.



Obrázek 31 - Část prototypu aplikace v Adobe XD (Zdroj: Vlastní)

Pokud bychom chtěli na každém ze systémů rozdílný design, je dobré s tím počítat již v úvodu tvoření prototypu aplikace. V mém případě se jedná pouze o drobnosti jako jsou tlačítka, lišta aplikace, či jiné grafické provedení u vstupů. Z tohoto důvodu jsem udělal pouze jeden grafický návrh, který jsem dle potřeby modifikoval při samotném vývoji aplikace.

8.3 VYTVOŘENÍ FLUTTER PROJEKTU

Pro vývoj ukázkové aplikace jsem si musel vytvořit projekt Flutter. Vývojářský balíček Frameworku takový projekt umožňuje vygenerovat automaticky.

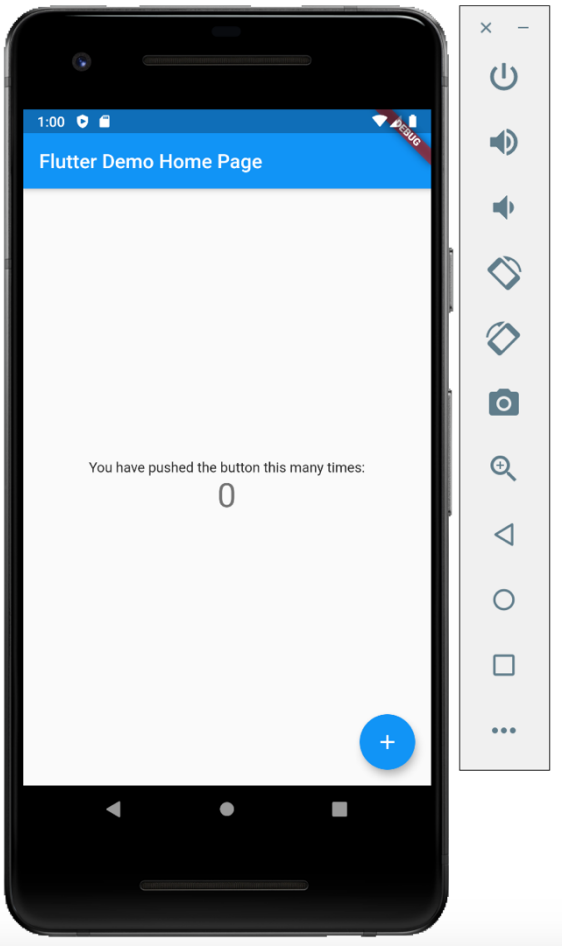
Pro vytvoření projektu je třeba spustit příkazový řádek a dostat se na úroveň adresáře, ve kterém má být projekt umístěn, v mém případě se jedná u tuto cestu:

```
/Users/vomajda96/Development/flutter_apps/
```

V takto zvoleném adresáři jsem spustil příkaz:

```
flutter create kvd_app
```

Vygenerovaný soubor vždy obsahuje ukázkovou aplikaci[30] (viz Obrázek 32), která však nijak neovlivňuje samotný chod, či práci s Frameworkem a slouží spíše k prvotnímu otestování vývojového prostředí.



Obrázek 32 - Android emulátor s ukázkovou aplikací (Zdroj: vlastní)

Při počátečním pokusu otestování ukázkové aplikace na Android emulátoru jsem narazil na problém, kdy jsem se nemohl na virtuální zařízení připojit. Jednalo se o Pixel 2 API level 29[31], kde jsem zvolil nejaktuálnější obraz systému Android s označením Q. Po průzkumu několika webových diskuzí jsem se dozvěděl, že tento problém není ojedinělý, a proto bych

[30] Uložena v kořenovém adresáři *~/lib/main.dart.*
[31] Verze Android 9.+

zde na něj rád upozornil. Ukázalo se, že se jedná o technický problém mezi Android SDK a samotným Frameworkem Flutter. V některých případech Visual studio code, a dle diskuzí ani Android studio, nedokáží virtuální zařízení s nejaktuálnějším Android systémem načíst. Řešení tedy spočívá v ponížení nejnovější verze systému Android na verzi staršího data. V mém případě jsem zvolil verzi operačního systému s označením *Pie,* kde se tedy znovu jednalo o virtuální zařízení Pixel 2, tentokrát však s API levelem 28[32]. Po opravení této chyby jsem vygenerovanou aplikaci smazal.

8.4 PŘIPOJENÍ DÍLČÍCH SOUBORŮ V PODOBĚ OBRÁZKŮ A FONTŮ

Před samotným vývojem aplikace, jsem do souboru *pubspec.yaml* načetl dílčí soubory v podobě obrázků a fontů. Soubor již obsahuje předpřipravené šablony[33] pro deklarování různých dat. Jediné, na co si je třeba dát pozor, je správný zápis syntaxe, který je u *yaml* souborů velmi důležitý. Například jedno špatné odsazení navíc a soubor nebude fungovat tak jak má. Pro dílčí soubory je v projektu třeba vytvořit nový adresář. V mém případě jsem zvolil tyto cesty a názvy:

```
~/assets/img
~/assets/fonts
```

Příklad deklarací dílčích dat v souboru *pubspec.yaml* vidíme na Obrázku 33. V levé části se jedná o zápis pro obrázky a v pravé části pro fonty. Použil jsem ještě jeden font[34], který na obrázku není uveden, jedná se však o téměř totožný zápis, který se liší pouze jménem daného fontu.

```
assets:
  - assets/img/pozadi_login.png
  - assets/img/logo.png
  - assets/img/zcu-logo.png
  - assets/img/avatar.png
```

```
fonts:
  - family: Raleway
    fonts:
      - asset: assets/fonts/Raleway-Regular.ttf
      - asset: assets/fonts/Raleway-Bold.ttf
        weight: 700
      - asset: assets/fonts/Raleway-Black.ttf
        weight: 900
```

Obrázek 33 - Příklad deklarací dílčích souborů (Zdroj: vlastní)

Takto deklarované soubory lze používat v celém projektu. Občas dochází k případům, kdy se musí celá aplikace restartovat, protože se modifikovaný soubor *pubspec.yaml* špatně načetl a je třeba ho aktualizovat.

[32] Verze Android 9.0
[33] Tyto šablony jsou okomentované, stačí je pouze zbavit komentujících tagů
[34] Jedná se o *Roboto Condensed*

8.5 DEKLARACE VLASTNÍHO GRAFICKÉHO TÉMATU

Vývoj aplikace jsem začal deklarací grafického tématu, pod kterým se definují globální grafické prvky celé aplikace. Deklaraci jsem provedl v kořenovém souboru *Main.dart*, ve kterém celou aplikaci spouštím. Tento soubor obsahuje třídu *MyApp,* ve které se definuje widget *MaterialApp*, který obsahuje důležitý argument *theme*. V tomto argumentu jsem definoval celé grafické téma, které obsahuje například tyto vlastnosti:

- Primární paletu barev
- Doplňkové barvy
- Barvy canvasu
- Fonty pro základní text
- Vlastní konfiguraci písma pro specifické případy

Pro lepší představu, co se myslí pojmem grafické téma, můžeme použít analogii z vývoje webových technologií, kde grafické rozhraní v obecném případě vytváříme pomocí CSS vlastností. Tyto vlastnosti přiřazujeme HTML elementům pomocí selektorů. V CSS si také můžeme definovat proměnné, které nám umožňují lépe využívat přidruženou vlastnost na více místech. To samé je v podstatě vytvoření grafického tématu ve Flutteru, s tím rozdílem, že zde nestylujeme každý zobrazený element, ale pouze deklarujeme obecné styly, které následně využíváme v celé aplikaci.

Názornou ukázku grafického tématu, které jsem pro aplikaci použil vidíme na Obrázku 34. Z ukázky lze vyčíst, že jednotlivé elementy aplikace budou v odstínech šedé barvy. To je definováno v argumentu *primarySwatch*. Jako doplňkovou barvu jsem zvolil odstín červené, ta je definována v argumentu *accentColor*. Pro pozadí aplikace jsem zvolil barvu světle šedou, která je odstínem blíže k bílé. To je definováno v argumentu *canvasColor*. Font pro celou aplikaci jsem zvolil *Raleway*, který jsem si připojil v předešlém kroku viz kapitola 8.4. Zde si lze také všimnout, jak jednoduše lze přidaný font použít, pouhým zadáním názvu, který jsem definoval v *pubspec.yaml* souboru. Na ukázce je také příklad deklarace vlastních textových témat. Například téma s názvem *title*. Jak již název napovídá, toto téma je určeno ke stylizaci nadpisů, které v aplikaci používám. Zde jsem nastavil velikost písma na 20px, rodinu písma *RobotoCondensed*, váhu písma na tučnou a barvu bílou.

```
return MaterialApp(
  title: 'KVD_APP',
  theme: ThemeData(
    primarySwatch: Colors.grey,
    accentColor: Colors.redAccent,
    canvasColor: Color.fromRGBO(193, 193, 193, 1),
    fontFamily: "Raleway",
    textTheme: ThemeData.light().textTheme.copyWith(
          body1: TextStyle(
            color: Color.fromRGBO(20, 51, 51, 1),
          ), // TextStyle
          body2: TextStyle(
            color: Color.fromRGBO(100, 51, 51, 1),
          ), // TextStyle
          title: TextStyle(
            fontSize: 20,
            fontFamily: "RobotoCondensed",
            fontWeight: FontWeight.bold,
            color: Colors.white,
          ), // TextStyle
        ),
  ), // ThemeData
```

Obrázek 34 - Názorná ukázka deklarace grafického tématu (Zdroj: vlastní)

8.6 DEFINOVÁNÍ NAVIGACE MEZI PLOCHAMI

V dalším kroku jsem provedl routování. Tímto pojmem se označuje definování jednotlivých cest, pod kterými jsou uloženy jednotlivé stránky, respektive plochy aplikace. Deklarace opět probíhá v kořenovém souboru *Main.dart* v metodě *MaterialApp*. Tento proces se většinou vykonává v průběhu vývoje, kdy nám vznikne potřeba definovat novou plochu. Já jsem si však tento krok dovolil uskutečnit hned na začátku, protože jsem věděl, že budu mít pouze dvě plochy. Plochu úvodní, která zobrazuje přihlašovací formulář a plochu druhou, která slouží jako hlavní plocha poté, co uživatel úspěšně přihlásí do systému.

Definovat cesty pro jednotlivé plochy lze docílit několika způsoby. V mém případě, kdy jsem pracoval pouze se dvěma obrazy, jsem zvolil velmi jednoduchý a skromný přístup (viz Obrázek 35).

```
home: auth.isAuth ? MainMenuScreen() : LoginScreen(),
routes: {
  MainMenuScreen.routeName: (ctx) => MainMenuScreen(),
},
```

Obrázek 35- Názorná ukázka definování cest k obrazům aplikace (Zdroj: vlastní)

Na ukázce si lze všimnout argumentu *home*, který označuje domovskou stránku. Pro nastavení hodnoty jsem použil ternární operátor, ve kterém jako podmínku uvádím to, jestli

je uživatel přihlášen. Pokud uživatel není na svém zařízení přihlášen, je mu místo domovské stránky zobrazen přihlašovací formulář. Pro přihlášeného uživatele je domovskou stránkou seznam jeho vypůjčených zařízení.

Argument *routes* pak označuje ostatní routy, v mém případě tedy pouze jednu stránku, kterou mám označenou jako *MainMenuScreen*. Hodnota je datového typu *Map* a v mém případě obsahuje pouze jednu hodnotu. Unikátní klíč *MainMenuScreen.routeName*, pod kterým se skrývá název cesty a jeho hodnota, která obsahuje anonymní funkci s parametrem ctx neboli context, přes který se Flutter dostává do potřebných vrstev widgetového stromu. Celá tato funkce vrací obraz v podobě uživatelské stránky s vypůjčenými položkami. Na takto definované cesty se pak můžu odkázat v celém projektu a použít je jako navigaci, například přes tlačítko, nebo ikonu.

Další možný přístup, je definovat úvodní cestu a pak ji připojit k argumentu *routes* (viz Obrázek 36). Na obrázku je ještě jeden argument, který jsem v aplikaci nepoužil, a to *onUnknownRoute*. Jedná se o cestu, která se vybere v případě, že uživatel zvolí route, který není definovaný v argumentu *routes.* K těmto případům však dochází u aplikací, které mají cesty tvořené dynamicky, což není případ mé ukázkové aplikace.

```
initialRoute: "/",
routes: {
  "/": (ctx) => LoginScreen(),
  MainMenuScreen.routeName: (ctx) => MainMenuScreen(),
},
onUnknownRoute: (settings) {
  return MaterialPageRoute(
    builder: (ctx) => LoginScreen(),
  ); // MaterialPageRoute
```

Obrázek 36 - Příklad rozdílné deklarace pro definování cest (Zdroj: vlastní)

8.7 TVORBA PŘIHLAŠOVACÍ STRÁNKY

Tuto kapitolu jsem rozdělil do dvou částí. V první části názorně popisuji grafické rozhraní a jeho tvorbu pomocí widgetů. Ve druhé se zaměřuji na logickou strukturu této stránky a na různá propojení s externími widgety. Pro tyto příklady jsem vždy zvolil ukázku kódu a jeho následnou grafickou vizualizaci v uživatelském rozhraní aplikace. Kód celé této stránky lze najít na přiloženém CD pod touto cestou:

`~/kvd_app/lib/screens/login_screen.dart`

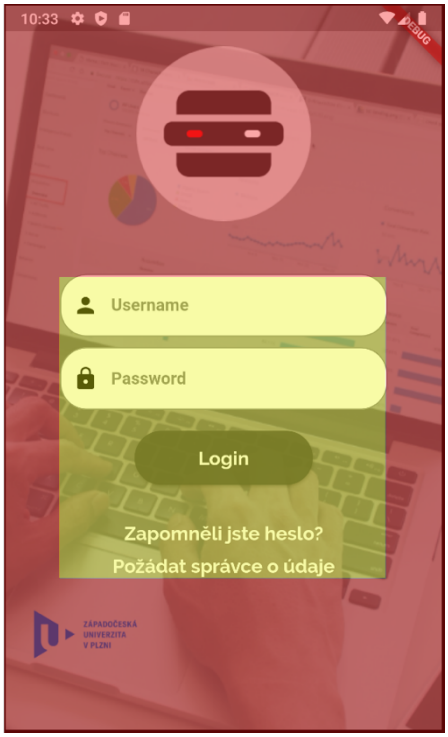
8.7.1 VÝVOJ UŽIVATELSKÉHO ROZHRANÍ POMOCÍ WIDGETŮ

Rozložení celé přihlašovací stránky jsem si rozdělil do dvou hlavních widgetů (viz Obrázek 37[35]). Statické části jsem vložil do *Stateless* widgetu s názvem *LoginScreen.* Dynamické části jsem vložil do *Stateful* widgetu s názvem *AuthCard* a následně jej vnořil do widgetu *LoginScreen*. Tuto drobnou úpravu jsem zvolil z důvodu zvýšení výkonu. Při změně dynamických polí se znovu vykreslí pouze widget *AuthCard,* nikoli *LoginScreen*, který zůstane beze změn.

```
class LoginScreen extends StatelessWidget {…

class AuthCard extends StatefulWidget {
  @override
  _AuthCardState createState() => _AuthCardState();
}

class _AuthCardState extends State<AuthCard> {…
```



Obrázek 37 - Rozdělení přihlašovací obrazovky na widgety (Zdroj: vlastní)

LoginScreen widget

Celé červené pole z Obrázku 37 je vnořeno pod rodičovským widgetem *Stack*[36]. Ten umožňuje skládat jednotlivé elementy na sebe v úrovních podle toho, v jakém pořadí byly zadány v kódu[37]. V mém případě se na nejnižší úrovni nachází pozadí v podobě obrázku, následuje logo, žluté pole v podobě formuláře a doplňujících tlačítek a na nejvyšší úrovni je logo fakulty. Příklady implementací vidíme v následujícím textu:

[35] Kód v levé části obrázku je zmenšený pro lepší formu ilustrace.
[36] Odkaz do dokumentace: https://api.flutter.dev/flutter/widgets/Stack-class.html
[37] Jedná se o datovou strukturu zásobník (Stack).

1. **Pozadí obrazovky**

```
Container(
  decoration: BoxDecoration(
    image: DecorationImage(
      image: AssetImage("assets/img/pozadi_login.png"),
      fit: BoxFit.cover,
    ), // DecorationImage
  ), // BoxDecoration
), // Container
```

Obrázek 38 - Implementace pozadí plochy (Zdroj: vlastní)

Na Obrázku 38 si lze všimnout, že přidání fotky místo pozadí není vůbec složité. Použil jsem zde widget *Container,* na který lze uplatnit vlastnost *decoration*. Pomocí této vlastnosti jsem celý kontejner vyplnil obrázkem *pozadi_login.png.* Takto vloženému obrázku jsem přidal vlastnost *BoxFit.cover*, která zapříčiní to, že obrázek překryje celou plochu a podle její velikosti se ořízne. Tuto možnost jsem zvolil pro správné zobrazovaní na různě velkých zařízeních.

2. **Logo**

```
Container(
  height: (MediaQuery.of(context).size.height -
          MediaQuery.of(context).padding.top) *
      0.25,
  width: (MediaQuery.of(context).size.height -
          MediaQuery.of(context).padding.top) *
      0.25,
  decoration: BoxDecoration(
    image: DecorationImage(
      image: AssetImage("assets/img/logo.png"),
      fit: BoxFit.fill,
    ), // DecorationImage
  ), // BoxDecoration
), // Container
```

Obrázek 39 - Implementace loga (Zdroj: vlastní)

Stejně jako u pozadí plochy jsem zde použil widget *Container*, který díky svým vlastnostem patří mezi nejpoužívanější widgety. Vlastnosti tohoto kontejneru jsou však trochu jiné. Nastavil jsem kontejneru dynamické hodnoty pro výšku a šířku. Opět z důvodu responzivity. Vzorcem, který je na obrázku uveden lze vypočítat hodnotu velikosti zařízení a následně s ní pracovat v souladu s velikostí kontejneru. V mém případě tedy bude logo vždy zabírat 25 % celkového prostoru zařízení.

Obrázek byl přidán stejným způsobem jako u pozadí s tím rozdílem, že jsem zde použil vlastnost *BoxFit.fill*, která obsah kontejneru vyplní, a to bez jakéhokoliv ořezávání samotného obrázku.

AuthCard widget

Tento widget, který je na Obrázku 37 zobrazen ve žlutém poli se skládá ze dvou vstupů, potvrzovacího tlačítka a dvou doplňujících tlačítek v podobě textu. Jelikož se jedná o přihlašovací formulář, celý balíček zmíněných elementů musí být pro správnou funkčnost uzavřen ve widgetu *Form*. Jelikož je tento widget velmi rozsáhlý a pro ukázku příliš komplexní, v následujícím textu představím pouze zjednodušené příklady jednotlivých widgetů.

1. **Vstupní pole pro hodnotu emailové adresy**

```
TextFormField(
  keyboardType: TextInputType.emailAddress,
  textInputAction: TextInputAction.next,
  onFieldSubmitted: (_) {
    FocusScope.of(context).requestFocus(_passFocusNode);
    if (value.isEmpty || !value.contains('@')) {
    _authData['email'] = value;
    errorStyle: TextStyle( // TextStyle
    color: Colors.red,
    fontWeight: FontWeight.bold,
  ), // TextStyle
), // TextFormField
```

Obrázek 40 - Vstupní pole v podobě emailu (Zdroj: vlastní)

Vstupním polím lze obecně nastavit velké množství vlastností. Na Obrázku 40 jsem uvedl jen pár opravdu základních. Například typ klávesnice, která se objeví po kliknutí do pole, kde jsem zvolil hodnotu *TextInputType.emailAddress*, která na klávesnici zobrazí potřebné znaky pro vyplnění emailové adresy. Dalším příkladem je například *TextInputType.number*, který na klávesnici zobrazí pouze číselné hodnoty. Na každý vstup lze uplatnit vlastnost *onFieldSubmitted*, která určuje, co se stane po potvrzení daného vstupu. V mém případě uživatele přepne do následujícího vstupního pole. Lze zde nastavit různé grafické prvky, například barvy textů, ohraničení, ikony, odsazení, fonty a plno dalších. Další důležitou funkcí je validátor (viz Obrázek 41), který by měl být součástí každého vstupu z důvodu ošetření vstupních hodnot.

```
validator: (value) {
  if (value.isEmpty || value.length < 5) {
    return 'Heslo je příliš krátké!';
  }
  return null;
```

Obrázek 41 - Příklad validátoru vstupních hodnot (Zdroj: vlastní)

2. **Tlačítko pro potvrzení formuláře**

```
child: RaisedButton(
  child: Text(
    "Login",
    style: TextStyle(
      fontSize: 18,
      fontWeight: FontWeight.bold,
      color: Colors.white,
    ), // TextStyle
  ), // Text
  onPressed: _submit,
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(25),
  ), // RoundedRectangleBorder
  padding: EdgeInsets.symmetric(vertical: 15, horizontal: 60),
  color: Color.fromRGBO(51, 51, 51, 0.9),
), // RaisedButton
```

Obrázek 42 - Implementace potvrzujícího tlačítka pro formulář (Zdroj: vlastní)

P ro potvrzovací tlačítko jsem zvolil widget *RaisedButton.* To oproti jiným tlačítkům již má jako výchozí vlastnost čtvercovou výplň . Tlačítku jsem nastavil vnitřní text a jeho následnou stylizaci v podobě velikosti písma, tloušťky a barvy. Dále jsem mu nastavil kruhový rádius, vnitřní odsazení a barvu celkové výplně. Nejdůležitější vlastnost každého tlačítka je však *onPressed*. K tomuto parametru se přiřazuje funkce, která se po kliknutí spustí. V mém případě jsem jako hodnotu zvolil privátní metodu *_submit*, kterou podrobněji popisuji v následující kapitole.

8.7.2 LOGICKÁ STRUKTURA A FUNKCE EXTERNÍCH WIDGETŮ

Co se týče uživatelského rozhraní, není přihlašovací stránka příliš složitá. Naproti tomu v sobě skrývá velké množství logických funkcí a větvených struktur. V této části popíši metodu pro potvrzení přihlašovacího formuláře a následný proces samotného přihlašování. Soubory k tomuto tématu se nacházejí na CD příloze pod touto cestou:

```
~/kvd_app/lib/screens/login_screen.dart
~/kvd_app/lib/providers/auth.dart
```

1. **Metoda pro potvrzení formuláře**

```dart
Future<void> _submit() async {
  if (!_formKey.currentState.validate()) {
    // Invalid!
    return;
  }
  _formKey.currentState.save();
  setState(() {
    _isLoading = true;
  });
  try {
    await Provider.of<Auth>(context, listen: false)
        .login(_authData["email"], _authData["password"]);
  } on HttpException catch (e) {
    var errorMessage = "Přihlášení se nezdařilo.";
    if (e.toString().contains("EMAIL_EXISTS")) {
      errorMessage = "Tuto adresu již někdo používá.";
    } else if (e.toString().contains("INVALID_EMAIL")) {
      errorMessage = "Mail byl zadán v chybném formátu.";
    } else if (e.toString().contains("WEAK_PASSWORD")) {
      errorMessage = "Toto heslo není dostatečně silné.";
    } else if (e.toString().contains("EMAIL_NOT_FOUND")) {
      errorMessage = "Chybný email.";
    } else if (e.toString().contains("INVALID_PASSWORD")) {
      errorMessage = "Chybné heslo.";
    }
    _showErrorDialog(errorMessage);
  } catch (e) {
    const errorMessage =
        "Momentálně se nelze přihlásit. Zkuste to znovu později.";
    _showErrorDialog(errorMessage);
  }

  setState(() {
    _isLoading = false;
  });
}
```

Obrázek 43 - Metoda pro potvrzení formuláře (Zdroj: vlastní)

Metoda *_submit* je datového typu *Future<void>,* která v konečném čase vrátí prázdnou hodnotu *Future*. Celá metoda je asynchronní, protože pracuje s daty ze serveru a není žádoucí, aby blokovala činnost celého programu.

V první části metody je podmínka, která kontroluje, zdali je formulář validní, pokud není, funkce se zde ukončí. V opačném případě proces pokračuje s uložením momentálního stavu formuláře a nastavením proměnné *_isLoading* na hodnotu *true*. Tato hodnota v programu způsobí, že se spustí animace načítání v podobě rotujícího kolečka.

Výkonná část je v *try-catch* bloku. V této části se funkce pokusí o zavolání externí metody *login* a zkontroluje, zdali server nevrátil nějakou z uvedených chyb. Pokud vše proběhlo v pořádku proměnné *_isLoading* se nastaví hodnota *false*, což indikuje, že hodnota, na kterou se čekalo, je již přístupná a animace může zmizet. V tuto chvíli je uživatel přihlášen a přesměrován na úvodní stránku.

2. **Metoda pro přihlášení uživatele**

```
Future<void> login(String email, String password) async {
  final url =
      "https://identitytoolkit.googleapis.com";
  try {
    final response = await http.post(
      url,
      body: json.encode({
        "email": email,
        "password": password,
        "returnSecureToken": true,
      }),
    );
    final responseData = json.decode(response.body);
    if (responseData["error"] != null) {
      throw HttpException(responseData["error"]["message"]);
    }
    _token = responseData["idToken"];
    _userId = responseData["localId"];
    _expiryDate = DateTime.now().add(
      Duration(
        seconds: int.parse(responseData["expiresIn"]),
      ),
    );
    notifyListeners();
    _autoLogout();
  } catch (e) {
    throw e;
  }
}
```

Obrázek 44 - Metoda *login* pro komunikaci se serverem (Zdroj: vlastní)

Jako serverovou službu jsem zvolil Firebase[38] od firmy Google. Firebase jsem zvolil proto, abych nemusel vytvářet své vlastní serverové řešení. Tato služba mimo jiné funguje jako REST API. Využil jsem ji v projektu pro ukládání dat do databáze a jejich následnou správu.

[38] Odkaz na stránky Firebase: https://firebase.google.com/

Metoda *login* má dva povinné parametry *email* a *password*. Tyto parametry zastupují hodnoty, které uživatel zadal do vstupů ve formuláři. Celá metoda je opět asynchronní, protože pracuje s daty ze serveru. V proměnné *url* je uložená adresa na Google API, která je potřeba pro komunikaci se serverem. Celá funkčnost je v bloku *try-catch*. V bloku *try* metoda posílá http post dotaz, který obsahuje adresu serveru a tělo dotazu. Tělo má hodnotu v podobě *.json* souboru, který obsahuje email, heslo a logickou hodnotu, kterou serveru říká, aby v odpovědi vrátil bezpečností token. Odpověď serveru je uložena do proměnné *responseData*. Jednotlivé informace v podobě tokenu, uživatelského ID a expirační doby jsou dočasně uloženy v paměti telefonu. Celý blok je nastaven na základní detekci chybových zpráv.

3. **Metoda pro ověření přihlášeného uživatele**

```
bool get isAuth {
  return token != null;
}
```

Obrázek 45 - Metoda pro ověření přihlášení (Zdroj: vlastní)

Tato metoda je opravdu jednoduchá. Jedná se o tzv. getter, který vrací logickou hodnotu *true* nebo *false.* V případě, že je token prázdný vrátí hodnotu *false*, v opačném případě *true*. Tuto metodu jsem využil pro zvolení úvodní obrazovky aplikace (viz Kapitola 8.6).

4. **Metoda pro odhlášení**

```
void logout(){
  _token = null;
  _userId = null;
  _expiryDate = null;
  notifyListeners();
}
```

Obrázek 46 - Metoda pro odhlášení uživatele (Zdroj: vlastní)

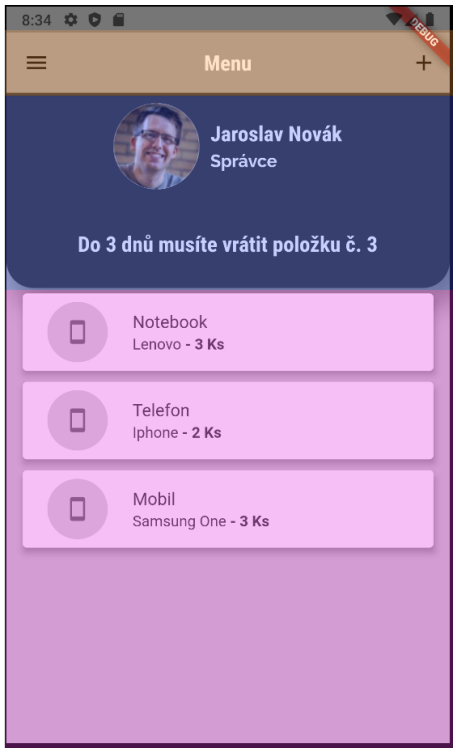
Metoda pro odhlášení vymaže všechny proměnné, které nastavuje metoda pro přihlášení. V tuto chvíli již neplatí getter *isAuth*, uživatel je odhlášen a přesměrován na přihlašovací stránku aplikace.

8.8 TVORBA ÚVODNÍ STRÁNKY

Tato stránka je rozdělena do tří hlavních sekcí. První z nich je lišta, která obsahuje dvě akční tlačítka v podobě ikon. Tlačítko s ikonou tzv. hamburgeru slouží pro vysunutí navigační lišty z levé části obrazovky. Druhá ikona vysune ze spodní části formulář pro přidávání položek. Druhou částí je uživatelská karta, která zobrazuje obecné informace o momentálně přihlášeném uživateli. V poslední části se nachází dynamický posuvný seznam, který načítá vypůjčené položky se základními informacemi v podobě úzkých bloků. Každý z těchto bloků disponuje funkcí smazání po přejetí prstem po jeho obsahu. Vizuální ukázku této stránky vidíme na Obrázku 47[39]. Kód celé této stránky je k nalezení na přiloženém CD pod touto cestou:

`~/kvd_app/lib/screens/mainMenu_screen.dart`

```
return Scaffold(
  appBar: appBar,
  drawer: MainDrawer(),
  body: SafeArea(
    child: SingleChildScrollView(
      child: Column(
        children: <Widget>[
          UserCard(appBar, mediaQuery),
          Container(
            height: (mediaQuery.size.height -
                appBar.preferredSize.height -
                mediaQuery.padding.top) *
              0.7,
            child: FutureBuilder( // FutureBuilder
```



Obrázek 47 - Rozdělení úvodní stránky na widgety (Zdroj: vlastní)

8.8.1 VYTVOŘENÍ NAVIGAČNÍ LIŠTY A JEJÍCH SOUČÁSTÍ

Tento element je vyobrazen v oranžovém bloku na Obrázku 47. Kromě své statické podoby obsahuje i dvě akční tlačítka v podobě ikon. První z nich zastupuje funkci vysouvací lišty, která se po zmáčknutí vysune z levé části obrazovky. Druhé tlačítko zobrazí formulář pro přidávání položek. V této části popíši tvoření widgetu *AppBar* a jeho základní vlastnosti,

[39] Kód v levé části obrázku je zmenšený pro lepší formu ilustrace.

dále pak *drawer* widget a jeho implementaci a také samotný formulář pro zadávání položek a jeho komunikaci se serverem. Kompletní kód k této kapitole se nachází na přiloženém CD pod touto cestou:

```
~/kvd_app/lib/widgets/main_drawer.dart,
~/kvd_app/lib/widgets/new_item.dart,
~/kvd_app/lib/providers/items.dart,
```

1. **AppBar widget**

```dart
PreferredSizeWidget _buildAppBar(BuildContext context) {
  return AppBar(
    title: Text("Menu", style: Theme.of(context).textTheme.headline6),
    centerTitle: true,
    actions: <Widget>[
      IconButton(
        onPressed: () => _startAddNewItem(context),
        icon: Icon(Icons.add),
      ), // IconButton
    ], // <Widget>[]
  ); // AppBar
}
```

Obrázek 48 - Widget AppBar a jeho základní vlastnosti (Zdroj: vlastní)

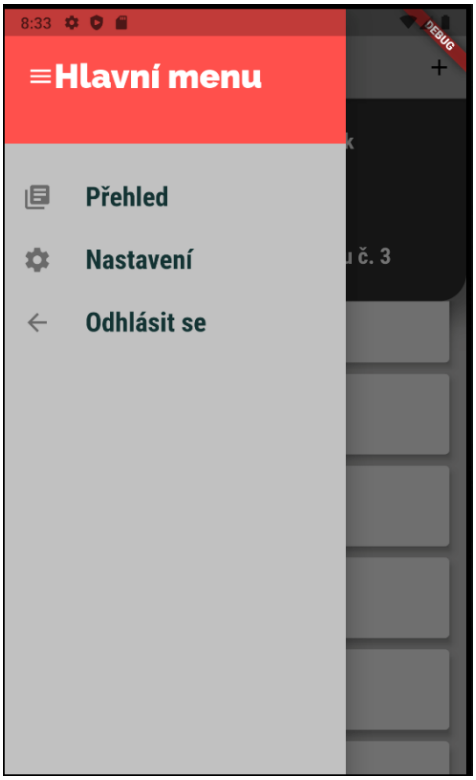
Celý *AppBar* widget jsem převedl do samostatné privátní metody *_buildAppBar*. Metodám tohoto typu se říká *builder methods*, a to z toho důvodu, že se pomocí nich vytváří nové widgety. Tato separace pomáhá v čitelnosti a přehlednosti celého kódu. Metoda vrací widget, který se může skládat z vlastností, které jsou uvedeny na Obrázek 48. Například z titulu, ve kterém je textová hodnota *Menu* a má nastavený globální styl *headline6*. Titul je pomocí logické hodnoty vycentrovaný. Dále pak tento widget obsahuje vlastnost *actions,* která nabývá hodnoty datového typu pole. V tomto případě obsahuje pouze jedno akční tlačítko *IconButton*, které má nastavené dvě vlastnosti. *OnPressed* vlastnost je nastavena pomocí anonymní funkce vracející privátní metodu, která se vykoná po stisknutí tohoto tlačítka. Druhá vlastnost *icon* slouží pro nastavení ikonky, která se jako tlačítko zobrazí. Hodnota *Icons.add* zobrazí ikonu znaménka plus.

2. **Drawer widget**

```
@override
Widget build(BuildContext context) {
  return Drawer(
    child: Column(
      children: <Widget>[
        Container(
          height: 120,
          width: double.infinity,
          padding: EdgeInsets.all(20),
          alignment: Alignment.centerLeft,
          color: Theme.of(context).accentColor,
          child: Row(
            children: <Widget>[
              Icon( // Icon
              Text(
                "Hlavní menu",
                style: TextStyle( // TextStyle
              ), // Text
            ], // <Widget>[]
          ), // Row
        ), // Container
        SizedBox( // SizedBox
        buildListTile(…
        buildListTile(…
        buildListTile(…
      ], // <Widget>[]
    ), // Column
  ); // Drawer
}
```

Obrázek 49 - Widget pro vysouvací lištu (Zdroj: vlastní)

Drawer widget je rozdělen do tří částí. První část je horní lišta s nadpisem a ikonkou. Celá je uzavřena ve widgetu *Container*, který má nastavené vlastnosti výška, šířka, vnitřní odsazení, zarovnání obsahu, barva a potomek, který je obsahem celého kontejneru. Druhá část je widget *SizedBox*, který svými vlastnostmi patří mezi základní. Slouží pouze k vytvoření prázdného obsahu, který má nastavenou výšku a šířku. Je to takový oddělovač. Třetí část obsahuje odkazy na případné stránky aplikace.



Obrázek 50 - Grafické znázornění drawer widgetu (Zdroj: vlastní)

Grafické znázornění celého widgetu vidíme na Obrázku 50.

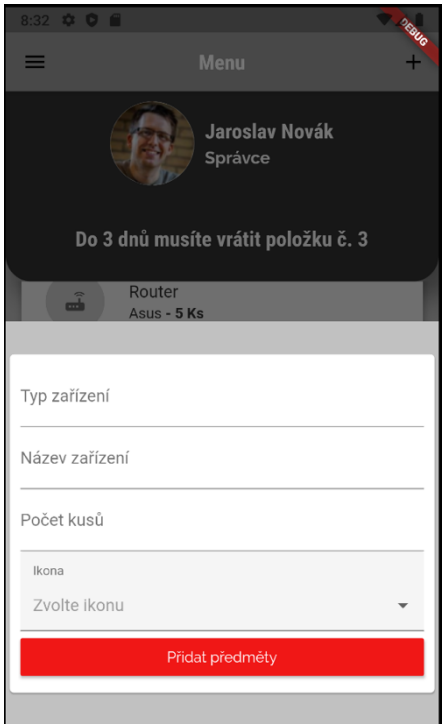
3. **Formulář pro přidávání položek**

```
void _startAddNewItem(BuildContext ctx) {
  showModalBottomSheet(
    context: ctx,
    builder: (_) {
      return GestureDetector(
        onTap: () {},
        child: Center(
          child: NewItem(),
        ), // Center
        behavior: HitTestBehavior.opaque,
      ); // GestureDetector
    },
  );
```

Obrázek 51 - Metoda pro zobrazení formuláře k přidání nové položky (Zdroj: vlastní)

Metoda *_startAddNewItem* je spuštěna po stisknutí tlačítka v podobě ikony znaménka plus. Obsahuje funkci *showModalBottomSheet*, která je již součástí Frameworku Flutter. Základní funkčnost si tak funkce řídí sama, já jsem musel vyplnit pouze její obsah. Celé tělo vysouvací lišty jsem zapouzdřil do widgetu *GestureDetector*, jehož smysl spočívá v tom, že zařízení sleduje, kam na obrazovce

klikám. V případě, že kliknu mimo její obsah, lišta se schová. Tuto funkci jsem zavedl pro příjemnější a intuitivnější ovládání. Samotný obsah se pak ukrývá pod widgetem *NewItem*, který obsahuje formulář a tlačítko na jeho potvrzení a odeslání na server. Grafické znázornění tohoto widgetu vidíme na Obrázku 52.



Obrázek 52 - Grafické znázornění přidávacího formuláře (Zdroj: vlastní)

4. **Metoda pro přidání předmětu do databáze**

```
Future<void> addItem(Item item) async {
  final url = 'https://kvd-app-original.firebaseio.com/
  try {
    final response = await http.post(
      url,
      body: json.encode({
        'type': item.type,
        'name': item.name,
        'count': item.count,
        "icon": item.icon,
        "creatorId": userId,
      }),
    );
    final newItem = Item(
      type: item.type,
      name: item.name,
      count: item.count,
      icon: item.icon,
      id: json.decode(response.body)['name'],
    );
    _devices.add(newItem);
    notifyListeners();
  } catch (error) {
    print(error);
    throw error;
  }
```

Obrázek 53 - Metoda pro přidání předmětu do databáze (Zdroj: vlastní)

Tato metoda je opět asynchronní, protože pracuje s daty ze serveru. Parametr *item* zastupuje hodnoty vyplněné ve formuláři. V bloku *try* se v proměnné provádí dotaz post na adresu, která je v proměnné *url*[40]. Tělem dotazu je *.json* soubor, který je vyplněný daty z formuláře. Jedinou hodnotou navíc je *userId,* které slouží k přiřazení jednotlivých položek k určitému uživateli. Předmět ukládám i do paměti telefonu pro efektivnější manipulaci, a to do proměnné *_devices*, která je datového typu pole. Blok *catch* se spustí v případě chybného vykonání bloku *try*. V takovém případě systém v příkazovém řádku zobrazí vygenerovanou chybovou zprávu.

8.8.2 UŽIVATELSKÁ KARTA

Tato část je vyobrazena na Obrázku 47 v modrém poli. Celý její obsah je ve statické podobě. Je neměnný a hodnoty nejsou stahovány z databáze. Uvědomuji si to jako možný nedostatek, ale z důvodu, že služba Firebase je od určitého obnosu dat placená, jsem ukázkovou aplikaci chtěl držet na co nejmenším objemu serverových dat.

Skládá se tedy z kontejneru, který má určité proporce. Uvnitř je obrázek uživatele a jeho jméno spolu s funkcí, kterou v systému zastupuje. Pod těmito informacemi je textová zpráva, která by mohla sloužit jako upozornění na možné vypršení výpůjční doby jednotlivých položek. Kompletní kód k této kapitole se nachází na přiloženém CD pod touto cestou:

`~/kvd_app/lib/widgets/userCard.dart,`

[40] URL není kompletní

1. **Kontejner uživatelské karty**

```
Container(
  height: (mediaQuery.size.height -
          appBar.preferredSize.height -
          mediaQuery.padding.top) *
      0.3,
  width: double.infinity,
  decoration: BoxDecoration(
    boxShadow: [
      BoxShadow(
          color: Colors.black38,
          blurRadius: 20,
          spreadRadius: 5,
          offset: Offset(
            10,
            10,
          )) // Offset // BoxShadow
    ],
    color: Color.fromRGBO(51, 51, 51, 1),
    shape: BoxShape.rectangle,
    borderRadius: BorderRadius.only(
      bottomLeft: Radius.circular(25),
      bottomRight: Radius.circular(25),
    ), // BorderRadius.only
  ), // BoxDecoration
  child: Column( // Column
) // Container
```

Obrázek 54 - Kontejner widgetu UserCard (Zdroj: vlastní)

Kontejner má nastavenou dynamickou výšku tak, aby splňoval vlastnosti responzivního designu. Celý jeho obsah tedy bude vždy zabírat 30 % celkové výšky zařízení. Šířka má také dynamickou hodnotu *double.infinity*, což znamená, že obsah kontejneru se vždy roztáhne po celé šířce zařízení. Vlastnost *decoration* kontejneru nastavuje stín, který má odstín černé barvy a určité hodnoty dosahu a průhlednosti. Celý kontejner má tmavě šedou barvu a tvar obdélníku, který má zakulacené oba spodní rohy.

2. **Obsah kontejneru uživatelské karty**

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  crossAxisAlignment: CrossAxisAlignment.center,
  children: <Widget>[
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        CircleAvatar(
          maxRadius: 40,
          minRadius: 30,
          backgroundImage: AssetImage("assets/img/avatar.png"),
        ), // CircleAvatar
        Padding(
          padding: const EdgeInsets.only( // EdgeInsets.only
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
              Text( // Text
              SizedBox(
                height: 5,
              ), // SizedBox
              Text( // Text
            ], // <Widget>[]
          ), // Column
        ), // Padding
      ], // <Widget>[]
    ), // Row
    Padding(
      padding: const EdgeInsets.all(20),
      child: Text("Do 3 dnů musíte vrátit položku č. 3",
          style: Theme.of(context).textTheme.headline6), // Text
```

Obrázek 55 - Obsah uživatelské karty v podobě widgetů (Zdroj: vlastní)

Celý obsah je zapouzdřen ve widgetu *Column*, který má tři vlastnosti. První vlastnost je *mainAxisAlignment*, která určuje rozestupy na hlavní ose sloupce. Hlavní osa v případě widgetu *Column* je osa Y. Druhá vlastnost *crossAxisAlignment* určuje zarovnaní obsahu ve sloupci na křížové ose, tedy ose X. Třetí vlastnost je *children* neboli potomek, který je obsahem widgetu *Column*.

Potomek je rozdělen do dvou widgetů. První z nich je *Row*, ten všechny své prvky skládá do řady. Mezi ně tedy patří avatar v podobě obrázku, textové pole se jménem a textové pole s funkcí daného uživatele. Druhý widget je *Padding,* který je hodně podobný widgetu *Container* a klidně by jím mohl být v tomto případě nahrazen. V případě, kdy jsem textovému poli potřeboval nastavit pouze vnitřní odsazení, hodil se zde *Padding* více. Tento widget má potomka v podobě textového pole, které obsahuje upozorňující zprávu.

8.8.3 DYNAMICKÝ POSUVNÝ SEZNAM

Tato část je graficky vyobrazena na Obrázku 47 v růžové sekci. Celý seznam je dynamicky generován. Při načtení obrazovky se zavolá metoda, která naváže spojení se serverem, požádá o položky daného uživatele, pošle je zpět a následně je zobrazí na obrazovce zařízení. Seznam je generován v podobě jednoduchých karet, které se skládají z ikony, typu, názvu a počtu. Na každou z položek je implementována funkce smazání pomocí přejetí prstem. Celý seznam je od určitého počtu položek posuvný. Kompletní kód k této kapitole se nachází na přiloženém CD pod touto cestou:

```
~/kvd_app/lib/widgets/listItem.dart,
~/kvd_app/lib/providers/item.dart,
~/kvd_app/lib/providers/items.dart,
```

1. **Grafické zobrazení jednotlivých položek**

```
child: ListTile(
  leading: CircleAvatar(
    backgroundColor: Color.fromRGBO(206, 206, 206, 1),
    radius: 35,
    child: FittedBox(
      child: Icon(_whatIcon),
      fit: BoxFit.cover,
    ), // FittedBox
  ), // CircleAvatar
  title: Text(
    "$type",
    style: TextStyle(
      color: Color.fromRGBO(51, 51, 51, 1),
    ), // TextStyle
  ), // Text
  subtitle: RichText(
    text: TextSpan( // TextSpan
```

Obrázek 56 - Widget ListTile pro zobrazení položky (Zdroj: vlastní)

Struktura *ListTile* widgetu je dána samotným Flutterem. Vlastnost *leading* určuje, který prvek bude v kartě jako vedoucí. V mém případě se opět jedná o widget *CircleAvatar*, ve kterém zobrazuji ikonu, kterou si uživatel při vytváření položky zvolil. Dále je zde vlastnost *title*, do které zadáváme titulek karty. Jedná se o standartní widget *Text*, který obsahuje hodnotu načítanou z databáze v podobě typu zařízení. Této hodnotě je nastavena barva. Vlastnost *subtitle* je podtitulek,

který obsahuje widget *RichText*, který dovoluje zadávat textové hodnoty do různých spanů a jednotlivě je tak formátovat. Tento widget obsahuje dvě textové hodnoty v podobě jména zařízení a jeho počtu.

2. **Načítání položek ze serverové databáze**

```
Future<void> fetchAndSetProducts() async {
  var url =
      'https://kvd-app-original.firebaseio.com/items.json?auth=$authToken&ord
  try {
    final response = await http.get(url);
    final extractedData = json.decode(response.body) as Map<String, dynamic>;
    final List<Item> loadedItems = [];
    if (extractedData == null) {
      return;
    }
    extractedData.forEach((prodId, prodData) {
      loadedItems.add(Item(
        id: prodId,
        type: prodData['type'],
        name: prodData['name'],
        count: prodData['count'],
        icon: prodData['icon'],
      ));
    });
    _devices = loadedItems;
    notifyListeners();
  } catch (error) {
    throw (error);
  }
}
```

Obrázek 57 - Metoda pro načtení položek z databáze Firebase (Zdroj: vlastní)

Tato metoda se zavolá před načtením úvodní stránky. V bloku *try* se provede http dotaz *get* na adresu z proměnné url. V následující proměnné *extractedData* se dekóduje *.json* soubor do proměnné datového typu *Map*, kde klíče budou datového typu *String* a hodnoty datového typu *dynamic*. Pokud dotaz nevrátí žádná data celá metoda se ukončí a seznam zůstane prázdný. V opačném případě se každý z předmětů načte do proměnné *loadedItems*, která je datového typu pole. Tato proměnná se pak přepíše do privátní proměnné _devices. V případě výkazu nějaké chyby se vykoná blok *catch*.

8.9 PŘÍPADNÁ VYLEPŠENÍ APLIKACE

Přiložená aplikace slouží hlavně k ukázkám základního používaní Frameworku Flutter. Tyto ukázky jsou zjednodušené a ve finální aplikaci by vyžadovaly úpravy. V této kapitole uvedu příklady možných řešení, které by dle mého názoru v kompletní aplikaci neměly chybět.

1. **Zajištění funkčnosti tlačítka „Zapomněli jste heslo?“**

   Toto tlačítko není momentálně funkční. Uvedl jsem ho zde pouze z důvodu ukázky jako jiné možnosti zvolení tlačítka v podobě jiného widgetu. Finální verze by měla zobrazit formulář, například v podobě vyskakovacího okna, kde by uživatel zadal svůj email, na který by mu přišel odkaz k resetování hesla. Tento odkaz by vedl zpět do aplikace, kde by pomocí formuláře uživatel zadal nové heslo.

2. **Zajištění funkčnosti tlačítka „Požádat správce o údaje“**

   V této části by mohl například uživatel vyplnit jednoduchý formulář, který by obsahoval celé jméno, funkci a textové pole pro volitelnou poznámku pro správce. Celý tento formulář by se odeslal správci na mail, případně přímo do systému aplikace, kde by mohl být interní chat nebo mailový klient.

3. **Přidávání jednotlivých položek**

   Momentálně lze položky zadávat jen přes formulář. Ten by v případě hromadného přidávání více položek mohl být uživatelsky nepřívětivý. Místo formuláře bych zde použil fotoaparát, který by zaznamenával QR kódy. Program by z nich do systému stahoval potřebná data. To by velmi urychlilo přidávání položek a zefektivnilo práci s aplikací.

4. **Umožnit správci přidávat/odebírat uživatele**

   Aplikace momentálně neumožňuje spravovat jednotlivé uživatele. Celá správa se tak musí řídit interně ze systému Firebase, což je opět uživatelsky nepohodlné. Aplikaci by šlo vylepšit o novou stránku, která by se zpřístupnila pouze správci celého systému. Stránka by obsahovala kompletní přehled uživatelů, které by správce mohl spravovat.

5. **Seznam uživatelů a jejich vypůjčených položek**

   Tato funkce by umožnila získat přehled o tom, kdo má položku aktuálně vypůjčenou. Uživatelé by se tak mohli společně domlouvat a v případě potřeby vypůjčení již vypůjčené položky i kontaktovat.

6. **Modifikace profilu**

   Další možnou funkcí je modifikace vlastního profilu. Ta může spočívat v podobě změny obrázku, jména, hesla, ale i třeba grafického rozhraní celé aplikace.

7. **Rezervační systém**

   Pro ještě lepší využitelnost zařízení by do aplikace mohl být integrován rezervační systém. Ten by mohl být formou nástěnky, kde by uživatelé, například do tabulky, vyplňovali požadované rezervace na jednotlivé dny. Všichni by tak měli přehled o tom, který den, kdo a co potřebuje a podle toho se přizpůsobil.

8. **Interní chat**

   Funkce interního chatu by byla spíše pro zvýšení komfortu a eliminaci používání dalšího systému. Jednalo by se o chatovací aplikaci, ve které by se uživatelé systému mohli rychle zkontaktovat.

9. **Notifikace**

   Aplikace by uživateli mohla posílat notifikace, které by jej například upozorňovaly na možné vypršení výpůjční lhůty dané položky, na zprávu z chatu, či možné aktualizace.

ZÁVĚR

Hlavním cílem práce bylo seznámit s Frameworkem Flutter, který jako soubor nástrojů a knihoven slouží k vývoji multiplatformních mobilních aplikací pro systémy Android a iOS. Práce byla koncipována tak, aby čtenář danou problematiku co nejlépe pochopil. K tomuto cíli napomáhá zejména tvorba ukázkové aplikace, při níž se propojují teoretické a praktické poznatky.

V první polovině práce jsem se věnoval teoretickému představení samotného Frameworku. Popisuji zde jeho základní součásti a funkčnosti, dále pak programovací jazyk Dart, na kterém celý Framework funguje. V následující kapitole se věnuji widgetům, základním stavebním prvkům, ze kterých se Flutter aplikace tvoří. Představeny jsou i balíčky, které práci s Frameworkem usnadňují. V další kapitole popisuji vygenerovanou strukturu Flutter projektu, význam složek i základních souborů. Dále se věnuji rozdílům ve vývoji aplikací pro systémy Android a iOS. V závěru teoretické části představuji a popisuji používané vývojové nástroje.

Ve druhé části práce se prakticky zaměřuji na tvorbu ukázkové aplikace, sloužící jako systém pro evidenci a správu pracovníků a zařízení KVD. Tento blok názorně seznamuje s procesem tvorby aplikace od návrhu jejího uživatelského rozhraní, jeho převodem do widgetů Frameworku, až po programování funkčních částí aplikace, např. pro přihlášení uživatele. Čtenáře tak seznamuji s vizuální i logickou funkčností celého projektu. Jednotlivé části ukázkové aplikace jsou představeny postupně, a to pomocí obrázků, ukázek kódu i jeho popisu, s cílem čtenáře co nejlépe provést jejím vývojem.

V závěru práce uvádím možná doplnění a vylepšení, kterými by bylo možné aplikaci modifikovat.

Součástí diskového média, které je k práci přiloženo, je i zdrojový kód aplikace a vyexportovaný soubor .apk, který lze do Android zařízení nainstalovat. K práci jsem přiložil návod, jak postupovat v případě, že by si čtenář chtěl vytvořit vlastní export aplikace pro systémy Android nebo iOS.

Všechny stanovené cíle práce se tak podařilo naplnit.

RESUMÉ

This bachelor‘s thesis is dedicated to Framework Flutter, which is a new technology in the development of multiplatform mobile applications. In the first part the thesis is focused on the theoretical description of this technology, and also on the programming language Dart, widgets, and packages that the user can use. Then, development differences for Android and iOS systems, Framework development tools, and the file structure generated by Flutter project. The second part of the work is focused on the process of developing a sample application for the Department of Computing and Didactic Engineering. I try to present the development of the application in graphic displays for a better understanding of the issue. The sample application is part of the CD attachment to this work.

SEZNAM LITERATURY

CLOW, M., 2019. *Learn Google Flutter Fast: 65 Example Apps*. ISRC ISBN-10.

api.flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://api.flutter.dev/flutter/widgets/Row-class.html

api.flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://api.flutter.dev/flutter/widgets/Row-class.html

api.flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://api.flutter.dev/flutter/widgets/Column-class.html

api.flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://api.flutter.dev/flutter/widgets/Container-class.html

api.flutter.dev, 2020. *Flutter.dev* [online] [cit. 2020-Březen-20]. Dostupné z: https://api.flutter.dev/flutter/widgets/Text-class.html

dart.dev, 2020. *www.dart.dev* [online] [cit. 2020-Únor-10]. Dostupné z: https://dart.dev/guides/language/sound-dart

FLUTTER, 2020. flutter.dev. In: *flutter* [online].2020 [cit. 2020-březen-05]. Dostupné z: https://flutter.dev/docs/development/ui/widgets

FLUTTER, 2020. flutter.dev. In: *flutter* [online].2020 [cit. 2020-duben-06]. Dostupné z: https://flutter.dev/docs/resources/platform-adaptations

Flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z:https://flutter.dev/docs/get-started/flutter-for/web-devs

flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z:https://flutter.dev/docs/development/ui/widgets-intro

flutter.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://flutter.dev/docs/development/packages-and-plugins/using-packages#css-example

flutter.dev, 2020. *Flutter.dev* [online] [cit. 2020-Březen-21]. Dostupné z: https://flutter.dev/docs/development/ui/widgets-intro

Flutter.dev, 2020. *Flutter.dev* [online] [cit. 2020-Březen-21]. Dostupné z: https://api.flutter.dev/flutter/widgets/Text-class.html

NAPOLI, M. L., 2019. *Beginning Flutter*. Wrox. ISRC ISBN.

pub.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z: https://pub.dev/packages/url_launcher

pub.dev, 2020 [online] [cit. 2020-Březen-20]. Dostupné z:https://pub.dev/packages/http

SCHWARZMÜLLER, M., 2019. *Flutter basics* [PDF dokument].

stacksecrets, 2019 [online] [cit. 2020-červen-04]. Dostupné z: https://stacksecrets.com/flutter/breaking-down-flutter-project-file-folders

SEZNAM OBRÁZKŮ

PŘÍLOHA 1 - INSTALACE POTŘEBNÉ K POUŽÍVÁNÍ FRAMEWORKU FLUTTER

V této příloze popisuji postup instalacemi potřebných pro správnou funkčnost Frameworku a jeho nedílných součástí.

Budu zde provádět instalace těchto aplikací:

- Framework Flutter
- Xcode (pouze MacOS uživatelé)
- Android studio
- Visual Studio Code (nebo jiný editor)
- Virtuální emulátor (alespoň jeden pro každý systém)

INSTALACE FRAMEWORKU FLUTTER

Při instalaci[41] Flutter SDK si můžeme vybrat ze dvou možností. Stáhnout poslední stabilní verzi, která je na stránkách Flutter vždy dostupná nebo zvolit druhou možnost naklonování oficiálního Flutter úložiště ze stránek GitHub. Doporučuji však stáhnout poslední stabilní verzi.

Stažený soubor dekomprimujeme do předem zvolené složky, nejlépe však do uživatelského adresáře, například *~/User/development/tools/flutter*. Tato cesta bude důležitá při definování globální proměnné *$PATH* v operačním systému, kterou je třeba vytvořit pro využití Flutter SDK v příkazovém řádku. To uděláme tak, že vytvoříme, či pozměníme *rc* soubor našeho příkazového řádku. Já používám Z shell, to znamená, že si vytvořím soubor v uživatelském adresáři ~/User/.zshrc, do kterého vložím tento příkaz:

```
export PATH="$PATH:/User/development/tools/flutter/bin"
```

Soubor uložíme, a pro ověření, zdali se vše vydařilo, spustíme v příkazovém řádku příkaz *flutter doctor*, který by měl ukázat, které součásti potřebné pro práci s Flutterem máme již nainstalované, případně které je třeba doinstalovat.

XCODE

K základní konfiguraci nám postačí, když Xcode stáhneme buď z oficiálních stránek Apple[42], nebo ze samotného App storu. Po stažení a následné instalaci stačí nakonfigurovat

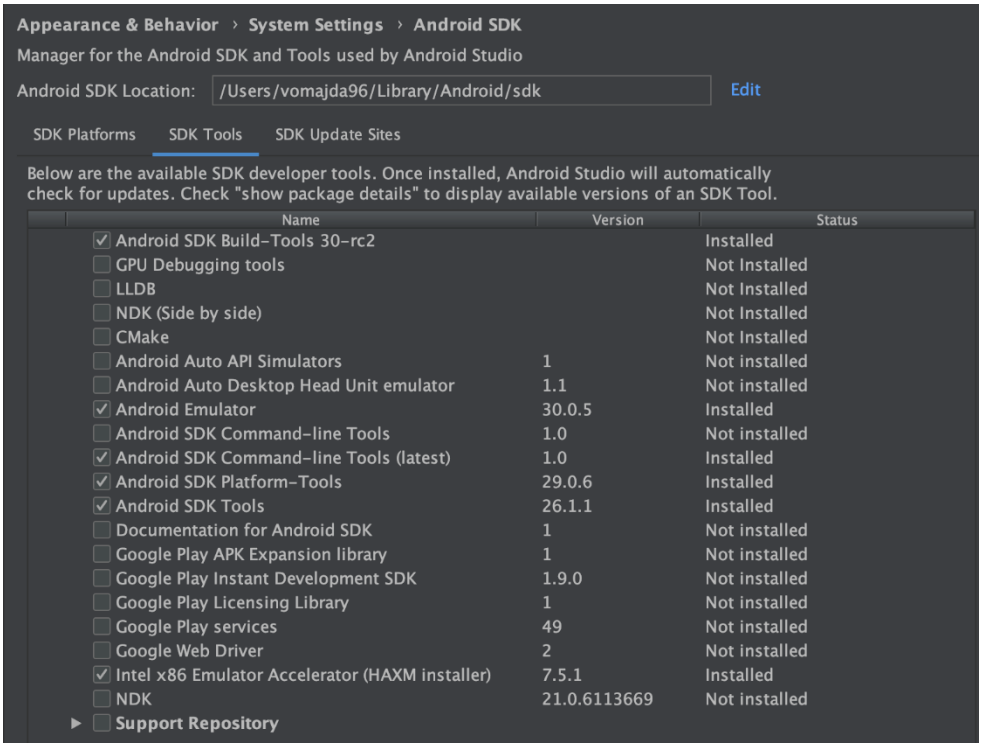
[41] **Instalace Flutter SDK** - https://flutter.dev/docs/get-started/install
[42] **Xcode** - https://developer.apple.com/xcode/

příkazovou řádku, abychom mohli používat potřebné nástroje v jakékoliv vrstvě souborového systému. Konfiguraci provedeme zadáním těchto příkazů do příkazové řádky:

1. *sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer*
2. sudo xcodebuild -runFirstLaunch
3. *sudo xcodebuild -license*

ANDROID STUDIO

Po instalaci[43], kterou nás provede běžný instalační průvodce, je třeba zjistit, zdali se nám nainstalovaly i potřebné nástroje. Po zapnutí Android studia půjdeme do nastavení *configure -> SDKmanager -> SDKtools,* kde by měly být zaškrtnuty minimálně tyto položky viz Obrázek 58.



Obrázek 58 - Konfigurace Android SDK (Zdroj: vlastní)

VISUAL STUDIO CODE

Visual studio code[44] stáhneme z oficiálních stránek tohoto editoru. Instalací nás provede běžný průvodce instalací.

[43] **Android studio** - https://developer.android.com/studio
[44] **Visual studio code** - https://code.visualstudio.com

Flutter balíček[45], který je nutno stáhnout do VS code nám pomůže při vývoji Flutter aplikací. Spolu s ním se stáhne rozšíření i pro programovací jazyk Dart. Toto rozšíření nám nabízí funkce pro různé úpravy a členění kódu jako takového, ale i funkce pro jednoduché spuštění, vypnutí nebo znovu načtení aplikace skrze Android, či iOS simulátor.

ANDROID/IOS VIRTUÁLNÍ EMULÁTORY

Pro testovaní a zobrazení vytvořené aplikace budeme potřebovat virtuální emulátory, které je třeba stáhnout do programů Xcode a Android studio.

V případě **Android studia** tuto možnost najdeme v *Configure->AVD Manager->Create Virtual Device,* kde si zvolíme jeden z nabízených mobilních telefonů a vybereme mu vhodný obraz systému, který je třeba stáhnout. Takto vytvořený emulátor můžeme spustit dvěma způsoby. Přímo z Android studia, kde si v *AVD Manager* najdeme námi vytvořený emulátor a následně jej spustíme, nebo z našeho IDE Visual studio code, které je s Android studiem propojené a pod položkou *No Device*, která je umístěna na spodní liště GUI, najdeme název našeho Android emulátoru.

Pro **Xcode** je tento postup trochu méně obsáhlý, protože základní emulátory bývají součástí instalace. V případě potřeby konkrétního typu mobilního telefonu je zde možnost simulátory doinstalovat. Po zapnutí Xcode tuto možnost nalezneme v *Preferences-> Components*, kde uvidíme obsáhlý list verzí systému iOS, pod kterými se nachází i ona zařízení, která tyto verze podporují. Simulátor iOS můžeme opět pustit přímo z našeho editoru stejným způsobem jako Android simulátor, s tím rozdílem, že tentokrát vybereme zařízení iPhone. Další způsob je spuštění přes příkazovou řádku pomocí příkazu *open -a simulator*.

[45] **Flutter rozšíření** - https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter

PŘÍLOHA 2 - VYDÁNÍ APLIKACE

V této příloze popisuji jednotlivé kroky pro vydání aplikace. Je třeba připomenout, že pokud uživatel nevlastní MacOS zařízení, není zde oficiální a legální způsob, jak aplikaci vytvořit a posléze vydat na platformu App Store. Obě možnosti vytvoření vývojářského účtu jsou placené a je třeba s tím počítat.

GOOGLE PLAY STORE

První věc, co musíme udělat je vytvořit si tzv. *keystore*, který slouží jako naše známka, či podpis pod aplikacemi. Pro vytvoření napíšeme do příkazového řádku tento příkaz:

keytool -genkey -v -keystore ~/key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias key[46]

Po zadání příkazu je třeba vyplnit naše osobní údaje. Takto vygenerovaný klíč je pouze náš a nikdo by k němu neměl mít přístup. Takže pokud naši aplikaci ukládáme například na GitHubu, měl by být soubor s klíčem zaznamenán v souboru .gitignore.

Nyní k tomuto klíči potřebujeme vytvořit cestu v našem projektu tak, že ve složce *android* vytvoříme soubor s názvem *key.properties* a přidáme do něj následující reference, které vyplníme:

```
storePassword=<heslo z předešlého kroku>
keyPassword=< heslo z předešlého kroku>
keyAlias=key
storeFile=<cesta ke klíči, příklad: /Users/<user name>/key.jks>
```

Nyní je třeba nakonfigurovat soubor *~/android/app/src/build.gradle*. To uděláme ve dvou krocích. V prvním kroku přidáme před blok *android{}* odkaz na soubor, který obsahuje vygenerovaný klíč:

```
def keystoreProperties = new Properties()
def keystorePropertiesFile = rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
  keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
}

android {
   ...
}
```

[46] Pro **Windows**: keytool -genkey -v -keystore c:/Users/USER_NAME/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key

V druhém kroku přidáme před blok *buildTypes{...}* konfiguraci s přihlašovacími údaji:

```
signingConfigs {
  release {
    keyAlias keystoreProperties['keyAlias']
    keyPassword keystoreProperties['keyPassword']
    storeFile keystoreProperties['storeFile'] ?
           file(keystoreProperties['storeFile']) : null
    storePassword keystoreProperties['storePassword']
  }
}
buildTypes {
  release {
    signingConfig signingConfigs.release
  }
}
```

Pro vygenerování aplikace, která je vhodná k distribuci na Google play je v kořenovém adresáři třeba spustit příkaz v příkazové řádce, a sice *flutter build appbundle*. Vygenerovaná aplikace bude v tomto adresáři:

<kořenový_adresář_aplikace>/build/app/outputs/bundle/release/app.aab

Takto vygenerovanou aplikaci již můžeme vložit na Google Play přes Google Play Console[47]. Pro tento úkon je však nutné zaplatit registrační poplatek 25 $. Po zaplacení je tento proces velmi intuitivní a webová aplikace vás navede. Stručný postup k interakci v Google Play:

Create Application -> App releases -> Production -> Manage -> Create Release -> vložíme aplikaci, která je umístěna pod cestou výše.

Pro podrobnější postup a možné změny je v případě vydání aplikace pro Android nutné číst oficiální dokumentaci Flutter[48].

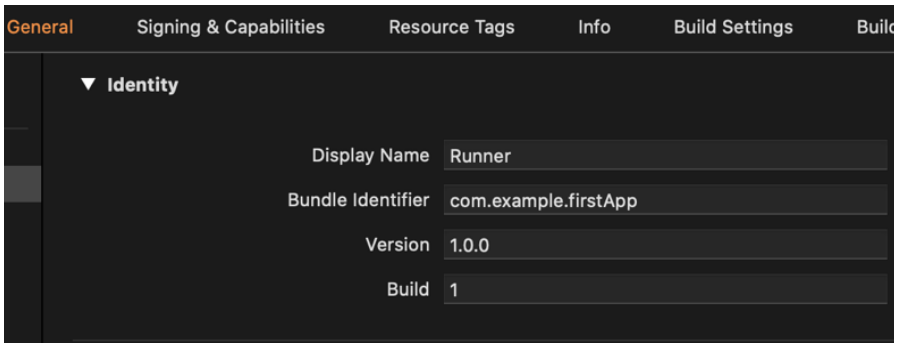
APP STORE

Pro vydání aplikace na systémy iOS je třeba mít předplacený vývojářský účet, který vyjde na 99$ za jeden rok. Další podmínkou je vlastnit počítač se systémem MacOS, jinak nemůžeme aplikace vytvářet a ani je vydávat. Pokud splňujeme předchozí podmínky, aplikaci přidáme v několika krocích.

1. Pod sekcí *App IDs*[49] registrujeme nové unikátní ID pro náš Bundle. To uděláme tak, že vyplníme jednoduchý formulář, kde například volíme platformu, jednoduchý popisek účelu aplikace, ale hlavně **explicitní ID**, které najdeme v souboru s názvem

[47] **Google Play Console -** https://play.google.com/apps/publish/signup/
[48] **Vydání aplikace pro systém Android -** https://flutter.dev/docs/deployment/android
[49] **App IDs -** https://developer.apple.com/account/resources/

Runner.xcodeproj, který v Xcode musíme otevřít. Samotné ID je pak v sekci *General/Identity/Bundle_Identifier.* Část ID, která obsahuje *com.example,* změníme na unikátní hodnotu, například *com.vomackap.firstApp* (viz Obrázek 59). Po vyplnění našeho ID, jej můžeme registrovat.



Obrázek 59 - Xcode Bundle ID (Zdroj: vlastní)

2. Přihlásíme se na stránce App Store Connect[50], kde pod položkou *MyApps* přidáme novou aplikaci. Vyplníme jméno aplikace, primární jazyk, v seznamu najdeme naše Bundle ID, které jsme vytvořili v předešlém kroku a potvrdíme.

3. Nyní můžeme vytvořit samotný build pro naší iOS aplikaci. V kořenovém adresáři naší aplikace spustíme v příkazové řádce následující příkaz *flutter build ios.* V tuto chvíli znovu spustíme v aplikaci Xcode soubor *Runner.xcodeproj* a v sekci *Product/Scheme* zaškrtneme položku Runner, další položku *Generic iOS Device* zaškrtneme v sekci *Product/Destination.* Finální Bundle vytvoříme v sekci *Product* a zvolíme položku *Archive*. Po dokončení se nám otevře nové okno, kde naši aplikaci můžeme vydat do App Store, či ji nejprve validovat a následně vydat.

Pro podrobnější postup a možné změny je v případě vydání aplikace pro iOS nutné číst oficiální dokumentaci Flutter[51].

SPRÁVA VERZE

Verze naší aplikace se spravuje pouze v souboru *pubspec.yaml*, který je v kořenovém adresáři naší aplikace. Hned v úvodu je položka *version: 1.0.0+1.* Pokud tuto položku změníme, například když budeme aplikaci opravovat, či do ní přidávat nějaká nová rozšíření, Flutter si ji automaticky vezme při finální kompilaci pro Android, či iOS Bundle.

[50] **App Store Connect -** https://appstoreconnect.apple.com/login
[51] **Vydání aplikace pro systémy iOS** - https://flutter.dev/docs/deployment/ios

Část za samotnou verzí, tedy *+1* slouží pro náš přehled kolikrát jsme aplikaci měnili a museli tak přepsat její verzi. Při každé změně bychom měli toto číslo navýšit.