FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Mobilní aplikace pro výuku hry na brumle
Student:	Xuan Tam Trinh
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je návrh a implementace specializováné mobilní aplikace určené k výuce hry na brumle. Odborným konzultantem zadání bude brumlista Petr Jasinčuk.
Aplikace bude sestávat z části pro hráče a administrátorského rozhraní, které umožní přidávat a měnit obsah kurzu. Výsledná aplikace bude lokalizována do angličtiny a češtiny.

Postupujte v těchto krocích:
1. Vypracujte rešerši podobných aplikací pro jiné hudební nástroje.
2. Vyjděte z analýzy zpracované v nedokončené bakalářské práci na stejné téma, validujte předchozí analýzu s konzultantem práce a formalizujte požadavky na systém.
3. Pomocí metod softwarového inženýrství proveďte návrh aplikace.
4. K implementaci použijte platformu Flutter, návrh implementujte a řádně otestujte.
5. Zhodnoťte výsledek a navrhněte další možný rozvoj.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 23. září 2020 v Praze.

ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F8

Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Mobilní aplikace pro výuku hry na brumle

Xuan Tam Trinh
Webové a softwarové inženýrství

9. května 2021
Vedoucí práce: Ing. Michal Valenta Ph.D.

Poděkování / Prohlášení

Chtěl bych poděkovat svému vedoucímu Ing. Michalovi Valentovi Ph.D., za jeho odborné vedení, pomoc a ochotu při tvorbě bakalářské práce. Mé poděkování patří taktéž odbornému konzultantovi Petru Jasinčukovi za jeho přátelský přístup a trpělivost při konzultacích. Dále bych chtěl také poděkovat všem, kteří mě podporovali v průběhu psaní této práce.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 9. května 2021

.....

Abstrakt / Abstract

Tato bakalářská práce se zabývá analýzou, návrhem a implementací multiplatformní mobilní aplikace s pomocí technologie Flutter. V práci je proveden výzkum konkurenčního trhu a poskytuje stručný popis využitých technologií. V jednotlivých částí práce jsou představeny fáze klasického vývoje software — analýza požadavků, návrh řešení, implementace a testování. Výstupem je funkční mobilní aplikace, která slouží pro výuku hry na brumle poskytováním výukových materiálů ve formě videí a doprovodných textů.

Klíčová slova: mobilní výuková aplikace, vývoj mobilních aplikací, Flutter, Firebase, Clean Architecture, multiplatformní framework, brumle

This bachelor thesis deals with the analysis, design and implementation of a multiplatform mobile application using Flutter SDK. The thesis conducts a competitive analysis and provides a brief description of the technologies used. Each part of the thesis provides an insight into classical software development phases — requirements analysis, design, implementation and testing. The result is a functional mobile application that is used to provide jaw harp lessons in the form of videos and accompanying texts.

Keywords: mobile learning application, mobile application development, Flutter, Firebase, Clean Architecture, cross-platform framework, jaw harp

Obsah /

Tabulky / Obrázky

Úvod

Význam mobilních telefonů v každodenním životě časem nepochybně nabývá na velikosti. Je tomu tak proto, že mobilní telefony již nejsou běžným komunikačním zařízením, jakým bývaly kdysi, nýbrž s rapidně rostoucím technickým pokrokem dnešní doby se staly univerzálními zařízeními, které jsou centrem zábavy, práce ale také vzdělání. Kumulativní pokrok mobilních technologií vyvolalo revoluci, která přiměla studenty a učitele k častějšímu používání elektronických zařízení pro výukové činnosti. Tato skutečnost se stala motivací právě pro Petra Jasinčuka, který chce využít trh mobilních aplikací pro výuku hry na brumle.

Brumle je lidový hudební nástroj patřící do skupiny idiofonů, jenž je tvořen kovovým či bambusovým jazýčkem, který hráč vloží do úst a brnkáním vytváří tón. Jelikož se jedná o hudební nástroj lidového charakteru předávaný z generace na generaci, hrozí brumlím postupem času částečné či úplné vymizení v některých zemích. Proti tomuto problému lze bojovat využitím mobilních technologií a tvorbou specializované výukové platformy, která bude volně dostupná a přístupná pro zaujaté hráče, což je právě tématem této bakalářské práce.

Práce se zabývá analýzou, návrhem a následnou implementací specializované multiplatformní výukové aplikace pro výuku hry na brumle, která bude fungovat na mobilních operačních systémech iOS a Android. Aplikace bude vyvíjena ve spolupráci s výše zmíněným lektorem a dlouhodobým hráčem na brumle Petrem Jasinčukem, který bude spolu s menší komunitou brumlistů využívat tuto platformu jako prostředek k poskytování kvalitních výukových materiálů primárně formou videí a doprovodných textů, jejichž cíl je zvýšení povědomí a zájmu o hru na brumle u široké veřejnosti. Zároveň je tato práce navázáním na nedokončenou bakalářskou práci, která byla předčasně ukončena ve fázi sběru požadavků.

Tato práce je rozčleněna do sedmi kapitol. V první kapitole jsou představeny jak hlavní, tak dílčí cíle práce. V druhé kapitole je proveden průzkum konkurenčního trhu s obdobnými výukovými aplikacemi a zhodnoceny jejich silné a slabé stránky. Dále, ve třetí kapitole, je provedena komplexní analýza funkční a nefunkčních požadavků, která je následně podpořena případy užití a analytickým doménovým modelem. Čtvrtá kapitola je věnována rozboru technologie Flutter, která je využita k implementaci aplikace. V páté kapitole je realizován návrh architektury, serverové části a uživatelského rozhraní. Poté, v šesté kapitole, jsou popsány použité vývojové nástroje, vysvětleny konkrétní ukázky implementace, zhodnoceny výsledky a představeny další možnosti pro budoucí rozvoj. V kapitole sedmé jsou probrány metody, které byly využity k testování aplikace během vývoje.

Kapitola 1
Cíl práce

Hlavním cílem bakalářské práce je vytvořit specializovanou výukovou mobilní aplikaci, která bude vyhovovat potřebám lektora a hráče na brumle Petra Jasinčuka. Výuková platforma bude sestávat z uživatelské části pro hráče a administrátorské části pro správce, která umožní přidávat a měnit obsahu kurzu. Výsledná aplikace bude lokalizována do českého a anglického jazyka, přičemž bude operovat na mobilních platformách iOS a Android. K implementaci bude použita technologie Flutter, která umožňuje multiplatformní vývoj mobilních aplikací.

K dosažení hlavního cíle je nutné splnit několik dílčích cílů. V úvodní části vývoje je potřeba provést průzkum trhu s mobilními aplikacemi, identifikovat stejná či podobná řešení a vyhodnotit jejich slabé a silné stránky, což bude nesmírně užitečné pro zjištění požadavků koncových uživatelů. Poté, co bude utvořena základní představa o správné výukové aplikaci, je důležité vykonat komplexní analýzu funkčních a nefunkčních požadavků. Jako podklad k analýze budou využity materiály předchozí nedokončené bakalářské práce, která byla ukončena ve fázi sběru požadavků. Další významnou částí vývoje, kterou je nutné se zabývat, je rozbor technologie Flutter, jež bude použita k implementaci aplikace, a následně vyhodnoceny její výhody a nevýhody. Poté je důležité vytvořit návrh architektury, serverové části a uživatelského rozhraní aplikace, tak aby vyhovovala funkčním i nefunkčním požadavků a zároveň neobsahovala nedostatky konkurenčních řešení. V závěrečné fázi je potřeba celý návrh realizovat a řádně ho otestovat.

Kapitola 2
Rešerše současných řešení

Analýza trhu je nesmírně důležitá pro vyhodnocení silných a slabých stránek existujících řešení, které jsou podobné povahy jako vyvíjená aplikace. Tyto informace poslouží dále jako podklad k analýze požadavků a jsou nezbytné pro tvorbu kvalitní aplikace, která má potenciál být spotřebiteli dobře přijata, jelikož lze již ve fázi návrhu eliminovat nedostatky, které jsou zjištěny u konkurenčních aplikací. [1]

V následujících podkapitolách jsou vybrány tři hudební výukové aplikace, u kterých je proveden rozbor a zhodnocení jejich kladných i záporných stránek z pohledu autora práce a také koncových uživatelů.

2.1 Metodika

V první řadě je nutné identifikovat existující řešení. Průzkum je proveden zejména na distribučních platformách Google Play a App Store s cílem nalézt výukové aplikace, které jsou specializované pouze na jeden konkrétní hudební nástroj, poskytující lekce primárně ve formě videí. Navíc, aby byl průzkum opravdu relevantní, jsou k analýze vybrány pouze populární aplikace s větším počtem uživatelů, které přesahují hranici alespoň 10 000 stažení na obou distribučních sítích.

Dalším krokem je zhodnocení silných a slabých stránek vybraných aplikací. Kritériem pro zhodnocení kvality aplikace jsou především hodnocení koncových uživatelů na oficiálních stránkách distribučních sítí [2–7] a zároveň také subjektivní posudek autora práce, který si aplikace nainstaluje do vlastních mobilních zařízení* a bude používat po dobu několika dní.

2.2 Analýza vybraných řešení

V době psaní této práce se na trhu mobilních aplikací pro mobilní operační systémy Android a iOS nenachází žádné řešení orientované přímo na brumle. Přesto však existuje řada populárních výukových aplikací pro jiné hudební nástroje, které mají natolik recenzí a uživatelských instalací, aby bylo možné na nich provést podrobnější analýzu. Konkrétně se bude tato podkapitola zabývat následujícími aplikacemi: TrueFire, flowkey a Drumeo.

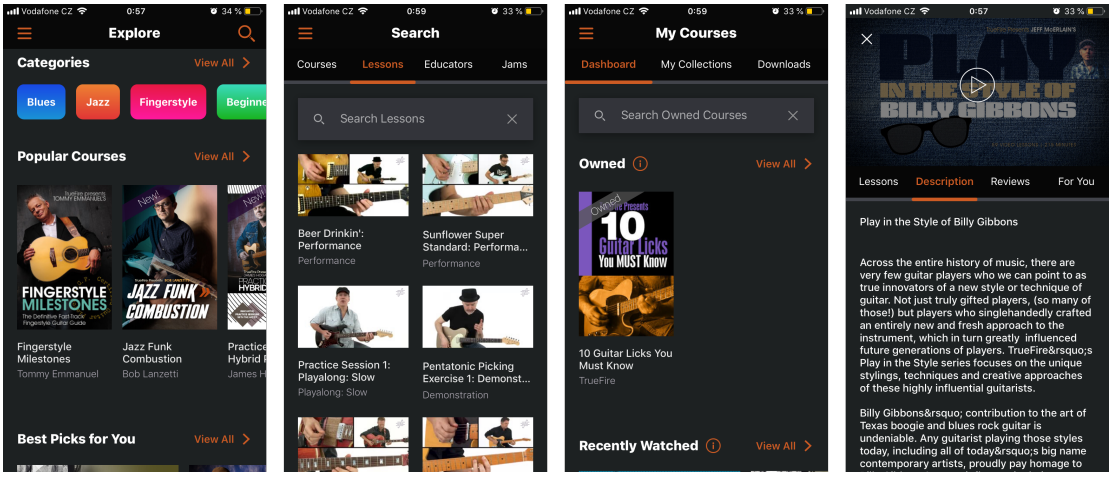
2.2.1 TrueFire Guitar Lessons

TrueFire je výuková platforma zaměřená na kytary, která kromě videí nabízí také interaktivní cvičení. Jedná se o opravdu rozsáhlou platformu s řádově desítek tisíc výukových materiálů pro různě pokročilé hráče. Aplikace Truefire je dostupná jak na desktopových operačních systémech Windows, macOS a Linux, tak i na mobilních platformách iOS a Android.

* iPhone 6 pro iOS aplikace, Xiaomi Redmi 9A pro android aplikace

Z pohledu uživatelské přívětivosti a použitelnosti je aplikace velmi kvalitně zpracovaná a nabízí velmi jednoduché, intuitivní a přehledné uživatelské rozhraní, ve kterém se lze snadno orientovat. Po registraci a přihlášení je uživatel uvítán nabídkou nejnovějších kurzů a nejpopulárnějších lekcí. Lekce jsou vedeny formou videí, u kterých lze měnit kvalitu a rychlost přehrávání, nastavit přehrávání ve smyčce či přetáčet je v čase. Uživatelé mají také možnost stáhnout video do lokálního úložiště pro pozdější přehrávání v režimu offline.

Dle uživatelských recenzí a osobního testování autora práce se aplikace však potýká s problémy se stabilitou. Aplikace se často zasekává, místy se sama vypne a obsahuje velký počet drobných chyb, které odrazují uživatele od používání mobilní verze. Na verzích pro android se dokonce vypíná obrazovka během sledování videa při neaktivitě uživatele nebo při obnovení zastavené aplikace zmizí u spuštěného videa ovládací prvky. Další slabou stránkou aplikace je absence cachování — aplikace nabízí obrovské množství obsahu, které se při každém požadavku načítá znovu. Při testování na osobním zařízení při běžném domácím připojení o rychlosti stahování 50 Mbps se obsah načítal průměrně 5–8 sekund při každém požadavku.



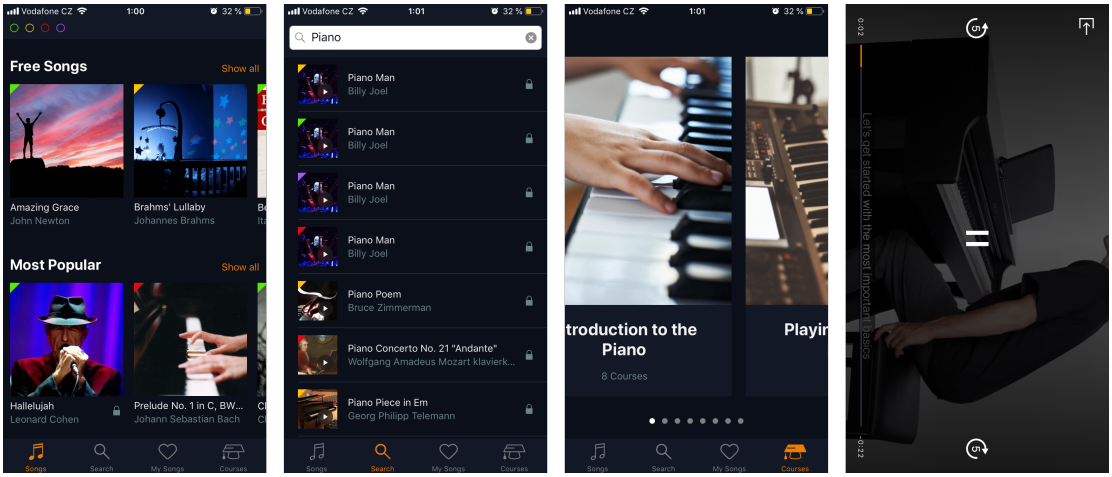
Obrázek 2.1. Ukázky aplikace TrueFire Guitar Lessons [2]

2.2.2 flowkey – Learn Piano

Flowkey je mobilní výuková aplikace specializovaná na klavírní hru. Obdobně jako u aplikace TrueFire je obsah tvořen formou výukových videí a interaktivních cvičení. Aplikace nemá desktopovou verzi a je podporována pouze na mobilních operačních systémech Android a iOS.

Uživatelské prostředí je taktéž jednoduché a přehledné ve velmi podobném stylu jako u předchozí aplikace. Ovládací prvky videí umožňují pouze přetáčení v čase, přičemž stahování do lokálního úložiště zde není podporováno a videa lze pouze spustit v horizontální orientaci na šířku. Aplikace umožňuje cachování obsahu, a tak se videa při druhém spuštění nemusí celé znova načíst, což zvyšuje uživatelský komfort.

Slabou stránkou aplikace jsou poměrně chudé ovládací prvky u videí — nelze změnit kvalitu, rychlost přehrávání či nastavit přehrávání ve smyčce. Uživatelé si tak nemohou nastavit vlastní tempo, což může ztížit učení na hudební nástroj. Dalším negativem je také chybějící stahování lekcí, což má za následek, že k používání aplikace je vyžadováno připojení k internetu.



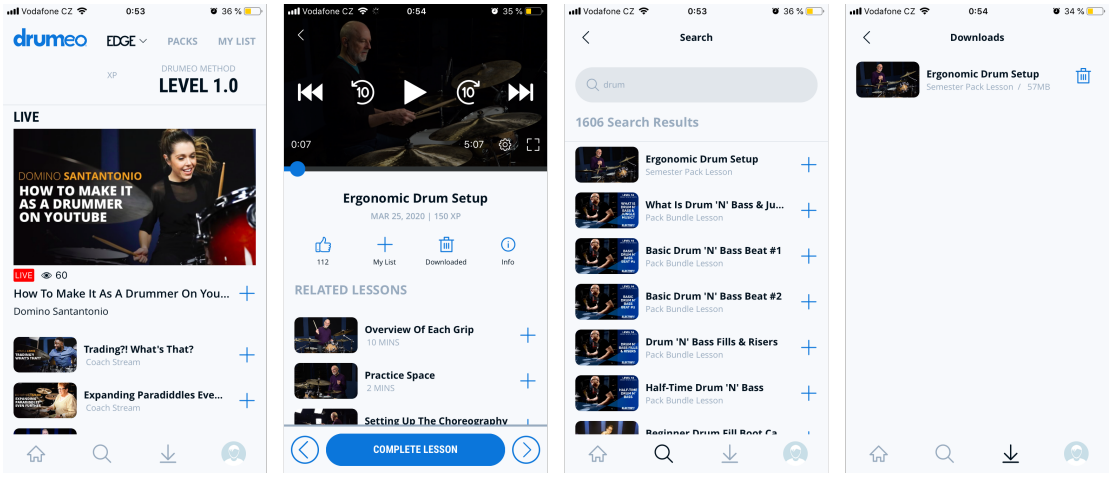
Obrázek 2.2. Ukázky aplikace flowkey [3]

2.2.3 Drumeo

Aplikace Drumeo se specializuje výukou hry na bicí nástroje. Lekce jsou koncipované čistě formou videí a aplikaci lze instalovat pouze na mobilních operačních systémech Android a iOS.

Uživatelské prostředí svoji formou připomíná známou platformu YouTube, a proto je aplikace z hlediska použitelnosti velmi intuitivní. U videí lze měnit rychlost a kvalitu přehrávání, přetáčet je v čase nebo pomocí technologie *Chromecast built-in* video streamovat do externích zařízení. Dále aplikace podporuje také stahování videí do úložiště zařízení. Oproti předchozím aplikacím Drumeo nenabízí tolik funkcionalit, ale zároveň má velice dobré uživatelské recenze. To je hlavně proto, že aplikace je stabilní, neobsahuje příliš mnoho chyb a plní to, k čemu byla navržená.

Hlavní nevýhodou aplikace je absence možnosti si aplikaci vyzkoušet. Aplikace je sama o sobě bezplatná, avšak při registraci je nutné se přihlásit ke zpoplatněnému měsíčnímu odběru. Toto není úplně chyba aplikace, ale spíše chyba návrhu UX, čemuž je také důležité věnovat pozornost.



Obrázek 2.3. Ukázky aplikace Drumeo [4]

2.3 Závěr

Z uživatelských recenzí je zjevné, že hlavním kritériem hodnocení je především věcný obsah aplikace. Avšak vedle obsahu jsou pro spotřebitele důležité podpůrné nástroje, které fungují bez chyb a usnadňují učení na hudební nástroj. Není tedy potřeba uživatele ohromit obrovským počtem funkcionalit, nýbrž je prioritou vytvořit jednoduché a stabilní prostředí, které maximálně podpoří výuku hry na hudební nástroj. Tato skutečnost je obzvlášť dobře pozorovatelná při porovnání aplikací TrueFire a Drumeo. Aplikace TrueFire je velmi interaktivní, umožňuje uživatelům personalizovat obsah a poskytuje mnoho dalších funkcionalit za cenu toho, že aplikace je nestabilní, obsahuje spoustu nedokonalostí a ve výsledku uživatelé často přecházejí od mobilní k desktopové verzi. Oproti tomu aplikace Drumeo umožňuje pouze prohlížení videí a jejich stahování do lokálního úložiště, ale z pohledu funkčnosti je zcela v pořádku, a proto byla taky aplikace koncovými uživateli velmi pozitivně přijata.

Při návrhu bude tedy kladen důraz na přívětivé UI a využijí se silné stránky výše uvedených aplikací jako je cachování obsahu pro zrychlení doby načítání, podpora stahování lekcí do lokálního úložiště a bohaté ovládací prvky pro kontrolu videí.

Dále, jelikož všechny výše uvedené aplikace mají velice podobné charakteristiky, lze si na základě analýzy utvořit základní představu o tom, jak by mohla vyvíjená aplikace vypadat. Tedy tato rešerše poslouží jako velmi užitečný podklad, který může výrazně urychlit sběr, vyjednávání a analýzu požadavků, jimiž se bude zabývat další kapitola.

Kapitola 3
Analýza požadavků

Aby bylo vůbec možné začít vývoj aplikace, je důležité nejprve identifikovat potřeby všech zúčastněných stran a definovat požadavky na vyvíjenou aplikaci. Je tedy nutné provést analýzu požadavků, což je proces, který obnáší sběr, validaci a dokumentaci všech požadavků. [8, s. 11] Důležitost analýzy požadavků nelze podceňovat. Správná analýza požadavků je pro úspěšný projekt klíčová, neboť ovlivňuje všechny následující vývojové fáze aplikace. Pokud analýza požadavků není provedena dostatečně kvalitně, bude zapotřebí vynaložit další úsilí k nápravě chyb. Čím později v průběhu životního cyklu vývoje se vyskytnou problémy spojené se špatným pochopením požadavků, tím více prostředků a času bude stát oprava. [9, s. 66]

Tato kapitola se bude zabývat právě specifikací funkčních i nefunkčních požadavků, jejich následným upřesněním pomocí analýzy případů užití a tvorby analytického doménového diagramu.

3.1 Popis aplikace

Aplikaci lze pro přehlednost rozdělit do tří modulů, které obsahují spolu související funkcionality, a to jmenovitě do autentizační, uživatelské a administrátorské části.

Při spuštění aplikace má uživatel možnost se přihlásit pomocí emailové adresy či facebookového profilu. Případně, pokud je uživatel v aplikaci poprvé nebo nemá vytvořený účet, se zaregistruje a následně je povinen ověřit svoji e-mailovou adresu, na kterou je zaslána ověřovací zpráva. Teprve po registraci a úspěšném přihlášení je uživateli umožněn vstup do aplikace.

Při vstupu do uživatelské části jsou uživateli zobrazeny nejnovější lekce. U jednotlivých lekcí lze vidět obrázek, název, obtížnost, kategorii a cenu. Pro zobrazení lekce musí platit, že buďto je vybraná lekce zdarma anebo je zpoplatněná a zároveň ji uživatel v minulosti zakoupil. V opačném případě pro odemknutí lekce je uživatel vyznán k zaplacení stanovené finanční částky. Při vstupu do detailu lekce uživatel může zhlédnout výukové video a přečíst si k němu doprovodný materiál, který se skládá z krátkého popisku a podrobnějšího textu. Pokud má uživatel zájem lekci se učit bez připojení k internetu, existuje možnost si obsah stáhnout do lokálního úložiště zařízení pro pozdější offline zobrazení. Dále, k urychlení hledání a zlepšení orientace v aplikaci, by měl mít uživatel možnost filtrovat a řadit lekce dle zadaných parametrů.

Administrátorská část aplikace je zobrazena pouze uživatelům, kteří mají dostatečná přístupová privilegia. Tuto část tvoří formuláře pro přidávání a editaci obsahu jednotlivých lekcí a kategorií, které umožní správu aplikace přímo z pohodlí mobilního zařízení. Veškeré provedené změny se po uložení projeví při dalším spuštění aplikace — tedy veškerá data se nachází ve vzdáleném úložišti.

3.2 Sběr a validace požadavků

Přestože projekt navazuje na nedokončenou bakalářskou práci, u které již sběr a validace proběhla, získané požadavky jsou velmi obecně definované a je velká šance, že mohou být zastaralé. Z toho důvodu bylo nutné sběr a validaci provést znovu. Provedená analýza nedokončené práce však významně urychlila identifikaci a pochopení požadavků.

Správné zachycení požadavků je klíčové, a proto je nutné zajistit, aby jednotlivé požadavky splňovaly správné vlastnosti, které během vývoje aplikace maximálně sníží riziko zanesení chyb spojených se specifikací požadavků. Podle [10, s. 56] by měl každý požadavek splňovat následující vlastnosti:

- **Srozumitelný** – Dobré požadavky jsou jasné, stručné a snadno pochopitelné. Požadavky nemohou být vágně definované či nekompletní a musí jim rozumět všechny osoby, které se účastní vývoje.
- **Jednoznačný** – Kromě toho, že požadavky musí být jasné a konkrétní, je nutné, aby požadavky byly také jednoznačné. U každého požadavku musí být zcela jasné kritérium splnění. Tedy co všechno se musí provést, aby se mohl považovat za splněný.
- **Konzistentní** – Požadavky je důležité za každou cenu udržovat navzájem v souladu. To neznamená jen to, že si nemohou navzájem odporovat, ale nesmí ani klást tolik omezení, aby byly některé požadavky nesplnitelné. Jinými slovy, je velice důležité, aby každý požadavek byl realizovatelný.
- **Prioritizovaný** – Při vývoji se může stát, že z časových či finančních důvodů bude potřeba některé požadavky vynechat. Každý požadavek bude tedy mít definovanou prioritu podle metody MoSCoW:
  1. *Must have* jsou požadavky, které musí být za jakýchkoliv okolností splněny. Jedná se o požadavky kritické k naplnění zájmů všech zúčastněných stran.
  2. *Should have* představují požadavky, které jsou důležité a měly by být splněny, pokud je to možné. Nezbyde-li pro tyto požadavky prostor, mohou být realizovány v další verzi systému.
  3. *Could have* jsou chtěné požadavky, které mohou být však bez následků vynechány, pokud to bude nutné.
  4. *Won't have* reprezentují plně volitelné požadavky, na kterých se všechny strany shodly, že mohou být realizovány až v pozdějších verzích systému.

3.3 Funkční požadavky

Funkční požadavky zachycují konkrétní chování, které musí vyvíjený systém splňovat, aby mohly zúčastněné strany uspokojit své potřeby. Dále určují také, jak systém funguje v daném kontextu, jak reaguje na vtupy a jaké jsou jeho výstupy. [11]

a) **Autentizace**

- **FP1 – Registrace – must have**
  Uživatelé si mohou v aplikaci vytvořit nový účet.
- **FP2 – Ověření účtu – must have**
  Pro dokončení registrace musí uživatelé ověřit svoji emailovou adresu.
- **FP3 – Přihlášení pomocí emailové adresy – must have**
  Uživatelé se mohou do aplikace přihlásit pomocí emailové adresy a hesla.
- **FP4 – Přihlášení pomocí facebookového profilu – could have**
  Uživatelé se mohou do aplikace přihlásit pomocí facebookového profilu.
- **FP5 – Odhlášení – must have**
  Přihlášení uživatelé se mohou odhlásit z aplikace.

b) **Uživatelská část**

- **FP6 – Zobrazení lekcí – must have**
  Lekce jsou definované názvem, kategorií, popiskem, doprovodným textem, obtížností, výukovým videem, náhledovým obrázkem a cenou. Při zobrazení lekce mohou nastat dvě situace. Lekce je zdarma nebo ji má uživatel zakoupenou, a tak je následně přesměrován do obsahu lekce. V druhém případě, kdy uživatel nemá lekci zakoupenou, je uživatel vyzván k zaplacení.
- **FP7 – Kategorizace lekcí – must have**
  Uživatelé mohou procházet kategorie, do kterých jsou rozděleny jednotlivé lekce.
- **FP8 – Nákup lekcí – must have**
  Některé lekce mohou být zpoplatněny, proto aplikace musí podporovat nákupy a platby.
- **FP9 – Stahování lekcí – should have**
  Aplikace umožňuje lekce stahovat do lokálního úložiště zařízení. Stažené lekce lze poté zobrazit i bez připojení k internetu.
- **FP10 – Řazení lekcí – won't have**
  Seznam lekcí je možné seřadit alespoň dle data přidání.
- **FP11 – Filtrování lekcí – won't have**
  Aplikace umožňuje uživatelům filtrovat alespoň odemčené a uzamčené lekce.

c) **Administrátorská část**

- **FP12 Autorizace – must have**
  Aplikace zobrazuje obsah uživatelům na základě jejich role. Uživatelé bez administrátorských privilegií mohou pouze zobrazovat a nakupovat techniky. Správcům aplikace je navíc umožněn přístup do administrátorské sekce pro správu lekcí a kategorií.
- **FP13 – Správa lekcí – must have**
  Administrátoři mohou přidávat nové lekce a měnit jejich obsah.
- **FP14 – Správa kategorií – must have**
  Administrátoři mohou přidávat nové kategorie a přiřazovat jim lekce.

3.4 Nefunkční požadavky

Zatímco funkční požadavky definují, co systém musí nebo nesmí dělat, nefunkční požadavky jsou vlastnosti, které musí být splněny, aby řešení mohlo fungovat efektivně a bez omezení. Příklady nefunkčních požadavků mohou představovat vlastnosti, jako jsou výkon aplikace, rozšiřitelnost, testovatelnost, bezpečnostní charakteristiky apod. Jak již plyne z názvu, nefunkční požadavky neovlivňují základní funkčnost systému. Systém bude tedy i nadále plnit svůj základní účel i přes to, že funkční požadavky nebudou splněny. Nicméně, nefunkční požadavky jsou stejně tak důležité jako funkční, jelikož mají přímý dopad na použitelnost a spolehlivost systému. [11]

- **NP1 – Flutter – must have**
  K vývoji aplikace je zvolena technologie Flutter.
- **NP2 – Multiplatformnost – should have**
  Aplikace bude dostupná na mobilních systémech Android a iOS. Minimální podporovaná verze OS závisí na systémových požadavcích technologie Flutter.
- **NP3 – Lokalizace – could have**
  Aplikace je lokalizována do anglického a českého jazyka.

3.5 Analýza případů užití

Funkční požadavky jsou typicky stručné a pouze nastiňují výsledné chování systému, což málokdy stačí k tomu, aby bylo možné si vytvořit stejnou představu o systému jako mají zadavatelé. Je tedy potřeba funkční požadavky dále upřesnit a ukázat na konkrétních příkladech — na případech užití. Případ užití je tedy situace, kdy vyvíjený systém je použit k splnění jednoho nebo více z funkčních požadavků, přičemž zachycuje část funkčnosti, kterou systém poskytuje. [12, s. 20]

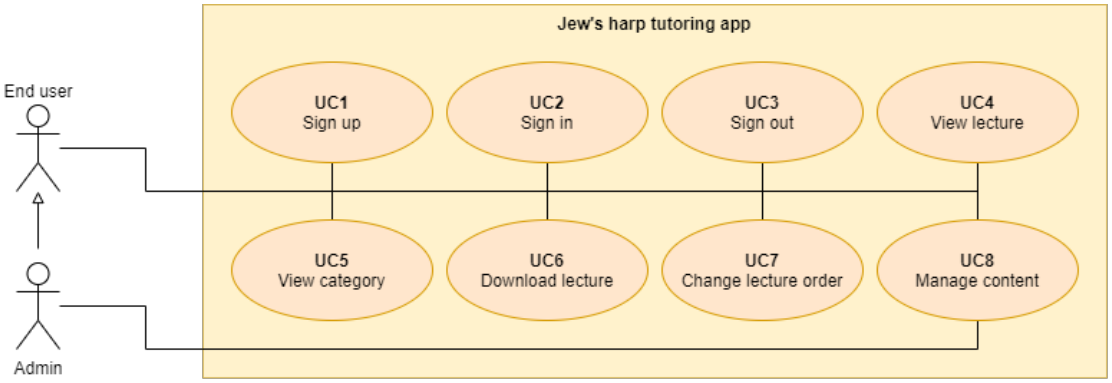
3.5.1 Aktéři

Prvním krokem k namodelování případů užití je identifikovat všechny prvky, které interagují s vyvíjeným systémem — tyto prvky se nazývají aktéry. [12, s. 22]

Z popisu aplikace v sekci 3.1 lze vyvodit dva hlavní aktéry: koncový uživatel a administrátor. Koncový uživatel představuje všechny osoby, které užívají aplikaci k osobnímu rozvoji v oblasti hraní na brumle. Administrátoři jsou koncoví uživatelé, kteří mají navíc administrátorská privilegia a mohou měnit obsah aplikace.

3.5.2 Případy užití

Jakmile jsou zachyceni všichni aktéři, kteří interagují se systémem, lze dále pokračovat sestavením modelu těchto interakcí. Tento model lze vyjádřit formou diagramu případů užití jako je na obrázku 3.1.



Obrázek 3.1. Diagram případů užití

Diagram případů užití je dobrým výchozím bodem pro začátek, avšak sám o sobě zdaleka neposkytuje dostatek podrobností, aby dokázal významně podpořit pochopení funkčních požadavků. Z toho důvodu je nutné k jednotlivým případům užití v diagramu přidat jejich textový popis. [12, s. 28]

Forma popisu případů užití není jednoznačně určena a mnohdy závisí na potřebách metodiky vývoje. Vzhledem k počtu požadavků a složitosti případů užití byl zvolen následující formát:

1. **Identifikátor** – Identifikační kód a jméno případu užití.
2. **Kontext** – Při jaké situaci může případ užití nastat.
3. **Související požadavky** – Funkční požadavky, které případ použití splňuje.
4. **Předpoklady** – Podmínky aby se mohl případ užití uskutečnit.
5. **Úspěch** – Jaký by měl být stav systému, pokud se případ použití úspěšně provede.
6. **Neúspěch** – Co se může během případu užití pokazit.
7. **Aktéři** – Netriviální aktéři, kteří se účastní případu užití.
8. **Hlavní scénář** – Posloupnost kroků, které popisují normální průběh případu užití.
9. **Vedlejší scénář** – Alternativní posloupnost kroků, jak splnit případ užití.

Tabulka 3.1. Identifikované případy užití

Kód	Název	Popis
UC1	registrace	příloha B.1
UC2	přihlášení	příloha B.2
UC3	odhlášení	příloha B.3
UC4	zobrazení lekce	příloha B.4
UC5	zobrazení kategorie	příloha B.5
UC6	stažení lekce	příloha B.6
UC7	změna uspořádání lekcí	příloha B.7
UC8	správa obsahu	příloha B.8

Všechny případy užití včetně jejich popisu ve výše uvedeném formátu lze najít v příloze B. V tabulce 3.1 lze vidět všechny identifikované případy užití s referencemi na jejich textový popis.

3.5.3 Pokrytí požadavků

Na závěr analýzy případů užití je důležité ověřit, zda-li případy užití skutečně pokrývají všechny funkční požadavky. Pokrytí jednotlivých funkčních požadavků je znázorněno v tabulce 3.2, která obsahuje na ose X identifikátory případů užití a na ose Y kódy funkčních požadavků. Zaškrtnutá buňka znamená, že daný případ užití realizuje funkční požadavek.

V tabulce lze vidět, že každý řádek obsahuje minimálně jednu zaškrtnutou buňku, z čehož plyne, že identifikované případy užití pokrývají celý rozsah funkčních požadavků.

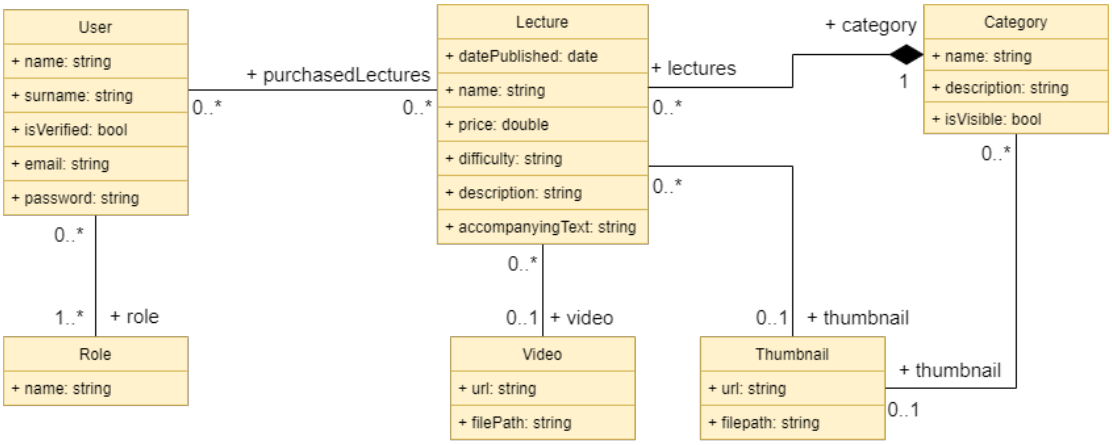
Tabulka 3.2. Pokrytí funkčních požadavků případy užití

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
FP1	X							
FP2	X							
FP3		X						
FP4		X						
FP5			X					
FP6				X				
FP7					X			
FP8				X				
FP9						X		
FP10							X	
FP11							X	
FP12								X
FP13								X
FP14								X

3.6 Analytický doménový model

Poté, co byly analyzovány požadavky na systém a definovány případy užití, je velmi užitečné vytvořit model problémové domény nebo některých jejich částí k identifikování netriviálních závislostí či složitějších detailů, které nejsou na první pohled zřejmé ze specifikace požadavků a případů užití.

Pro tento účel lze využít analytický doménový model, jenž vytváří strukturu propojených objektů zvané entity, na kterých lze vidět jejich vlastnosti, závislosti a vztahy pouze na čistě konceptuální úrovni bez jakýchkoliv implementační detailů. Analytický doménový model je užitečný pro podpoření případů užití a slouží také jako základ pro návrh dalších výstupů softwarového inženýrství jako jsou databázové modely, modely tříd apod. [13, s. 127–128]



Obrázek 3.2. Analytický doménový model

3.6.1 Uživatel

Entita `User` představuje všechny koncové uživatele a je jedna z nejdůležitějších prvků v doménovém modelu, neboť veškerý obsah aplikace budou konzumovat právě uživatelé. Každá osoba z reálného světa má jméno a příjmení. Zároveň všechny osoby používající vyvíjenou aplikaci musí mít kontaktní údaj v podobě emailové adresy. Informaci, zda-li je emailová adresa ověřená, poskytuje příznak `isVerified`. Dále, jelikož může být některý obsah aplikace zpoplatněn, udržuje se u uživatelů informace o zakoupených lekcí. Mezi koncové uživatele patří také administrátoři aplikace, a proto je u všech entit `User` definovaná role. Povinná vazba mezi entitami `User` a `Role` značí, že každý uživatel má implicitně nastavenou roli konzumenta aplikace, což znamená, že i administrátoři mohou zakupovat a odebírat zpoplatněný obsah aplikace.

3.6.2 Lekce

Entita `Lecture` reprezentuje lekce hry na brumle, které jsou hlavním obsah aplikace. Jak bylo zmíněno již v analýze požadavků, každá lekce má svůj název, kategorii, cenu, obtížnost, popisek a doprovodný text. Vizuální část lekce tvoří video a obrázek, které jsou v doménovém modelu také zachyceny, přičemž tvoří s lekcí nepovinnou vazbu. Tedy každá lekce může mít přiřazeno video či obrázek, ale nesmí jich mít více najednou.

3.6.3 Kategorie

Entita `Category` tvoří kategorie, do kterých jsou uspořádány jednotlivé lekce. Kategorie jsou identifikovány podle názvu a krátkého popisku, který charakterizuje jejich obsah. Každá lekce má právě jednu kategorii a kategorie může mít libovolný počet přiřazených lekcí. Pro usnadnění správy aplikace má každá entita kategorie příznak `isVisible`, který umožňuje správcům skrýt kategorii včetně jejího obsahu a kdykoliv později ji opět zveřejnit. Dále může mít každá kategorie volitelně obrázek, který koncovým uživatelům usnadní její identifikaci.

3.6.4 Role

Jako způsob autorizace slouží entita `Role`, která je definovaná pouze svým názvem. Díky rolím mají správci přístup do administrátorského menu a zároveň uživatelům bez oprávnění zůstává menu skryto. Role budou také užitečné při validací citlivých požadavků na serverové straně.

3.6.5 Video a obrázek

Vizuální obsah aplikace tvoří videa a obrázky, které mohou být uloženy vzdáleně či lokálně v úložišti zařízení pro offline zobrazení. Z toho důvodu existuje u obrázků a videí atributy `url` a `filePath`, které identifikují lokaci daného videa či obrázku.

Kapitola 4
Analýza technologie Flutter

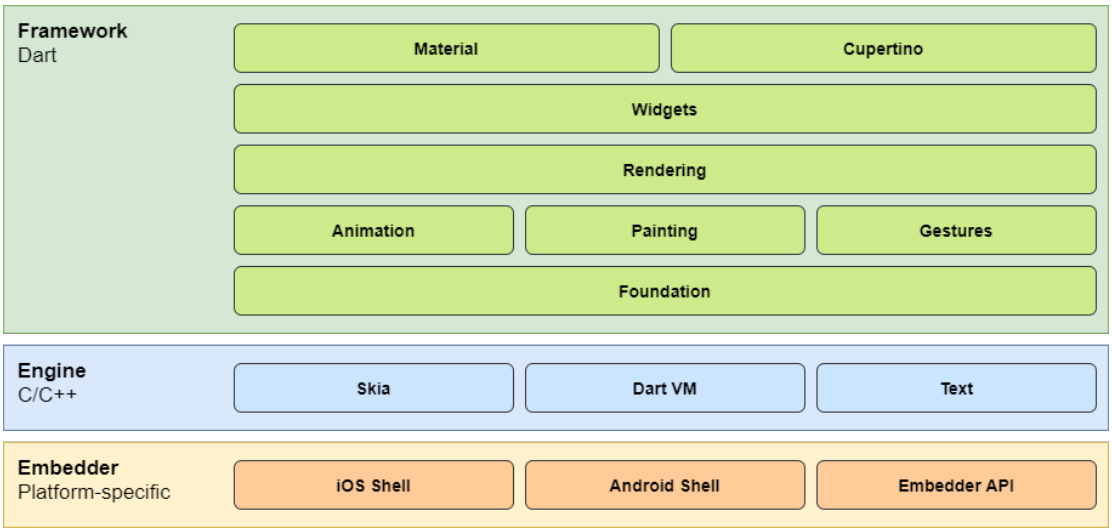
Předtím, než se vývoj aplikace posune do fáze návrhu, je dobré si osvojit a důkladně analyzovat technologii, jež je součástí nefunkčních požadavků. Tato kapitola se zabývá analýzou technologie Flutter, která bude později použita k implementaci mobilní aplikace.

Flutter je nová technologie pro vývoj multiplatformních aplikací od společnosti Google a je navržena tak, aby umožňovala maximální opětovné použití jednoho kódu napříč různými operačními systémy, který zároveň umí přímo interagovat s nativními službami hostitelského zařízení. Hlavním cílem technologie Flutter je umožnit vývojářům dodávat vysoce výkonné aplikace, které jsou výkonnostně i vzhledově téměř nerozeznatelné od nativních. [14]

4.1 Architektura

Technologie Flutter je navržena jako rozšiřitelný modulární systém, který je rozdělen do tří vrstev, jak je znázorněno na obrázku 4.1. Jednotlivé vrstvy jsou na sebe závislé směrem odzdola nahoru a každá hraje v systému důležitou roli:

1. **Framework** je nejvyšší vrstva, která poskytuje nástroje pro vývoj multiplatformních aplikací a slouží jako vstupní bod, pomocí kterého vývojáři interagují s ostatními vrstvami architektury.
2. **Engine** zapouzdřuje klíčové technologie, na kterých je postaven celý systém.
3. **Embedder** tvoří nativní část každé Flutter aplikace, která poskytuje rozhraní pro interakci a přístup k prostředkům cílové platformy. [14]



Obrázek 4.1. Architektura technologie Flutter [14]

4.1.1 Framework

Framework tvoří největší část architektury a je implementován čistě v programovacím jazyce Dart. Framework je dále abstrahován do několika modulů, které poskytují vývojářům různé funkcionality:

- Knihovny `Material` a `Cupertino` nabízejí sadu hotových grafických ovládacích prvků, které vyhovují různým designovým normám. Knihovna `Material` implementuje normu *Material Design* od společnosti Google a `Cupertino` dodržuje design platformy iOS. Díky těmto knihovnám mohou vývojáři dodávat mobilní aplikace, které jsou vzhledově velmi podobné nativním.
- Oproti knihovnám `Material` a `Cupertino`, které jsou z velké části tvořeny hotovými řešeními, knihovna `Widgets` poskytuje vývojářům sadu nástrojů pro tvorbu vlastních grafických elementů.
- Pod knihovnou `Widgets` lze najít vrstvu `Rendering`, pomocí které lze jednoduše modifikovat rozložení a vykreslování jednotlivých grafických prvků. Tato vrstva umožňuje vývojářům například změnit pořadí prvků, změnit jejich barvu, velikost, font apod.
- Vrstva `Foundation`, poskytuje základní třídy a funkce frameworku, které jsou nutné k vývoji každé Flutter aplikace, a jsou na ní závislé všechny výše uvedené vrstvy. Knihovny `Animation`, `Painting` a `Gestures` poskytují užitečné metody a funkce, které slouží jako abstrakce pro práci s touto vrstvou. [14]

4.1.2 Engine

Velká část *Flutter Engine* je implementovaná v programovacím jazyce C++. Velice důležitou součástí engine je grafická knihovna `Skia`, která se stará o vykreslování 2D grafiky. Díky integraci této knihovny je vykreslování grafiky zcela platformě nezávislé a probíhá přímo v technologii Flutter, což má za následek výrazné zvýšení výkonnosti aplikace a dává spoustu prostoru pro nová rozšíření. [14]

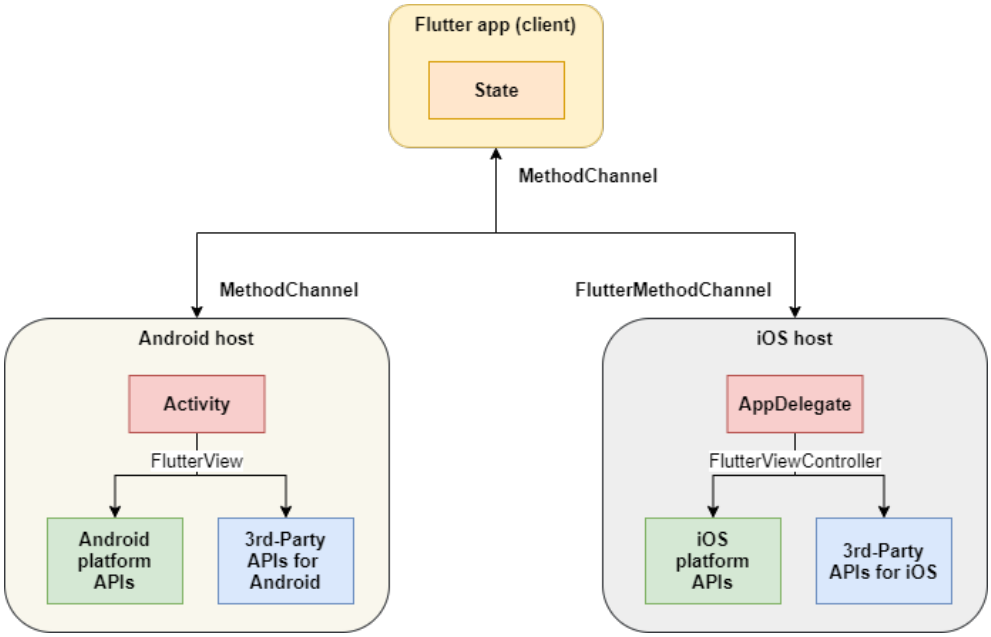
Během vývoje operují všechny Flutter aplikace v běhovém prostředí `Dart VM`, které je také velmi podstatnou součástí engine. `Dart VM` využívá kompilaci *just in time* (JIT), díky které Flutter poskytuje vývojářům funkci *hot-reloading* — vývojáři mohou zároveň psát a ladit software, přičemž pomocí kompilace JIT dokáže Flutter vložit nový kód přímo do spuštěné aplikace. Tato funkce je velmi užitečná, jelikož se tím změny zdrojového kódu projeví ve spuštěné aplikaci bez nutnosti restartování, a tím je ušetřen čas a urychlen vývoj aplikace. Kompilace JIT se využívá pouze během vývojové fáze. Naopak při sestavování aplikace přepne `Dart VM` do kompilačního módu *ahead of time* (AOT). V tomto módu je zdrojový kód zkompilován přímo do nativního strojového kódu, a proto se z pohledu hostitelského zařízení žádná Flutter aplikace neliší od nativní. [15]

4.1.3 Embedder

Nejspodnější vrstva *Flutter Embedder* je napsaná v platformě specifickém programovacím jazyce a slouží jako rozhraní pro přístup ke službám a knihovnám cílové platformy. Technologie Flutter implementuje embedder pro různé platformy jako jsou Android, iOS, macOS, linux, windows atd. Pokud je však aplikace vyvíjena pro platformu, která není oficiálně podporovaná, pomocí rozhraní `Embeder API` je možné implementovat vlastní embedder. [14]

4.2 Nativní rozhraní

Ačkoli technologie Flutter ve většině případů sama o sobě postačí k multiplatformnímu vývoji, existuje však řada situací, kdy řešení problému lze dosáhnout pouze pomocí nativního kódu. Může se například jednat o zjištění aktuálního stavu baterie, přístup k fotoaparátu či využití výhodné knihovny, která je dostupná pouze pro danou platformu. Flutter proto nabízí rozhraní *Platform Channel*, pomocí kterého lze propojit nativní kód s multiplatformní aplikací. Schéma nativního rozhraní lze vidět na obrázku 4.2.



Obrázek 4.2. Schéma nativního rozhraní Flutter [16]

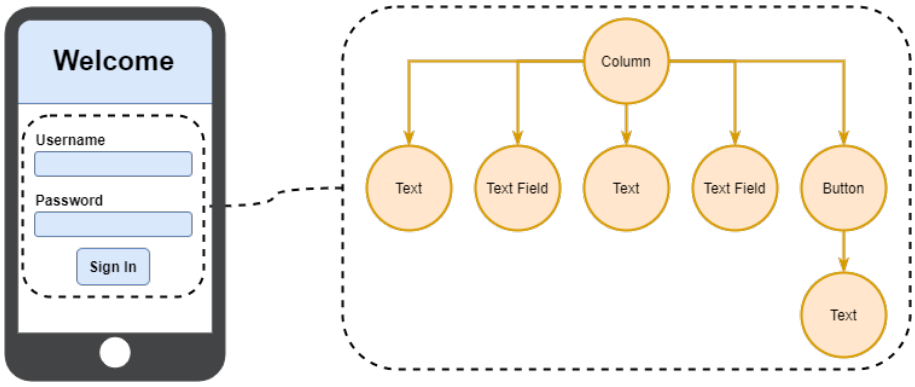
Platform Channel funguje na principu asynchronního odesílání a přijímání zpráv bez generování jakéhokoliv přídavného kódu, což umožňuje implementovat nativní funkcionality s velmi minimálním kódem. Na straně Flutter aplikace pomocí rozhraní `MethodChannel` lze zasílat zprávy, které korespondují volání nativních metod. Na druhém konci hostitelské zařízení vystavuje rozhraní*, které slouží k příjmu zpráv a následnému zaslání odpovědí. V případě potřeby lze volání metod odesílat také v opačném směru, kdy nativní kód na straně hostitelského zařízení bude v roli klienta a Flutter aplikace v roli hostitele. [16]

Jelikož se při volání nativních funkcí přechází do prostředí, které může obsahovat jiné datové typy než používá Flutter, je nutné se zabývat konverzí dat. Například při komunikaci s iOS zařízením proměnná typu `int` musí být z důvodů kompatibility zpracována jako `NSNumber(value: Int32)`. Flutter proto nabízí integrované řešení, které využívá kódování zpráv podporující efektivní binární serializaci jednoduchých hodnot typu `boolean`, `number`, `String`, `byte buffer`, `list` a `map`. Serializace a deserializace datových typů probíhá automaticky při zasílání zpráv mezi různými prostředími a je zcela odstíněna od vývojářů. [16]

* `MethodChannel` pro Android a `FlutterMethodChannel` pro iOS

4.3 Widgety

Jedním z nejdůležitějších konceptů frameworku jsou widgety, což jsou třídy v jazyce Dart, které definují prvky uživatelského rozhraní. Veškerá tlačítka, menu, text, formuláře a vše co lze vidět v uživatelském rozhraní aplikace je reprezentováno v kódu jako widget. Widgety však nemusí reprezentovat pouze viditelné elementy, ale mohou definovat také rozložení prvků, odsazení, animace apod. Tedy ve své podstatě je celé uživatelské rozhraní ve Flutteru kompozicí menších widgetů, které tvoří hierarchickou stromovou strukturu zvanou *Widget Tree*.



Obrázek 4.3. Stromová struktura widgetů

Všechny widgety jsou imutabilní, což znamená, že po jejich inicializaci je nelze dále modifikovat. Při nutnosti aktualizace UI jsou proto dané instance widgetů ve stromové hiearchii nahrazeny novými instancemi, což je velmi levná operace a zvyšuje výkonnost vykreslování elementů. Tato vlastnost má však za následek, že při nahrazení daného widgetu, dochází také k nahrazení všech jeho potomků a celého podstromu. Dost často je tedy žádoucí u některých widgetů si udržovat perzistentní modifikovatelný stav, aby nedocházelo ke ztrátě dat. Proto lze většinu widgetů lze kategorizovat do dvou podtypů: bezstavový `StatelessWidget` a stavový `StatefulWidget`.

4.3.1 StatelessWidget

`StatelessWidget` je vykreslen čistě na základě své vlastní konfigurace a nemění se dynamicky. Jinými slovy, nenese žádné informace, které by mohly být ztraceny při jeho opětovném vykreslení. Veškerý stav a konfigurace jsou zakomponovány přímo do samotného widgetu.

Příkladem může být například jednoduché tlačítko, které bude vždy stejné a lze jej vždy znovu ve stromové hierarchii vyměnit bez vedlejších efektů:

```
class MyButton extends StatelessWidget {
  final String text = "Click Me!";

  /// Builds the widget and renders it to the screen.
  @override
  Widget build(BuildContext context) {
    return TextButton(
      onPressed: () {...},
      child: Text(this.text),
    );
  }
}
```

4.3.2 StatefulWidget

Některé prvky uživatelského rozhraní mohou nést data, která se při překreslování widgetů mohou ztratit. Může se jednat například o načtená data ze vzdálené databáze, uživatelem zadaná hodnota do formuláře, zacachovaný obsah aplikace apod. Flutter proto poskytuje jednoduchý mechanismus pro perzistenci stavů UI komponent ve formě speciálního stavového widgetu.

Stavový widget je tvořen dvěma důležitými třídami: `StatefulWidget` a `State`. Třída `StatefulWidget` obsahuje imutabilní konfiguraci widgetu, podobně jako u `StatelessWidget`. Oproti svému bezstavovému protějšku je však ke každé třídě `StatefulWidget` přiřazen speciální objekt `State`, jenž zapouzdřuje proměnlivý stav widgetu. Při překreslování uživatelského rozhraní dochází tak pouze k nahrazení instance `StatefulWidget`, přičemž odpovídající `State` objekt přetrvává a následně inicializuje nový widget. Tímto způsobem lze zachovat stavy widgetů během aktualizace uživatelského rozhraní a předcházet ztrátám dat.

Příkladem využití `StatefulWidget` je například tlačítko s číslem, které je inkrementováno při každém stisknutí. Pokud by byl pro reprezentaci tlačítka zvolen bezstavový widget, mohlo by při překreslení UI dojít ke ztrátě informace, kolikrát bylo tlačítko předtím stisknuto, a proto je vhodné v tomto jednoduchém příkladu použít `StatefulWidget`:

```
/// Immutable widget configuration.
class MyButton extends StatefulWidget {
  final String label = "Counter: ";

  /// Creates the state object.
  @override
  _MyButtonState createState() => _MyButtonState();
}

/// Persisting widget state.
class _MyButtonState extends State<MyButton> {
  int _counter = 0;

  void _increment() {
    // Updates the state and then rebuilds this widget.
    this.setState(() {
      //Increments the counter by 1.
      _counter++;
    });
  }

  /// Builds the widget and renders it to the screen.
  @override
  Widget build(BuildContext context) {
    return TextButton(
      onPressed: () => this._increment(),
      child: Text(this.widget.label + this._counter.toString()),
    );
  }
}
```

4.3.3 InheritedWidget

Dost často je nutné předávat stavy mezi více obrazovkami aplikace nebo je propagovat dál v hierarchii widgetů. Například v mobilních e-shop aplikacích lze běžně prohlížet obsah nákupního košíku ve více obrazovkách, což je možné jen díky tomu, že stav nákupního košíku je udržován napříč několika obrazovkami. Samozřejmě, problém lze řešit naivním způsobem, kdy jsou stavy propagovány v aplikaci pomocí konstruktorů. Toto řešení však je problematické, když aplikace nabývá na velikosti a komplexitě, neboť se kód stává neudržitelným a obsahuje navíc spoustu redundantního kódu.

Aby bylo možné efektivně řešit tuto situaci, Flutter poskytuje speciální typ widgetu `InheritedWidget`, který definuje kontext v kořenovém vrcholu daného podstromu. Kontext není nic jiného než reference na umístění ve stromu widgetů a `InheritedWidget` umí tento kontext efektivně doručit do jakéhokoliv potomka v daném podstromu bez ohledu na jeho hloubku. Pomocí kontextu lze rychle lokalizovat daný `InheritedWidget` ve stromu widgetů, a tím získat přístup k jeho datům.

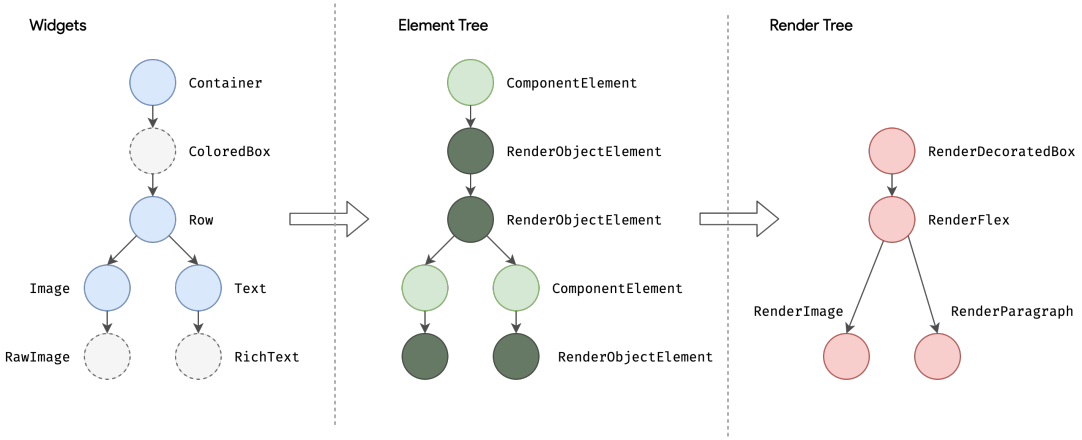
Funkčnost `InheritedWidget` lze předvést na jednoduchém příkladu, kde uživatelská data přihlášeného uživatele je nutné distribuovat mezi více widgetů. V uvedené ukázce jsou uživatelská data zapouzdřená do třídy `UserData`, která jako parametr, kromě jména a příjmení uživatele, dostala instanci třídy `MyWidget`. Pomocí kontextu a metody `dependOnInheritedWidgetOfExactType` může třída `MyWidget` včetně všech jejich potomků získat uživatelská data:

```
/// Widget that provides user data.
class UserData extends InheritedWidget {
  final String name;
  final String surname;
  final Widget child;

  UserData(this.name, this.surname, this.child) : super(child: child);
}

/// Widget that consumes UserData.
class MyWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
   // Get user data from context of the nearest parent InheritedWidget.
    var data = context.dependOnInheritedWidgetOfExactType<UserData>();
    return Text("Hi, " + data.name);
  }
}

/// Some screen.
class MyPage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
     // Wrap MyWidget with InheritedWidget.
    return UserData("John", "Doe", MyWidget());
  }
}
```

4.4 Vykreslování UI

V předchozí podkapitole byl představen koncept widgetů a jejich kompozice do hierarchické stromové struktury. Strom widgetů však není jediný strom, který se vytváří při běhu aplikace — naopak kvůli optimalizaci vykreslování grafiky, Flutter si udržuje dokonce tři různé stromy, z nichž každý má svůj speciální význam. Tyto stromy jsou vytvářeny již při prvním spuštění aplikace a to následujícím způsobem:

1. Flutter vytvoří strom obsahující všechny widgety.
2. Následně Flutter projde strom widgetů od jeho kořene do jednotlivých listů a vytvoří druhý strom, který obsahuje *Element* objekty (dále jako elementy).
3. Třetí strom je vytvořen a naplněn *RenderObjects*, které jsou vytvořeny příslušnými elementy. [17]



Obrázek 4.4. Stromová hierarchie widgetů, elementů a Render objektů[14]

RenderObjects obsahují veškerou logiku pro grafické vykreslování UI, kterou Flutter využívá ke zobrazení odpovídajících widgetů na obrazovku hostitelského zařízení. Jejich instancování je velmi nákladná operace, proto je tedy rozumné tyto objekty udržovat co nejdéle v paměti a maximálně je recyklovat. Právě k tomuto účelu slouží strom elementů, který propojuje *RenderObjects* s widgety. [17]

Jak již bylo v předchozí podkapitole zmíněno, při aktualizaci uživatelského rozhraní se vytváří nový strom widgetů, neboť jsou všechny widgety imutabilní. Poté pomocí elementů Flutter po hladinách porovná nový strom widgetů s již existujícím stromem *RenderObjects*. Element porovnává datové typy dvojic na stejné pozici a pokud se liší, Flutter odstraní celý podstrom *RenderObjects* a nahradí ho novým. V opačném případě, kdy je jejich datový typ stejný, není nutné vytvářet nový *RenderObject* a změní se pouze jeho atributy. Poté, co byl aktualizován strom *RenderObjects*, dojde k překreslení uživatelského rozhraní. [17]

Tento proces je velmi rychlý, jelikož Flutter dokáže efektivně vytvářet widgety, které představují pouze aktuální konfiguraci aplikace, a drahé *RenderObjects* tak zůstanou nedotčené, dokud ze stromu nebude odstraněn odpovídající typ widgetu. Díky této optimalizaci založené na cachování a recyklaci komponent je vykreslování grafiky ve Flutteru velmi rychlé a efektivní i při komplexnějších uživatelských rozhraních. [17]

4.5 Závěr

Přestože Flutter je poměrně mladá technologie, díky velkému počtu optimalizací a jednoduchosti použití, je schopná velmi dobře obstát konkurenčním platformám pro multiplatformní vývoj. Nástroje jako *hot-reload*, nativní rozhraní či knihovny `Cupertino` a `Material` se budou velice hodit při implementaci vyvíjené aplikace. Na závěr kapitoly je vhodné si znovu shrnout výhody a nevýhody, které vyplývají z analýzy této technologie.

4.5.1 Výhody

- **Výkon** – Flutter aplikace fungují výkonnostně na úrovni srovnatelné s nativními aplikacemi a vítězí nad většinou technologií pro multiplatformní vývoj. Hlavně proto, že na rozdíl od přístupu většiny frameworků jsou Flutter aplikace kompilovány pomocí kompilace AOT přímo do nativního strojového kódu cílové platformy.
- **Nativní vzhled** – Díky knihovnám `Material` a `Cupertino`, které nabízejí velký počet widgetů splňující různé designové normy, lze velmi jednoduše vyvíjet aplikace, které jsou vzhledem téměř nerozeznatelné od nativních.
- **Minimum nativního kódu** – Jelikož je zodpovědnost vykreslování grafiky přenechána Flutteru, stačí ve většině případů napsat jeden zdrojový kód, který bude funkční na více platformách. Pokud je potřeba využít některé služby hostitelského zařízení, nativní rozhraní Flutter je založeno na jednoduchém předávání zpráv, a proto je nativní kód minimální.
- **Hot-reload** – Díky kompilačnímu módu JIT Flutter poskytuje nesmírně užitečnou funkci *hot-reload*. Tato funkce umožňuje zobrazit jakékoli změny provedené v kódu téměř v reálném čase, bez nutnosti restartování aplikace. Aktualizovaný zdrojový kód se vloží do spuštěné aplikace, Flutter automaticky znovu vytvoří strom widgetů a na to aktualizuje strom *RenderObjects*. *Hot-reload* dramaticky zrychluje vývoj aplikace, pomáhá rychle identifikovat chyby a otestovat nové uživatelské rozhraní.
- **Dokumentace** – Díky svým výhodám oproti mnoha jiným frameworkům upoutal Flutter pozornost mnoha vývojářů z celého světa, a tím se vytvořila aktivní komunita, která přispívá do vývoje projektu. Na internetu lze tak najít spoustu lekcí, dokumentů, návodů, včetně rozsáhlé oficiální online dokumentace přímo od tvůrců Flutteru.
- **Open-source** – Flutter je *open-source* framework, díky čemuž je vývojový nástroj vysoce flexibilní a umožňuje vývojářům neomezeně přizpůsobit framework dle svých vlastních potřeb.

4.5.2 Nevýhody

- **Mladá technologie** – Flutter je poměrně mladá technologie, která byla zveřejněna v roce 2017. Vzhledem k tomu, že Flutter je nový framework, pravidelně prochází změnami a aktualizacemi, které mohou ovlivnit vývoj — údržba kódu může být výzvou v tak rychle se měnícím prostředí.
- **Velikost souborů** – Jelikož Flutter má vlastní engine, který je společně s aplikací zkompilován do strojového kódu, mívají aplikace vyvíjene ve Flutteru vyšší datové nároky.
- **Nedostupnost knihoven** – Opět z důvodu, že Flutter je nová technologie, počet externích knihoven dostupné pro Flutter je výrazně nižší než u konkurenčních frameworků.

Kapitola 5
Návrh řešení

Poté co byly představeny uživatelské požadavky a analyzována vývojová technologie, je důležité vytvořit podrobný návrh aplikace, který zajistí kvalitu aplikace a zároveň zabrání zbytečným komplikacím během vývoje. Tématem této kapitoly bude popořadě právě návrh architektury, serverové části a uživatelského rozhraní.

5.1 Clean Architecture

Během posledních několika let se v komunitě vývojářů objevilo několik inovativních architektonických stylů, jako jsou *Hexagonal Architecture*, *Onion Architecture*, *Screaming Architecture*, *Data-Context-Interaction* nebo *Boundary-Control-Entity*. Každá z těchto architektur nabízí řadu výhod, které se snaží *Clean Architecture* (dále CA) zkombinovat do jedné ucelené filozofie. [18]

CA je sada pravidel softwarového návrhu, které představil Robert C. Martin v roce 2017 ve své stejnojmenné knize. Cílem architektury je poskytnout metodologii pro psaní kódu, který je jednoduše rozšiřitelný a udržovatelný, srozumitelný, minimalizuje náklady na vývoj a zároveň maximalizuje produktivitu [19, str. 135].

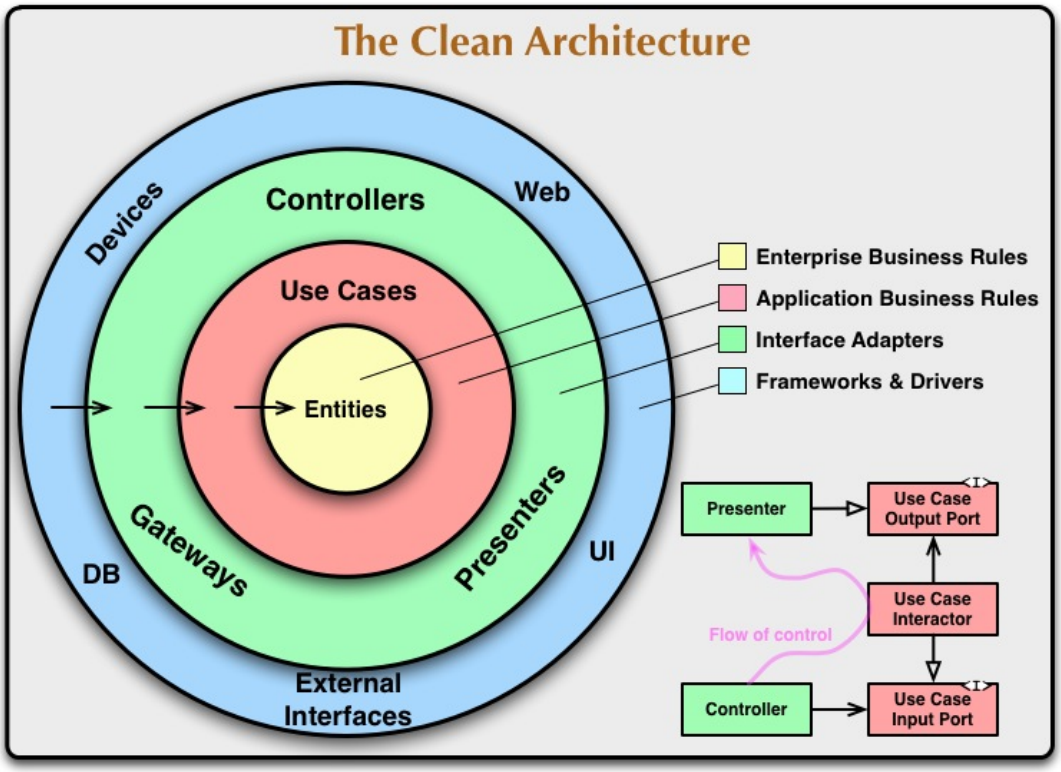
5.1.1 SOLID

Důležitou součástí CA je pět principů OOP návrhu nesoucí akronym SOLID, které definoval stejný autor. SOLID určují, jak by měly být funkce správně uspořádány a jak by třídy měly být vzájemně propojeny. Poskytují jasné a použitelné pokyny pro psaní čistého a udržovatelného kódu. Principy SOLID jsou následující:

- **Single-responsibility principle** říká, že třída by měla mít pouze jeden důvod ke změně. Podle Roberta C. Martina se softwarové systémy mění tak, aby uspokojily uživatele nebo zúčastněné strany. Modul by měl být tedy odpovědný pouze jednomu uživateli a zúčastněným stranám. Pokud stejná třída odpovídá na požadavky různých aktérů, měla by se rozdělit do dvou samostatných modulů.
- **Open-closed principle** určuje, že software by měl být vždy otevřený pro rozšíření, ale uzavřený pro modifikaci. To znamená, že softwarové systémy musí být snadno rozšiřitelné, ale musí být navrženy tak, aby ke změnám docházelo přidáním nového kódu, nikoli změnou stávajícího. Dobrá softwarová architektura by měla snížit množství změněného kódu na nejmenší minimum — v ideálním případě na nulu.
- **Liskov substitution principle** znamená, že objekty by měly být zaměnitelné za instance jejich podtypů, aniž by došlo k jakémukoliv ovlivnění programu.
- **Interface segregation principle** označuje použití rozhraní k oddělení závislosti třídy od ostatních tříd, které ji používají. Rozhraní vystavuje pouze podmnožinu metod, které závislá třída potřebuje. Tímto způsobem, když dojde ke změnám v jiných metodách, nebude ovlivněna závislá třída.
- **Dependency inversion principle** stanoví, že méně stabilní třídy a komponenty by měly záviset na stabilnějších, nikoli naopak. Pokud stabilní třída závisí na nestabilní třídě, pak pokaždé, když se nestabilní třída změní, ovlivní to také stabilní třídu. Směr závislosti je proto nutné převrátit. [20–21]

5.1.2 Vrstvy

Znáznornění CA podle představy Roberta C. Martina lze pozorovat na obrázku 5.1. Jednotlivé vrstvy nabírají směrem dovnitř míru abstrakce. V jádru architektury lze nalézt veškerou business logiku, u které nejsou změny tolik očekávané. Naopak vnější vrstvy obsahují implementační detaily a prvky směrem ven se stávají méně kritickými a náchylnějšími ke změnám, proto jsou zde umístěny databáze, frameworky, uživatelská rozhraní apod. Důležitým pravidlem CA je, že závislosti mohou směrovat pouze z vnějších úrovní dovnitř, přičemž vnitřní vrstvy nesmí mít žádné povědomí o vnějších. Tím lze zajistit oddělení odpovědností a jádro aplikace je zcela nezávislé na použitém frameworku, databázi či externích službách. [19, str. 203–204]



Obrázek 5.1. Clean Architecture podle Roberta C. Martina [18]

CA je velmi abstraktní koncept a způsobů, jak architekturu navrhnout, je více. Robert C. Martin však ve své knize popisuje čtyři základní vrstvy:

- *Entities* jsou sada souvisejících business pravidel, která jsou kritická pro fungování aplikace a jsou zcela nezávislé na jakékoliv jiné vrstvě v CA.
- *Use Cases* představují akce, které může uživatel v aplikaci provádět a jsou popisem, jakým způsobem by se měla aplikace chovat. *Use cases* definují aplikační logiku, na základě které se řídí interakce mezi uživateli a entitami.
- *Interface Adapters* je sada adaptérů, které převádějí data do vhodného formátu mezi *use cases* a vrstvou *frameworks and drivers*.
- *Frameworks and drivers* je nejsvrchnější vrstva CA, která je obykle tvořena z frameworků, databází, IO komponent apod. Jedná se o nejvíce volatilní vrstvu, u které jsou změny velmi pravděpodobné, proto je umístěna nejdál od jádra a nejsou na ní závislé žádné další vrstvy. [19, str. 204–206]

5.1.3 Výhody

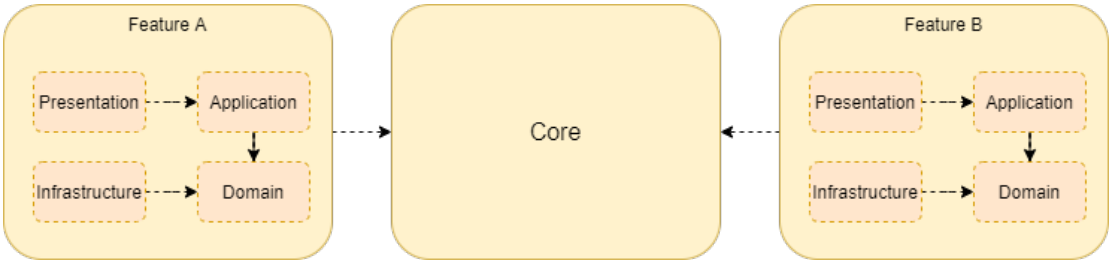
- **Rozšiřitelnost** – Veškeré implementační detaily jsou zcela izolované od aplikační a business logiky. Je tedy velice jednoduché vyměnit databázi, upravit uživatelské rozhraní či změnit framework bez zásahu do jádra aplikace.
- **Testovatelnost** – Aplikační a business logika má velice málo až žádné závislosti, a tak je lze testovat bez uživatelského rozhraní, databáze, webového serveru nebo jakéhokoliv jiného externího systému.
- **Udržitelnost** – Údržba je obvykle nejnákladnějším aspektem projektu vývoje softwaru. Přidávání nových funkcí a řešení chyb spotřebovává obrovské množství zdrojů. Díky pricipům SOLID a izolaci komponent pomocí stabilních rozhraní lze jednoduše provádět změny v kódu a výrazně snížit riziko neúmyslného zanesení chyb během vývoje.
- **Přehlednost** – Logika aplikace není rozptýlena na různých místech, nýbrž je zapouzdřena do odpovídajících vrstev, což usnadňuje ladění programu. Dále je aplikační logika realizována pomocí *use cases*, které přímo reflektují případy užití, a jsou tím velmi jednoduché na pochopení.

5.1.4 Nevýhody

- **Velikost kódu** – Hlavní nevýhodou CA je větší, místy i redudantní, kód. Z důvodů vysoké modularizace a principům SOLID, je i pro jednoduché CRUD operace nutné napsat více kódu než obvykle.
- **Komplexita** – Pro aplikace menšího rozsahu je oddělení do více vrstev a dodržování všech principů spíše zátěží a přináší do vývoje zbytečnou komplexitu navíc.

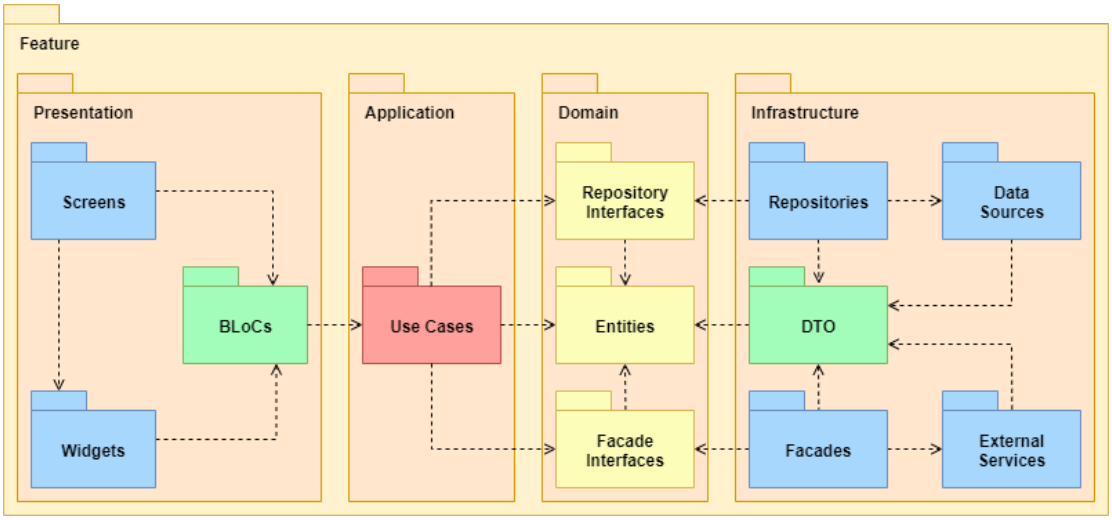
5.1.5 Vlastní návrh

Clean Architecture pouze definuje sadu návrhových vzorů, postupů a zásad pro vytváření softwarové architektury a její konkrétní podoba by měla být vždy zvolena tak, aby co nejlépe reflektovala požadavky aplikace. Počet vrstev či jejich uspořádání není pevně stanovené, důležité je zejména dodržení pravidla závislostí a všech principů SOLID. [19, str. 205]



Obrázek 5.2. Schéma struktury aplikace

Návrh vlastní aplikace sestává ze dvou typů modulů: `Core` modul a `Feature` moduly. `Feature` moduly zapouzdřují spolu související funkcionality a obsahují velkou většinu implementace aplikace. Naproti tomu je v aplikaci speciální `Core` modul, který je v aplikaci pouze jeden a obsahuje znovupoužitelné prvky pro ostatní moduly zejména grafické elementy, pomocné třídy, konstanty, nastavení apod.



Obrázek 5.3. Diagram balíčků Feature modulu

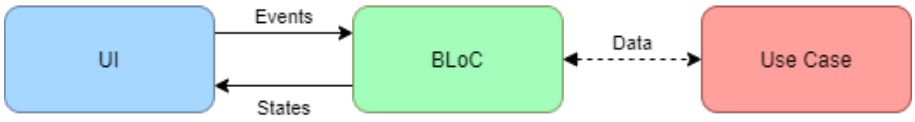
Jednotlivé `Feature` moduly (dále jako moduly) jsou strukturované v souladu s *Clean Architecture*. Oproti originální podobě CA je vlastní návrh obohacen o další vrstvy, což lze vidět na obrázku 5.3. Žluté balíčky by v originálním konceptu spadaly pod vrstvu *entities*, červené pod *use cases*, zelené pod *interface adapters*, modré pod *frameworks and drivers*. Dále, aby byly moduly dlouhodobě udržitelné a přehledné, jsou balíčky rozděleny do čtyř vrstev: prezentační, aplikační, doménová a infrastrukturní. I přes odlišného rozvržení a počtu vrstev od originálního návrhu CA, mezi vrstvami jsou zachovány pravidla závislostí, které mohou směřovat pouze směrem dovnitř.

Prezentační vrstva

Prezentační vrstva obsahuje prvky uživatelského rozhraní a slouží jako komunikační vrstva aplikace, která přímo interaguje s koncovým uživatelem. Hlavním účelem této vrstvy je přijímat uživatelské vstupy a následně zobrazovat výstupy.

- **Widgets** představují widgety technologie Flutter, které byly představeny v podkapitole 4.3.
- **Screens** obsahují obrazovky aplikace, které tvoří uživatelské rozhraní aplikace, a jsou pouze kompozicí widgetů z balíčku *Widgets*.
- **BLoCs** je akronym pro *Business Logic Components*. V sekci 4.3.2 byly představeny *StatefulWidgets*, které mohou obsahovat části aplikační logiky pro aktualizaci stavu. Toto však není ideální a pro lepší udržitelnost kódu je vhodné aplikační logiku oddělit od UI, k čemuž právě slouží návrhový vzor BLoCs.

  BLoC v sobě zapouzdřuje veškerou aplikační logiku pro aktualizaci stavu widgetů a jeho fungování je znázorněno na obrázku 5.4. BLoC přijímá události vyvolané uživateli, zpracuje požadavek, případně deleguje požadavek dál na *use case*, a uživatelům navrací odpověď v podobě nového stavu, který je následně využit k aktualizaci uživatelského rozhraní.



Obrázek 5.4. Návrhový vzor BLoC

Aplikační vrstva

Aplikační vrstva zapouzdřuje aplikační logiku, tedy veškerou funkcionalitu, kterou bude aplikace nabízet.

- **Use Cases** jsou implementace případů užití, které definují aplikační logiku.

Doménová vrstva

Doménová vrstva tvoří jádro aplikace a obsahuje prvky kritické pro fungování celého systému. Definuje rozhraní a modely, na kterých jsou postaveny ostatní vrstvy aplikace.

- **Entities** jsou implementace entit z analytického doménového modelu, který byl popsán v podkapitole 3.6.
- **Repository Interfaces** definují rozhraní pro *repositories* v infrastrukturní vrstvě.
- **Facade Interfaces** definují rozhraní pro *facade* v infrastrukturní vrstvě.

Infrastrukturní vrstva

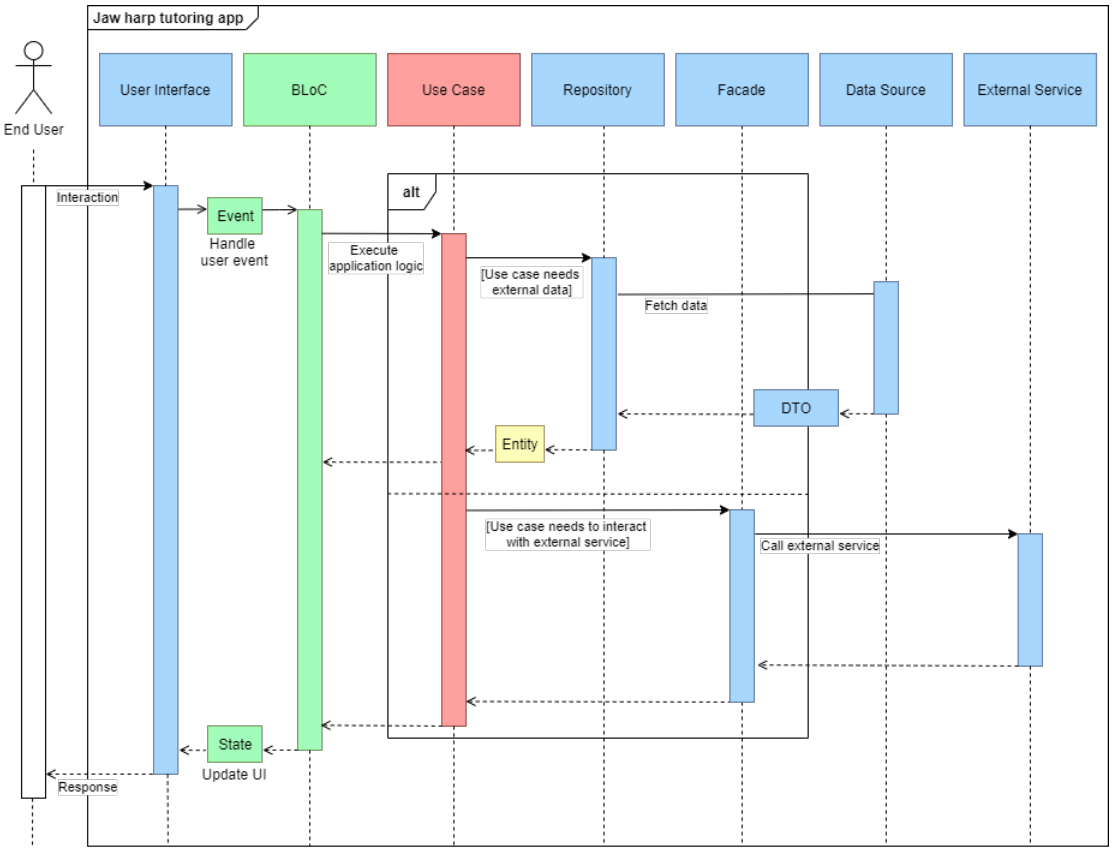
Infrastrukturní vrstva obsahuje implementace rozhraní doménové vrstvy a třídy, které umožňují komunikaci s externími systémy. Mohou se zde nacházet například implementace databázových spojení, externí webové služby, přístupy do souborového systému apod.

- **DTO** jsou potomky tříd z balíčku *Entities*. Jedná se o speciální objekty, které umí převádět data do vhodného formátu při komunikaci s externími systémy.
- **Repositories** slouží jako abstrakce nad přístupem k dat. *Repositories* se navenek tváří jako obyčejné kolekce podporující CRUD operace, interně však obsahují logiku pro komunikaci s více datovými zdroji zároveň.
- **Facades** plní velmi podobnou roli jako *Repositories*. *Facades* však obsluhují ostatní požadavky, které neobnáší datové operace.
- **Data Sources** jsou třídy umožňující komunikaci s datovými zdroji.
- **External Services** jsou třídy, které interagují s externími službami.

5.1.6 Interakce komponent

Jelikož návrh architektury obsahuje poměrně dost konceptů, je pro lepší pochopení vhodné interakci všech komponent namodelovat pomocí sekvenčního diagramu.

Hlavním cílem sekvenčních diagramů je zachytit komunikaci a posloupnost interakcí mezi částmi systému. Díky tomu lze velmi názorně popsat jakou roli v systému hrají jednotlivé komponenty a zároveň také v jakém pořadí a kdy se veškeré akce provádějí. [12, str. 109]



Obrázek 5.5. Sekvenční diagram

Komunikaci všech navržených balíčků lze pozorovat na výše uvedeném obrázku 5.5. Interakci se systémem zahajuje uživatel, který pracuje s uživatelským rozhraním. Pokud je nutné z jakéhokoliv důvodu aktualizovat uživatelské rozhraní, je v vytvořen `Event` objekt, který je zaslán do příslušné instance `BLoC` pro další zpracování. V `BLoC` se nachází pouze logika pro překreslení uživatelského rozhraní a je-li potřeba vykonat další aplikační logiku, požadavek je dál delegován na `Use Case` objekt. Dost často je při běhu `Use case` nutné načíst externí data či komunikovat se vzdálenými službami, k čemuž slouží popořadě `Repository` a `Facade`. Obě tyto komponenty zapouzdřují logiku pro interakci s více externími systémy, jež nejsou součástí aplikace. V případech, kdy je důvodem interakce s externími systémy právě získání dat, jsou navrácená nezpracovaná data převedeny do vhodného formátu pomocí `DTO` objektů, které poskytují funkce pro konverzi dat a zároveň jsou potomky `Entity` tříd. Zpět do instance `Use Case` jsou proto nakonec navraceny externí data, které se tváří jako obyčejné `Entity` objekty. Celý tento proces je zakončen v instanci `BLoC` vytvořením nového `State` objektu, který se následně využije pro aktualizaci uživatelského rozhraní.

5.2 Firebase backend

Firebase je platforma pro vývoj aplikací od společnosti Google, která poskytuje backendové služby, jako je vzdálená databáze, cloudové úložiště, autentizace, strojové učení, web hosting a další.

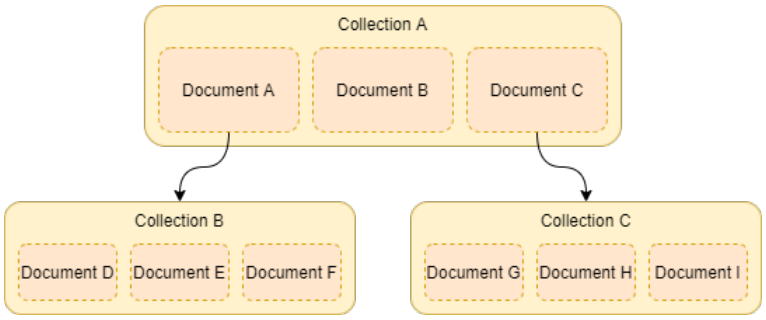
Platforma Firebase byla zvolena jako backendové řešení právě proto, že pokrývá mnoho funkčních požadavků aplikace a Flutter poskytuje knihovny, které velmi snadno umožní její integraci. Tato podkapitola se bude zabývat některými službami Firebase a využitím ve vyvíjené aplikaci.

5.2.1 Firebase autentizace

Většina aplikací potřebuje znát identitu uživatele, jelikož znalost identity uživatele umožňuje aplikaci bezpečně ukládat uživatelská data a poskytovat stejné osobní prostředí napříč zařízeními uživatele. Firebase Authentication poskytuje backendové služby, snadno použitelné SDK a knihovny uživatelského rozhraní pro ověřování uživatelů pomocí hesel, telefonních čísel, či poskytovatelů identit, jako jsou Google, Facebook, Twitter a další. [22] Firebase Authentication svými funkcemi pokrývá všechny funkční požadavky na autentizaci uživatelů, které byly představeny v podkapitole 3.3.

5.2.2 Cloud Firestore

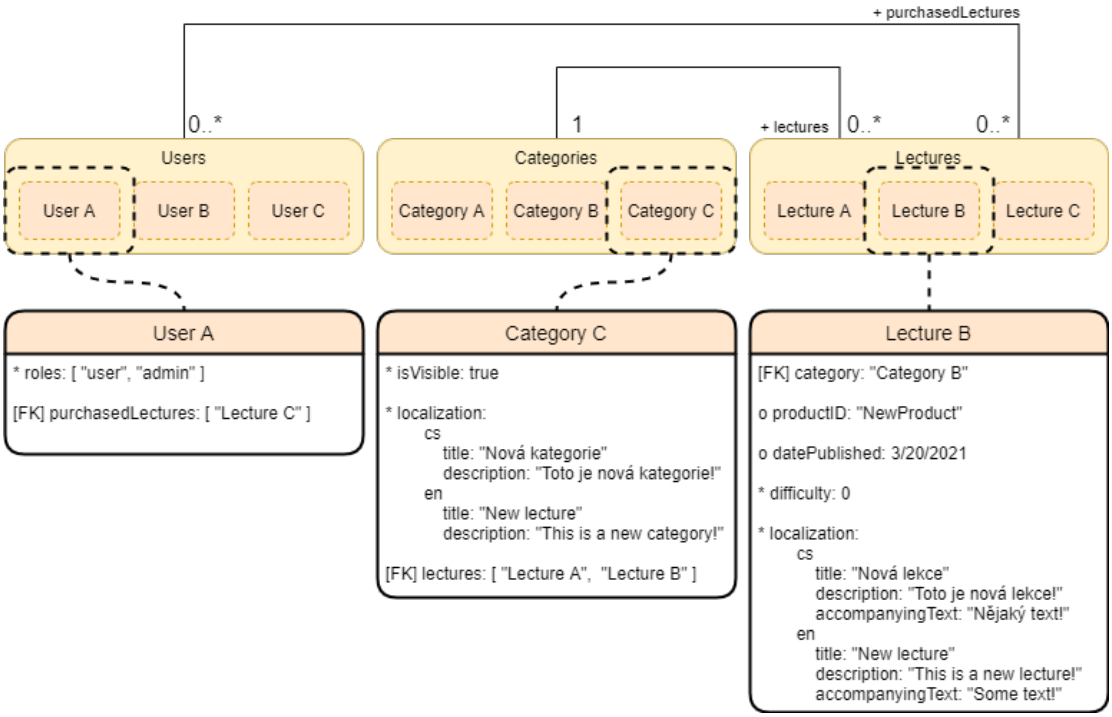
Cloud Firestore je flexibilní škálovatelná databáze pro vývoj mobilních, webových a serverových aplikací. Databáze v reálném čase udržuje data synchronizovaná mezi klientskými aplikacemi prostřednictvím *event listeners* a nabízí offline podporu pro mobilní zařízení a web, což umožňuje vytvářet responzivní aplikace, které fungují bez ohledu na latenci sítě nebo připojení k internetu. Cloud Firestore také umožňuje bezproblémovou integraci s dalšími produkty od Firebase a Google Cloud. [23]



Obrázek 5.6. Cloud Firestore databáze

Cloud Firestore je nerelační dokumentově orientovaná databáze, kde na rozdíl od databáze SQL neexistují žádné tabulky ani řádky. Místo toho se data ukládají do dokumentů, které jsou uspořádány do kolekcí. Znázornění této struktury lze vidět na obrázku 5.6.

Každý dokument ukládá data ve formě párů klíč–hodnota a zároveň může obsahovat další subkolekce. Všechny dokumenty musí být uloženy v nějaké kolekci a zároveň kolekce může obsahovat pouze dokumenty, což znamená, že kolekce nemůže přímo obsahovat data s hodnotami ani žádné jiné subkolekce. Cloud Firestore patří mezi *schemaless* databáze — všechny dokumenty v rámci jedné kolekce mohou obsahovat různá data a lze do nich ukládat různé typy dat. Dále jsou kolekce a dokumenty jsou vytvářeny implicitně. Jinými slovy, pokud kolekce nebo dokument neexistuje, vytvoří jej Cloud Firestore automaticky. [24]



Obrázek 5.7. Znázornění databázového schématu*

Návrh vlastního databázového schématu pro Cloud Firebase lze vidět na obrázku 5.7. Databázové schéma tvoří tři kolekce, které obsahují dokumenty uživatelů, kategorií a lekcí. Jelikož jsou názvy dokumentů z povahy Cloud Firestore vždy unikátní, jsou jako primární klíče dokumentů použity jejich názvy [24].

Dokumenty uživatelů obsahují pouze dva atributy a to seznamy uživatelských rolí a nakoupených lekcí. V porovnání s analytickým doménovým modelem v podkapitole 3.6 se na první pohled zdá, že u uživatelských dokumentů chybí atributy pro jméno, příjmení, email, `isVerified` apod. Tyto údaje však poskytuje služba Firebase Authentication, a proto není potřeba je ukládat znovu.

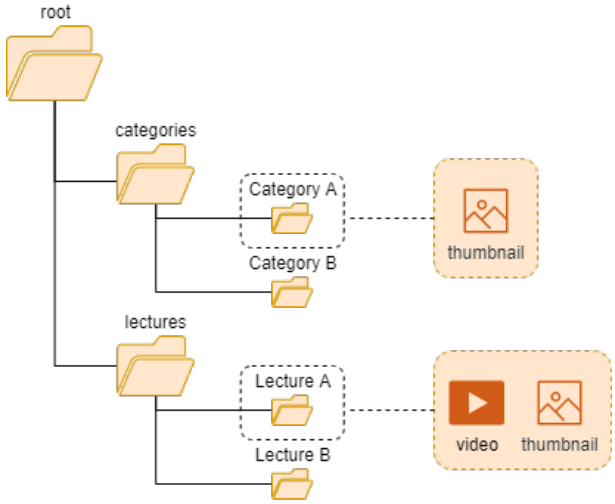
Dokumenty kategorií jsou pouze rozšířením entity kategorie v doménovém modelu. Jelikož mezi nefunkční požadavky patří lokalizace obsahu do více jazyků, je nutné název a popis ukládat ve více jazykových verzích v podobě asociativního pole, kde klíčem je kód země a hodnotou je lokalizovaný obsah. Příklad podoby konkrétního dokumentu kategorie lze vidět jako `Category C` na obrázku 5.7.

Dokumenty lekcí v porovnání s entitami v doménovém modelu se liší opět atributem pro lokalizaci a způsobem jak jsou ukládány informace pro realizaci plateb. Všechna data pro realizaci plateb jsou uložena na serverech cílových distribučních sítích, což bude více rozebráno v podkapitole 5.3. Tato skutečnost je v databázovém schématu reflektována absencí atributu `price` a novým atributem `productID`, který slouží ke spárování plateb.

* Jelikož standart UML nezná koncepty jako jsou dokumenty nebo kolekce, je pro znázornění databázového schématu použita vlastní notace.

5.2.3 Cloudové úložiště

Firebase Cloud Storage umožňuje ukládání souborů jakéhokoliv typu na vzdálené úložiště Google Cloud Storage do hierarchické struktury. K těmto datům lze později přistupovat z ostatních služeb Firebase či přímo z platformy Google Cloud. Firebase Cloud Storage je vytvořeno pro vývojáře aplikací, kteří potřebují ukládat a obsluhovat obsah generovaný uživateli, jako jsou například fotografie nebo videa. [25]

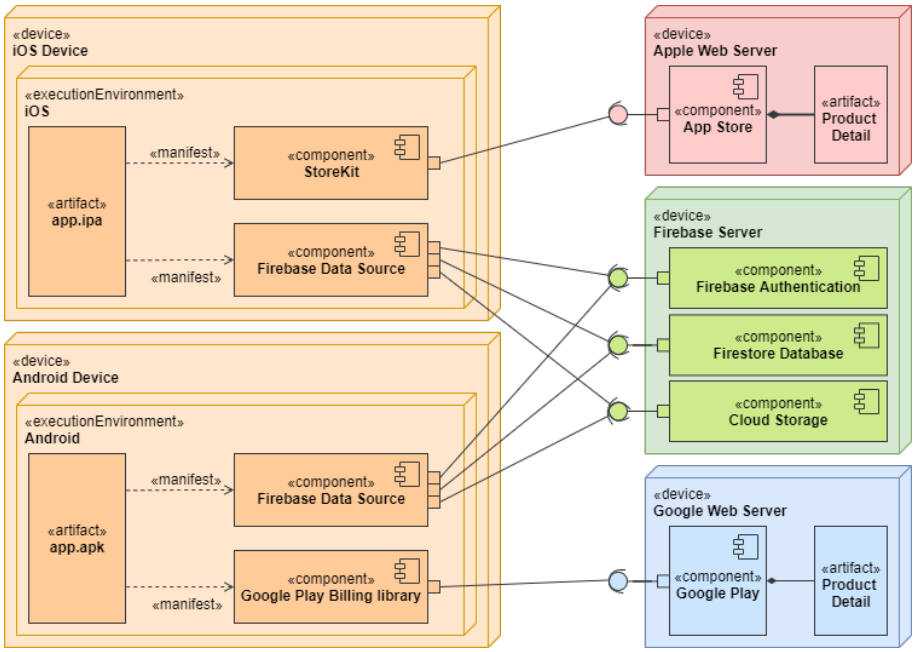


Obrázek 5.8. Schéma souborové struktury na Firebase Storage

Vzdálené úložiště od Firebase bude využito pro ukládání vizuálního obsahu lekcí a kategorií. Adresářová hierarchie je znázorněna na obrázku 5.8. Data jednotlivých lekcí a kategorií jsou uložena ve složce, která je pojmenovaná podle názvu příslušného dokumentu v databázi Cloud Firestore. Při požadavku na video či obrázek se pouze namapuje název dokumentu na název složky a zkontroluje se přítomnost souboru. Pokud je požadovaný soubor přítomen, Firebase Cloud Storage vygeneruje odkaz, pomocí kterého lze získat přístup k souboru.

5.3 Diagram nasazení

Diagram nasazení se běžně používá ke znázornění, jak jsou komponenty systému distribuovány po infrastruktuře a jak spolu komunikují. Hlavní položky v diagramu jsou `Nodes`, které jsou vzájemně propojeny komunikačními cestami. `Node` je obecně něco, co může hostovat software. K upřesnění role prvků v dané infrastruktuře lze použít přídavné notace zvané stereotypy, které rozšiřují a upřesňují význam. `Node` obohacený stereotypem `Device` představuje hardware, kterým může to být například počítač, mobilní zařízení či jednodušší hardware. Stereotyp `executionEnvironment` je software, který sám hostuje nebo obsahuje jiný software, příkladem mohou být operační systémy nebo kontejnerové procesy. Dále lze v diagramech nasazení nalézt také artefakty, které jsou fyzickými výstupy softwaru, což jsou obvykle spustitelné soubory nebo datové soubory, konfigurační soubory, dokumenty HTML apod. V diagramech nasazení se vykytují také komponenty, které představují modulární část systému, jejíž chování je definováno poskytnutými a požadovanými rozhraními. Artefakty bývají často implementací komponent, což je značeno vazbou se stereotypem `manifest`. [26, str. 78–79] Všechny tyto koncepty jsou využity při návrhu vlastního diagramu nasazení na obrázku 5.9, který popisuje infrastrukturu vyvíjené aplikace.



Obrázek 5.9. Diagram nasazení aplikace

Jak bylo již popsáno v kapitole 5.2, aplikace bude využívat backendové řešení Firebase a její služby Firebase Authentication, Firestore Database a Cloud Storage. Komunikace se servery Firebase bude probíhat z komponenty `Firebase Data Source` v infrastrukturní vrstvě aplikace pomocí oficiálního `Firebase SDK`.

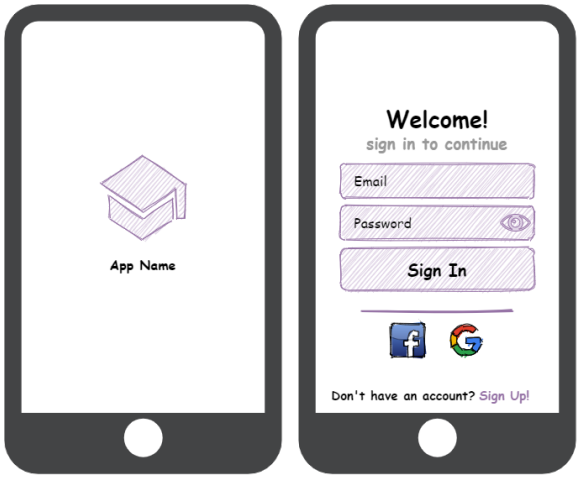
Kromě serveru Firebase, lze vidět v diagramu servery distribučních sítí pro integraci plateb. Jelikož smluvní podmínky Google Play a App Store vyžadují integraci jejich vlastních platebních systémů, je nutné platby pro každou mobilní platformu implementovat zvlášť [27–28]. Ceny, názvy a popisy produktů lze získat přes veřejné API, ke kterým lze přistoupit pomocí knihoven pro integraci plateb, jmenovitě `StoreKit` pro App Store a `Google Play Billing library` pro Google Play.

5.4 Uživatelské rozhraní

Jak již bylo rozebráno v kapitole 2 zabývající se rešerší současných řešení, uživatelské rozhraní je druhým nejdůležitějším aspektem výukové aplikace po jejím věcném obsahu. Proto je důležité si před implementací rozmyslet, jak se budou jednotlivé obrazovky chovat, a provést návrh uživatelského rozhraní.

5.4.1 Autentizace

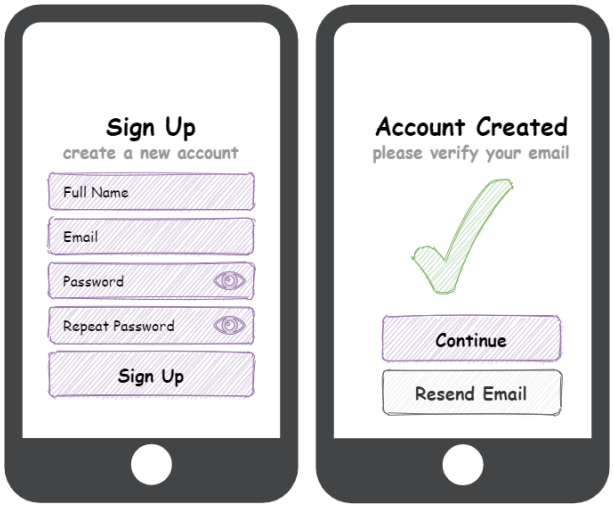
Autentizace je první krok, který uživatel musí provést při používání aplikace, a proto musí být tato část navrhnuta tak, aby byla maximálně intuitivní a nedávala prostor pro možné chyby.



Obrázek 5.10. Wireframes obrazovek přihlašování

Po spuštění je potřeba provést inicializaci aplikace, vytvořit spojení s databází, načíst důležitá data apod. Uživatel by však neměl být ponechán bez zpětné vazby. Proto je při každém spuštění aplikace jako první zobrazena úvodní obrazovka, jejíž účelem je snížit úzkost uživatele při čekání, přičemž v pozadí probíhají důležité procesy. Obsahem úvodní obrazovky je logo aplikace a její název. Po načtení všech závislostí aplikace je uživatel přesměrován na přihlašovací obrazovku.

Při vstupu do přihlašovací obrazovky si uživatel vytváří o aplikaci první dojem. Aby uživatel nebyl odrazen od používání aplikace, měl by být proces přihlášení i registrace rychlý a jednoduchý. Design přihlašovací obrazovky je minimalistický, aby uživatele při přihlašování příliš nerozptyloval. Jak lze vidět na druhé obrazovce obrázku 5.10, autentizaci zajišťuje přihlašovací formulář a maximálně dvě další alternativní autentizační metody. K přihlášení uživatel musí do formuláře vyplnit email a heslo. U pole pro vyplnění hesla bude mít uživatel možnost odkrýt heslo, pro snížení chybovosti při zadávání údaje. Tlačítko pro potvrzení se bude nacházet přímo pod formulářem a bude mít stejnou šířku a velikost jako jednotlivá pole, aby bylo dostatečně viditelné. Alternativně se mohou přihlásit uživatelé pomocí poskytovatelů identit, jejichž loga budou oddělená tučnou čarou od formuláře. Alternativní přihlášení budou zajišťovat pouze platformy Facebook a Google, jelikož šance, že uživatel již někdy v minulosti tyto platformy využíval je poměrně vysoká. Zároveň není vhodné poskytovat více než tři možnosti pro přihlášení, aby uživatelé nebyly zbytečně rozptýleni velkým výběrem.



Obrázek 5.11. Wireframes obrazovek registrace

Pokud uživatel nemá účet a potřebuje se zaregistrovat, klikne na tučný text s nápisem registrace ve spodní části přihlašovací obrazovky a je dále přesměrován k registračnímu formuláři. Design registračního formuláře je identický jako u přihlašovacího. Je vhodné zachovat všechny formuláře vizuálně konzistentní a dodržovat pouze jeden design, aby aplikace nepůsobila chaoticky a nebyla matoucí. Při registraci je po uživatelích vyžadováno minimum osobních informací a to pouze celé jméno, email a heslo — uživatelé neradi poskytují své osobní informace a registrace je nesmí odradit od používání aplikace. Alternativě se mohou uživatelé registrovat tím, že se poprvé přihlásí pomocí poskytovatelů identit.

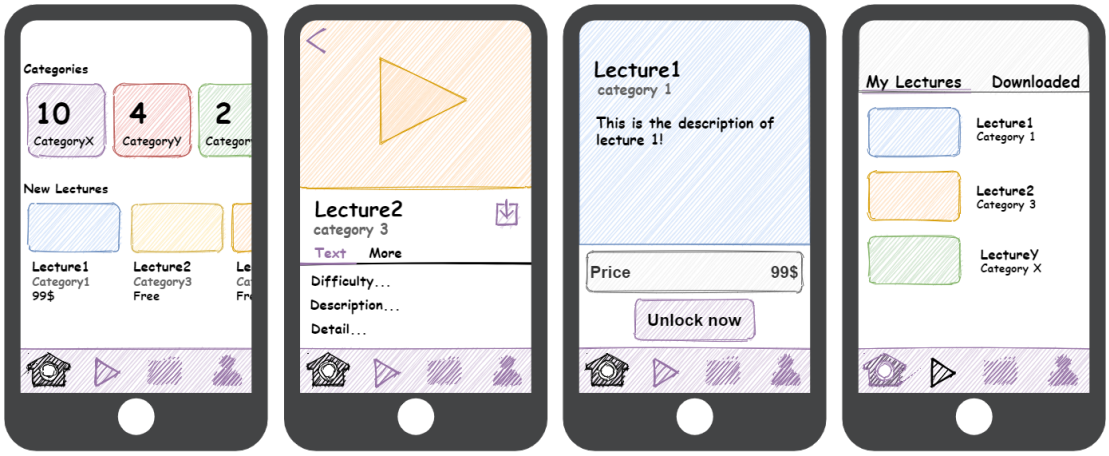
Po registraci jsou uživatelé přesměrování na ověřovací obrazovku. Ověřovací obrazovka obsahuje nadpis, který potvrzuje vytvoření účtu, a pod ním výzvu o ověření emailové adresy. Ve spodní části obrazovky je tlačítko pro pokračování, které ověří, zda-li byla potvrzena emailová adresa, a následně uživatele přesměruje do uživatelské části. Někdy se však může stát, že email z nějakého důvodu nedorazí a je potřeba ověřovací zprávu zaslat znovu — ověřovací obrazovka tedy obsahuje ve spodní části obrazovky tlačítko pro opětovné zaslání zprávy.

5.4.2 Uživatelská část

Uživatelská část tvoří největší část vyvíjené aplikace a její návrh je nejvíce komplexní. Zde se právě velice hodí poznatky z rešerše podobných aplikací a při návrhu budou využity designové principy, které se konkurenci osvědčili.

Obrázek 5.12. Wireframe navigace v uživatelské části

Esenciální součástí každé aplikace je způsob navigace mezi obrazovkami. K navigaci se hodí navigační panel ve spodní části obrazovky, jelikož většina uživatelů interaguje s aplikací pomocí palce ruky. Do navigačního panelu jsou umístěny tlačítka nejdůležitějších akcí: domovská obrazovka, uživatelský obsah, seznam kategorií a uživatelský profil.



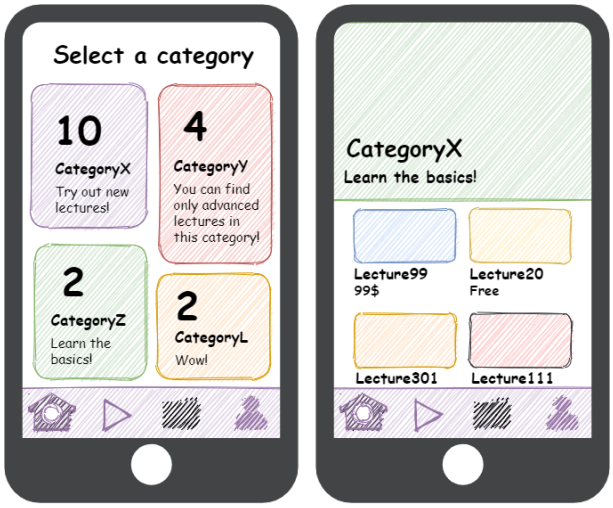
Obrázek 5.13. Wireframes obrazovek lekcí

Domovská obrazovka obsahuje seznamy kategorií a nových lekcí. Design domovské stránky lze vidět na první obrazovce obrázku 5.13. Oba seznamy jsou horizontálně posunovací, což šetří místem a dává prostor pro další možné budoucí rozšíření o nové seznamy. V seznamu kategorií mají jednotlivé kategorie v pozadí obrázek a v popředí název a počet lekcí dané kategorie. Pokud není dodán obrázek, je v pozadí zobrazena náhodná barva v odstínu celkového motivu aplikace. Obrázky a barvy pozadí jsou velmi důležité, jelikož pomáhají uživatelům se v seznamu zorientovat a rychleji najít požadovanou kategorii. Seznam lekcí vypadá obdobně, přičemž u každé lekce je zobrazen název, kategorie, cena a obrázek. Po kliknutí na lekci mohou nastat dvě situace: uživatel je přesměrován na detail lekce nebo na odemykací obrazovku.

Na detail lekce je uživatel přesměrován v případě, že je lekce zdarma nebo ji má uživatel zakoupenou. Návrh detailu lekce je znázorněn na druhé obrazovce v obrázku 5.13. Zhruba třetinu obrazovky tvoří výukové video a pod ním název s kategorií. Dále je vedle názvu umístěno tlačítko s ikonou pro stažení lekce. Je velmi pravděpodobné, že v budoucnosti bude detail lekce rozšířen o další funkcionality, jako jsou komentáře, chat s lektory či možnost psaní vlastních poznámek, a proto tvoří zbytek obrazovky menu se záložkami, pomocí kterého lze přepínat mezi dalším obsahem lekce. Menu zatím obsahuje pouze dvě záložky: doprovodný výukový text a další podobné lekce. V záložce s doprovodným výukovým textem lze nalézt obtížnost, krátký popisek a delší doprovodný text k videu lekce. Záložka s podobnými lekcemi zobrazí uživateli sloupcový seznam obsahující všechny lekce, které jsou zařazeny do stejné kategorie. Toto je velmi důležité, jelikož se může stát, že po shlédnutí lekce uživatelé nebudou vědět, jakou dále pokračovat. Seznam podobných lekcí má právě uživatelům ulehčit hledání nového obsahu, ušetřit čas a vést uživatele dál při učení.

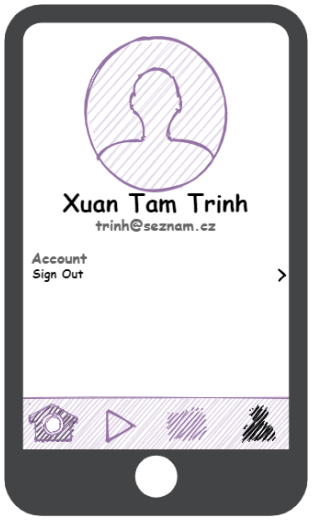
V případě, že lekce je zpoplatněná a zároveň ji uživatel nemá zakoupenou, aplikace zobrazí odemykací obrazovku. Odemykací obrazovka je znázorněna jako třetí v pořadí na obrázku 5.13. Horní polovinu obrazovky tvoří název, kategorie a krátký popisek, přičemž v pozadí je rozmazaný obrázek. Ve spodní polovině je tučným a velkým písmem zobrazena cena lekce a tlačítko pro nákup.

Všechny nakoupené a stažené lekce lze najít v části s uživatelským obsahem uživatele na čtvrté obrazovce v obrázku 5.13. Tato obrazovka sestává z menu se záložkami v podobném stylu jako u detailu lekce, a díky tomu ji lze velmi snadno v budoucnosti rozšířit o nové funkcionality.



Obrázek 5.14. Wireframes obrazovek kategorií

Další důležitou součástí uživatelské sekce je velký seznam kategorií, který lze vidět na první obrazovce obrázku 5.14. Seznam kategorií v domácí obrazovce je příliš malý a mohl by uživatele zdržovat při hledání požadované kategorie. Velký seznam kategorií je vertikálně posunutelný a zabírá celou obrazovku. Kategorie ve velkém seznamu obsahují navíc popisek, který stručně charakterizuje jejich obsah. Po otevření požadované kategorie je uživatel přesměrován do detailu kategorie. Detail kategorie je vidět na druhé obrazovce ve stejném obrázku. Horní část tvoří název a popisek s obrázkem v pozadí. Zbytek obrazovky tvoří vertikálně posunutelný seznam lekcí.

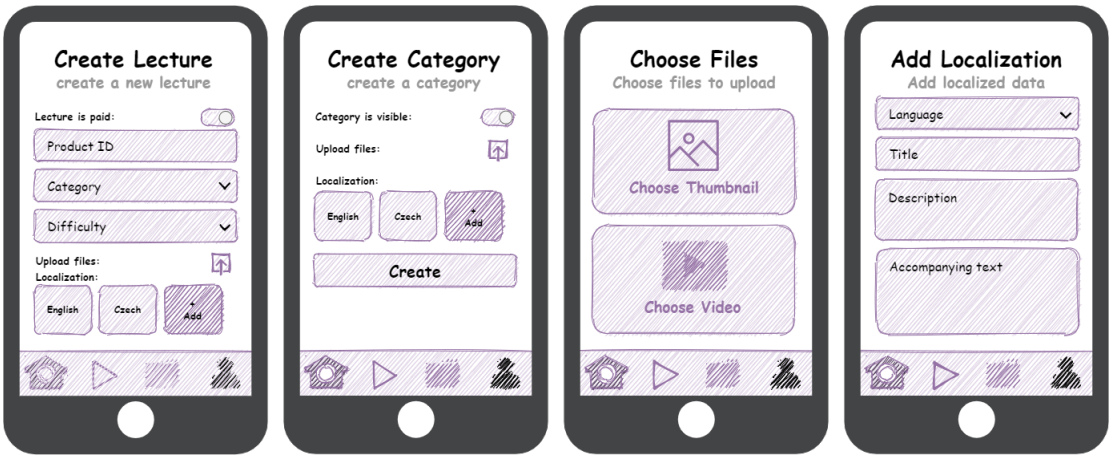


Obrázek 5.15. Wireframe obrazovky uživatelského profilu

Poslední součástí uživatelské sekce je uživatelský profil, který je zobrazen na obrázku 5.15. V této části se nachází veškerá nastavení aplikace a uživatelského účtu. Uživatelům s administrátorskými privilegii je zde navíc zobrazeno menu pro přístup do formulářů pro správu aplikace.

5.4.3 Administrátorská část

Návrh rozhraní administrátorské části nemusí být vizuálně estetický, nýbrž více prioritní je zařídit, aby správa obsahu aplikace byla co nejjednodušší a bezproblémová. Při návrhu obrazovek administrátorské části je tedy kladen důraz zejména na jednoduchost a praktičnost.



Obrázek 5.16. Wireframes obrazovek administrátorských formulářů

Administrátorské menu sestává z formulářů pro přidávání a editaci lekcí a kategorií. Formuláře pro editaci lekcí a kategorií jsou totožné s formuláři pro přidávání. Jediný rozdíl je v tom, že po otevření editačního formuláře dané lekce či kategorie budou do jednotlivých polí formuláře načtena příslušná data.

Na první obrazovce obrázku 5.16 lze vidět formulář pro vytvoření nové lekce. První položkou formuláře je přepínač, který určuje, zda-li je lekce zpoplatněná. Pokud je přepínač zapnutý, objeví se textové pole `ProductID`, které nastavuje identifikátor pro spárování plateb. V opačném případě bude pole `ProductID` skryté a lekce bude tím pádem zdarma. Dále formulář obsahuje rozbalovací seznam pro výběr kategorie a obtížnosti. Komplexnější data, které nelze jednoduše zadat do formuláře, jsou soubory a lokalizace obsahu. Pro tyto údaje jsou vyhrazeny samostatné obrazovky.

Pro nahrání obrázků a videí uživatelé mohou kliknout na tlačítko, které je přesměruje na speciální obrazovku pro nahrání souborů. Návrh této obrazovky lze vidět jako třetí v pořadí stejného obrázku. Na této obrazovce jsou vidět velké tlačítka s náhledem vybraného souboru. Při kliknutí na příslušné tlačítko je uživatel přesměrován do souborového systému pro výběr požadovaného souboru.

Dále se ve formuláři nachází horizontálně posunovací menu pro přidání a editaci lokalizovaného textu, ve kterém jsou tlačítka označené jazykem lokalizace. Pro přidání nového lokalizovaného textu uživatel je uživatel přesměrován na nový formulář, kde dále uživatel vybere jazyk lokalizace a vyplní povinná textová pole. Obrazovka pro přidání lokalizace je znázorněna jako poslední na obrázku 5.16.

Formulář pro přidávání kategorie je zobrazen na druhé obrazovce stejného obrázku. Formulář obsahuje přepínač symbolizující stav viditelnosti. Pokud je přepínač zapnutý, kategorie bude viditelná všem uživatelům aplikace. V opačném případě bude kategorie a její obsah skrytý, dokud se přepínač znovu nezapne. Dále formulář obsahuje tlačítka pro nahrání souborů a přidání lokalizovaného obsahu jako u formuláře vytvoření lekce.

Kapitola 6
Implementace

Poté, co byl vytvořen komplexní návrh aplikace, je na čase ho transformovat do zdrojového kódu. Část této kapitoly se věnuje představením vývojových nástrojů, které jsou využity při implementaci, následně jsou prozkoumány vybrané části aplikace, zhodnoceny výsledky a další možná rozšíření.

6.1 Vývojové nástroje

Předtím, než bude naplno zahájena implementace, je prvním krokem výběr vhodných nástrojů, které budou použity při této fázi. Výběr nástrojů je velice důležitý a může znamenat rozdíl mezi selháním a úspěchem projektu, neboť použitím správných nástrojů se drasticky zvyšuje produktivita, zlepšuje kvalita aplikace a hlavně se snižují náklady na vývoj.

6.1.1 Vývojové prostředí

Integrované vývojové prostředí (dále IDE) je software, který kombinuje více vývojářských nástrojů do jediného grafického rozhraní. IDE umožňují rychle začít programovat nové aplikace, neboť není potřeba ručně konfigurovat a integrovat najednou více nástrojů. Většina dnešních moderních IDE poskytují:

- Editor – Textový editor, který podporuje psaní zdrojového kódu funkcemi, jako je zvýraznění syntaxe, našeptávání a základní kontrola chyb.
- Automatizace sestavení – Nástroje pro automatizaci sestavení softwaru a souvisejících procesů včetně kompilace zdrojového kódu, vytváření balíčků a spuštění automatizovaných testů.
- Debugger – Program, který umožňuje během vývoje zkoumat vnitřní stav spuštěného software. [29]

K vývoji Flutter aplikací je oficiálně doporučeno Android Studio, které je postaveno na vývojovém prostředí IntelliJ IDEA od společnosti JetBrains, a je určeno speciálně pro vývoj mobilních aplikací. Android Studio poskytuje pluginy přímo od tvůrců Flutter, které umožňují využít funkci *hot-reload*, ladit vykreslování obrazovek, monitorovat výkon, prozkoumat strom widgetů, a další nástroje, které využívají plný potenciál technologie Flutter.

Jedním z nejužitečnějších nástrojů vývojového prostředí je integrovaný emulátor, který umí vytvářet, upravovat a spouštět virtuální Android zařízení. Tento nástroj umožňuje vyvíjené aplikace spouštět a testovat přímo ve vývojovém prostředí bez potřeby vlastnit fyzické zařízení, což je velmi praktické z časových i finančních důvodů. Naopak, pokud je potřeba aplikaci testovat na osobním zařízení, umožňuje Android Studio také připojit vlastní hardware a následně na něj nainstalovat vyvíjenou aplikaci k dalšímu testování.

6.1.2 Verzování

Verzování je soubor praktik a nástrojů, které slouží k sledování a správě změn výstupů softwarového vývoje. Účelem verzování je řídit změny a zároveň zlepšovat komunikaci a spolupráci v týmu. K verzování se velmi často používají systémy pro správu verzí (VCS), které slouží jako bezpečnostní mechanismus k ochraně výstupů před nenapravitelným poškozením a dávají tím vývojářům prostor k experimentaci. Verzovací systémy ukládají historii všech provedených změn, a díky tomu lze v případě potřeby navrátit změny do původního stavu, porovnávat různé verze či dohledat, kdo dané změny provedl. V dnešní době jsou verzovací systémy již nepostradatelným nástrojem při jakémkoliv vývoji software. [30]

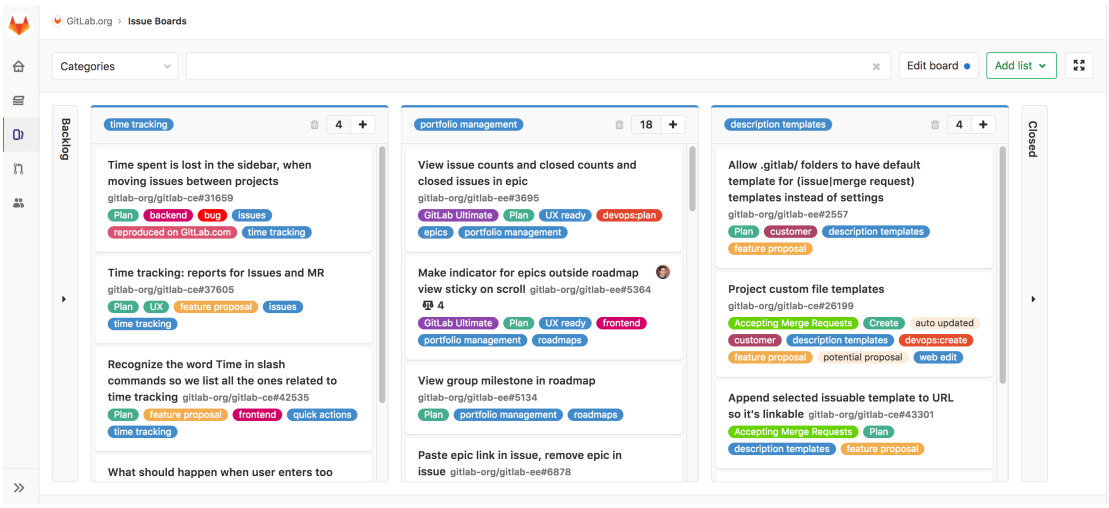
Pro verzování aplikace je zvolena webová platforma GitLab, která poskytuje nástroje pro podporu celého vývojového cyklu software. Mezi tyto nástroje patří právě gitový repozitář, který spadá pod distribuované VCS, což znamená, že historie změn se nachází jak na vzdáleném serveru, tak i lokálně jako kopie na uživatelském zařízení. Tato vlastnost přináší řadu výhod, mezi které patří například nezávislost vývoje na připojení k internetu.

GitLab kromě verzovacího systému nabízí další užitečné nástroje, které lze využít pro vývoj aplikace: *issue tracking*, *time tracking*, kontinuální integraci, průběžné doručování a další.

6.1.3 Správa úkolů

Správa úkolů obnáší aktivity, jako je plánování, odhadování a řízení nových cílů, které vznikají během softwarového vývoje. Jedná se velice důležitou činnost sloužící k určení uzávěrek, stanovení milníků a predikci časových i finančních nákladů, které ovlivňují, jaké funkcionality budou v aplikaci implementovány a jaké vynechány. Správa úkolů může probíhat různými způsoby od jednoduchých TODO listů až po specializované komplexní software.

Ke správě úkolů je zvolena funkce *issue tracking*, kterou zajišťuje platforma GitLab. Modul GitLab Issues sestává z jednoduchého seznamu úkolů, který lze libovolně modifikovat skrze webové uživatelské rozhraní. Všechny úkoly jsou následně zobrazeny na virtuální nástěnce, kde jsou vidět splněné, nesplněné a rozpracované úkoly. Jednotlivé úkoly lze označit vlastními štítky či přiřadit ke konkrétním milníkům, což dělá správu úkolů nesmírně jednoduchou a přehlednou.



Obrázek 6.1. Virtuální nástěnka pro správu úkolů platformy GitLab [31]

6.1.4 Kontinuální integrace

Kontinuální integrace (CI) je praktika softwarového vývoje, ve které jsou změny integrovány do sdíleného úložiště tak často, jak je to možné. Častá a pravidelná integrace snižuje počet konfliktních změn a pomáhá rychle odhalovat chyby při vývoji. Velmi často je kontinuální integrace doprovázena automatizací — každá nová integrace je ověřena automatickým sestavením, testy a validací předtím, než bude sloučena do sdíleného úložiště.

CI pomocí platformy GitLab funguje tak, že při vývoji jsou pravidelně integrovány menší úpravy do zdrojového kódu v GitLab repozitáři, přičemž při každé změně je spuštěn kanál skriptů, které sestaví, otestují a ověří změny předtím, než budou sloučeny do hlavní větve. Chování kontinuální integrace je možné v nakonfigurovat pomocí souboru s názvem `.gitlab-ci.yml`, který je umístěn v kořenovém adresáři úložiště. Tento soubor vytvoří *pipeline*, který se spustí při každé integraci. *Pipeline* se skládá z jedné nebo více fází, které běží v určeném pořadí a každá může obsahovat jednu nebo více úloh, které běží paralelně. Tyto úlohy (nebo také skripty) jsou prováděny programem GitLab Runner. [32]

Během implementace mobilní aplikace se při integraci nových změn budou spouštět automatizované testy a provede se statická analýza, což bude více probráno v kapitole 7, která je věnována testování. Podobu konfigurace souboru `.gitlab-ci.yml` lze vidět v níže přiloženém kódu:

```
# Docker image with Flutter installed.
image: cirrusci/flutter:2.0.1

# Get all Flutter dependencies before every script.
before_script:
  - flutter pub get

# The names and order of the pipeline stages.
stages:
  - test
  - analyze

# Run tests.
test:
  stage: test
  script:
    - flutter test

# Run static code analysis.
analyze:
  stage: analyze
  script:
    - flutter analyze
```

6.2 Ukázka implementace

V této podkapitole je předvedena ukázka vybrané funkcionality, na které budou vidět konkrétní příklady realizace *Clean Architecture* a zároveň také styl implementace aplikace. Jako vhodný demonstrační příklad je zvolena funkcionalita přihlášení uživatele pomocí e-mailové adresy a hesla, jelikož je poměrně snadno pochopitelná a zároveň demonstruje komunikaci s externími systémy. Postupně jsou probrány jednotlivé implementace od widgetů až po koncových datových zdrojů.

6.2.1 Dependency Injection

Předtím, než budou ukázány jednotlivé implementace CA, je nutné si představit důležitý koncept, na kterém je aplikace postavena — *dependecy injection*. Jedná se o návrhový vzor, který umožňuje automatické získávání závislostí za běhu aplikace. Jinak řečeno, zodpovědnost instancování závislostí je přenechána frameworku, a tím lze snížit provázanost mezi jednotlivými komponentami a dodávat lépe udržovatelný a testovatelný kód.

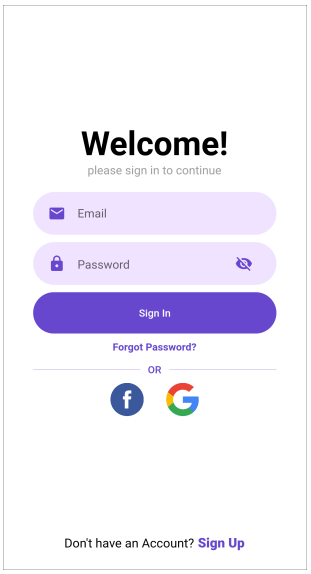
Dependency injection je implementována za pomocí Flutter balíčku `injectable`, který poskytuje speciální anotace `LazySingleton` a `Injectable`. V případě třídy označené anotací `LazySingleton` framework vytvoří jedinou instanci, která se bude vkládat do všech závisejících tříd, naopak s anotací `Injectable` se při vkládání závislostí vždy vytvoří nová instance. K jednotlivým anotacím lze také specifikovat parametr `env`, který umožňuje třídě přidělit jmenný prostor. Tímto způsobem lze mít více implementací jedné třídy pro různá prostředí, což je velmi užitečné pro testování aplikace.

6.2.2 Widgety

Jednotlivé obrazovky aplikace jsou pouze kompozicí widgetů, tudíž i samotná obrazovka je jeden komplexní widget. Implementace přihlašovací obrazovky je následující:

```
class AuthenticationScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      resizeToAvoidBottomInset: false,
      body: CenteredStack(
        children: [
          // Background widget
          AuthenticationScreenBackground(),
          // Authentication methods widget
          AuthMethods(),
          // Clickable sign up text widget
          Positioned(
            bottom: 20,
            child: SignUpText(),
          ),
        ],
      ),
    );
  }
}
```

Každá obrazovka aplikace začíná widgetem `Scaffold`, který vyplňuje celou obrazovku a poskytuje API pro zobrazení hlavních widgetů předané v parametru body. Jak lze vidět ve zdrojovém kódu, přihlašovací obrazovka je pouze kompozicí tří widgetů, které jsou poskládány přes sebe pomocí widgetu `CenteredStack`. Psaní kódu pomocí kompozice widgetů je velmi přehledné a význam tříd lze velmi často pochopit pouhým pohledem na zdrojový kód.



Obrázek 6.2. Vzhled přihlašovací obrazovky

Výsledkem výše uvedeného kódu je obrazovka, kterou lze pozorovat na obrázku 6.2. V dalších částí této podkapitoly bude věnována pozornost přihlašovacímu formuláři a všem souvisejícím akcím, které jsou nutné pro autentizaci uživatele pomocí e-mailové adresy.

6.2.3 BLoC

BLoC je návrhový vzor, pomocí kterého lze oddělit aplikační logiku pro aktualizaci stavu od widgetů. K implementaci návrhového vzoru je použit balíček `flutter_bloc`, který dělí BLoC do tří částí: `State`, `Event` a `Bloc`. Všechny části budou předvedeny na příkladu s přihlášením uživatele.

První částí návrhového vzoru je `State`, který obsahuje třídy signalizující různé stavy aplikace. V níže uvedeném zdrojovém kódu lze vidět, že formulář pro přihlášení pomocí e-mailové adresy obsahuje dva stavy:

1. `EmailAuthInitialState` – Počáteční stav, kdy uživatel není autentizovaný.
2. `EmailAuthSuccessState` – Stav signalizující úspěšné přihlášení, jehož součástí jsou uživatelská data zabalená do entity `User`.

```
class EmailAuthInitialState extends EmailAuthState {}

class EmailAuthSuccessState extends EmailAuthState {
  final User user;

  EmailAuthSuccessState(this.user);
}
```

Další částí je `Event`, kde lze nalézt různé třídy, jež reprezentují akce (události), které mohou nastat v rámci dané komponenty BLoC. V případě přihlašovacího formuláře existuje pouze jediná událost `EmailAuthenticationRequestEvent`, která nastane, když uživatel vyplní a odešle formulář ke zpracování.

```
class EmailAuthenticationRequestEvent extends EmailAuthEvent {
  final String email;
  final String password;

  EmailAuthenticationRequestEvent(this.email, this.password);
}
```

Nejdůležitější částí je třída `Bloc`, která si udržuje interní stav a poskytuje funkci `mapEventToState`, jež přijímá jako parametr událost `Event` a navrací nový stav `State`. Následně navrácený stav Flutter využije k aktualizaci UI. Pomocí právě této třídy mohou widgety vyvolávat události a dostávat odpověď ve formě nových stavů.

V implementaci pro přihlašovací formuláře lze vidět, že při vyvolání nové události `EmailAuthenticationRequestEvent` dochází k delegaci požadavku na *use case* objekt `emailAuthentication`, který bude rozebrán po této sekci. Při úspěšném přihlášení uživatele je stav třídy `Bloc` aktualizován na `EmailAuthSuccessState` a dojde k aktualizaci obrazovky. V opačném případě případě je vyvolána výjimka a signalizován chybový stav aplikace, který zobrazí dialogové okno s chybovou zprávou.

```
@Injectable(env: [Environment.prod, Environment.dev])
class EmailAuthBloc extends Bloc<EmailAuthEvent, EmailAuthState> {
  final EmailAuthentication emailAuthentication;
  final ErrorBloc errorBloc;

    EmailAuthBloc(
    this.emailAuthentication,
    this.errorBloc,
  ) : super(EmailAuthInitialState());

  @override
  Stream<EmailAuthState> mapEventToState(EmailAuthEvent event) async* {
    if (event is EmailAuthenticationRequestEvent) {
      try {
         final email = event.email;
         final password = event.password;
         final user = await emailAuthentication(email, password);
         yield EmailAuthSuccessState(user);
      } on BaseError catch (e) {
         final title = "Failed to sign in";
         final message = e.message;
         errorBloc.add(UserErrorEvent(title, message));
      }
    }
  }
}
```

6.2.4 Use Case

Use cases byly představeny v sekci 5.1.5 jako třídy, které implementují veškerou funkcionalitu aplikace. Všechny *use cases* obsahují metodu `call`, které umožňují instanci třídy používat jako obyčejnou funkci. Třída `EmailAuthentication` přijímá požadavky k přihlášení od `Bloc` komponenty a zkontroluje, zda-li byla jednotlivá pole formuláře vyplněna korektně. Jelikož je k ověření identity uživatele potřeba komunikovat s externím systémem, je požadavek dál předán třídě `IUserAuthRepository`, a při úspěšném přihlášení je zpět do `BLoC` komponenty navrácena uživatelská entita.

```
@LazySingleton(env: [Environment.prod, Environment.dev])
class EmailAuthentication {
  final IUserAuthRepository userRepository;

  EmailAuthentication(this.userRepository);

  Future<User> call(String email, String password) async {
    if (email.isEmpty || password.isEmpty)
       throw ValidationError("Please fill out all fields!");
    if (!RegExMatchers.email.hasMatch(email))
       throw ValidationError("Invalid email format!");
    if (!RegExMatchers.password.hasMatch(password))
       throw ValidationError("Invalid password format!");
    return userRepository.getUserWithEmailAndPassword(email, password);
  }
}
```

6.2.5 Entity

Implementace entity uživatele přímo vychází z analytického doménového modelu. Všechny entity jsou imutabilní a dědí ze třídy `Equatable`, která automaticky implementuje operaci porovnání na základě specifikovaných atributů v `props`, což je velice užitečné například pro srovnávání výsledků při běhu automatických testů.

```
class User extends Equatable {
  final String uid;
  final String name;
  final String email;
  final bool isVerified;
  final Set<String> purchasedLectures;

  const User({
    required this.uid,
    required this.name,
    required this.email,
    required this.isVerified,
    this.purchasedLectures = const {},
  });

  @override
  List<Object> get props => [this.uid, this.name, this.email];
}
```

6.2.6 Repository

Repository slouží jako abstrakce nad přístupem k datům, která může interagovat s více datovými zdroji zároveň. Implementace *repository* pro autentizaci obsahuje jediný datový zdroj `firebaseAuthDataSource`, který realizuje komunikaci se službou Firebase Authentication.

```
@LazySingleton(as: IUserAuthRepository, env: [Environment.prod])
class UserAuthRepository extends IUserAuthRepository {
  final FirebaseAuthDataSource firebaseAuthDataSource;

  UserAuthRepository(this.firebaseAuthDataSource);

  @override
  Future<User> getUserWithEmailAndPassword(String email, String pass) {
   return firebaseAuthDataSource.signInWithEmail(email, pass);
  }

  ...
}
```

6.2.7 Data Source

Data Sources jsou třídy, které implementují komunikaci s datovými zdroji. Interakce se službou Firebase Authentication je realizována pomocí balíčku `firebase_auth`, díky čemuž je interakce se službou velice přímočará a probíhá skrze poskytnutý objekt `FirebaseAuth.instance`. Pro přihlášení uživatele balíček vystavuje metodu `signInWithEmailAndPassword`, která v případě úspěchu navrací uživatelská data zapouzdřená do instance `UserCredential` a v případě neúspěchu vyhodí výjimku. Navrácená uživatelská data jsou však v nevhodném formátu, který je nutný převést do uživatelské entity pomocí DTO.

```
@LazySingleton(env: [Environment.prod])
class FirebaseAuthDataSource {
  final FirebaseAuth auth = FirebaseAuth.instance;

  Future<UserDTO> signInWithEmail(String email, String pass) async {
    try {
      final credential = await auth.signInWithEmailAndPassword(
         email: email,
         password: pass
      );

      return UserDTO.fromFirebaseUser(credential.user);
    } on FirebaseAuthException {
      throw WrongEmailOrPasswordError();
    }
  }

  ...
}
```

6.2.8 DTO

Všechny třídy DTO jsou potomky entit a obsahují pomocné metody pro překládání dat do vhodného formátu mezi *repositories* a datovými zdroji. V příkladu pro autentizaci uživatelů třída `UserDTO` dokáže vytvořit entitní objekt z dat navrácených službou Firebase Authentication.

K vytvoření `User` entity jsou však nutné informace o zakoupených lekcích, které jsou uložené na straně databáze Cloud Firestore. Komunikaci s databází lze realizovat za pomocí balíčku `cloud_firestore`, který ve velmi podobném stylu jako u balíčku autentizace vystavuje objekt `FirebaseFirestore.instance`. V uvedeném příkladu lze vidět, že pro získání dat o nakoupených lekcí je nejprve načtena z Cloud Firestore kolekce uživatelů a poté ze stejné kolekce je načten dokument konkrétního uživatele.

```
class UserDTO extends User {
  const UserDTO({
    required String uid,
    required String name,
    required String email,
    required bool isVerified,
    Set<String> purchasedLectures = const {},
  }) : super(
          uid: uid,
          name: name,
          email: email,
          isVerified: isVerified,
          purchasedLectures: purchasedLectures,
          profilePictureUrl: profilePictureUrl,
        );

  /// Translate raw Firebase Auth data into User entity.
  static Future<UserDTO> fromFirebaseUser(Firebase.User fbUser) async {

    // Load user document from Cloud Firestore
    final firestoreConnection = await FirebaseFirestore.instance;
    final collection = firestoreConnection.collection('users');
    final doc = await collection.doc(fbUser.uid).get();

    return UserDTO(
      uid: fbUser.uid,
      name: fbUser.displayName ?? "",
      email: fbUser.email!,
      isVerified: fbUser.emailVerified,
      purchasedLectures: Set<String>.from(doc["purchasedLectures"]);,
    );
  }

  ...
}
```

6.3 Vývojářská dokumentace

Vývojářská dokumentace byla vygenerována pomocí nástroje `Dartdoc`, který vytváří webovou dokumentaci na základě analýzy zdrojového kódu a komentářů. Dokumentace celého zdrojového kódu je dostupná v příloze D.

6.4 Závěr

Aplikace byla úspěšně implementována podle provedeného návrhu a pokrývá všechny nutné požadavky. Integrace platformy Firebase hrála důležitou roli při implementaci a drasticky urychlila vývoj aplikace. Výsledný vzhled lze vidět na obrázku 6.3, kde jsou zachyceny snímky všech důležitých obrazovek, a nahrané ukázky funkčnosti aplikace je možné najít v příloze D.

Pomocí integrace služby Firebase Authentication jsou pokryty všechny požadavky spojené s autentizací uživatelů. Do aplikace je možné se jednoduše zaregistrovat a pomocí Firebase poté zaslat ověřovací e-mail, který je lokalizovaný do jazyka zařízení. Přihlásit se je možné pomocí e-mailové adresy, facebookového profilu či účtu Google. Firebase umožňuje také resetování hesla, a tak byla tato funkce implementována nad rámec požadavků, jelikož se jedná o velmi běžnou funkcionalitu podporovanou konkurenčními aplikacemi.

Uživatelská část aplikace využívá výhody konkurenčních aplikací a poskytuje intuitivní uživatelské rozhraní, které bylo implementováno na základě návrhu v podkapitole 5.4. K zlepšení uživatelského komfortu aplikace využívá cachování obsahu pro rychlejší načítání, stahování lekcí do úložiště zařízení a ovládací prvky, které umožňují například změnit rychlost přehrávání videa. Dále je aplikace lokalizována do angličtiny i češtiny a veškerý textový obsah aplikace se automaticky přeloží na základě nastaveného jazyka hostitelského zařízení. V případě, že zařízení je nastaveno na nepodporovaný jazyk, aplikace je implicitně lokalizována do angličtiny.

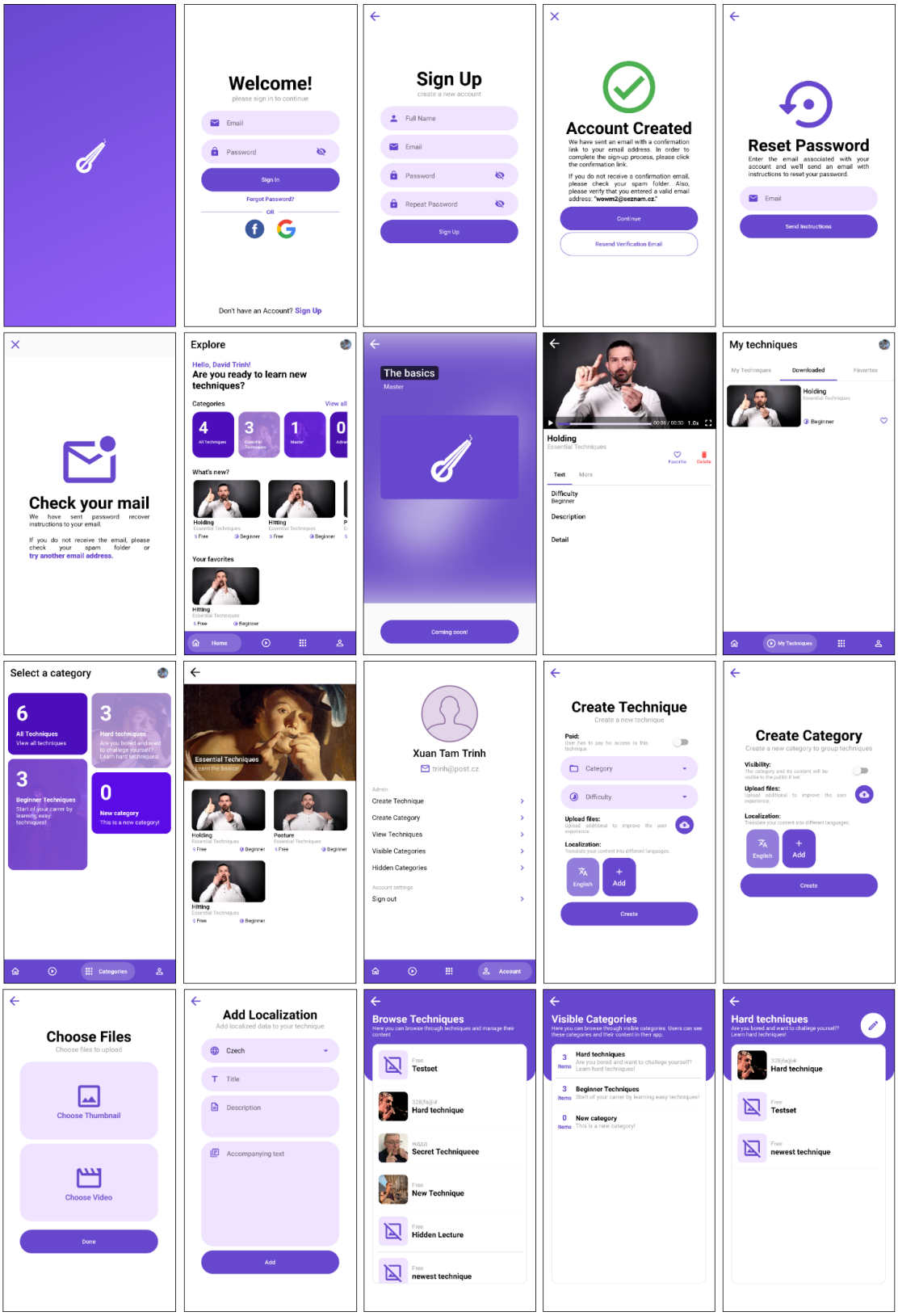
Přístup k administrátorské části lze získat na základě role uživatele v databázi Cloud Firestore. V profilové části se správcům zobrazí administrátorské menu umožňující vytvořit a editovat lekce s kategoriemi, zobrazit seznam všech lekcí a zobrazit seznam viditelných i skrytých kategorií. Nad rámec funkčních požadavků byla implementována viditelnost kategorií, která umožňuje správcům si nové lekce připravit do skrytých kategorií a poté jednoduše kategorii s lekcemi najednou zveřejnit.

6.4.1 Další možná rozšíření

Jasným možným rozšířením jsou funkční požadavky, které byly ve fázi analýzy v podkapitole 3.3 označeny prioritou *won't have*. Tedy jedná se o požadavky, které byly plně volitelné a lze je následků vynechat. Konkrétně se jedná o funkční požadavky na řazení a filtrování lekcí, které budou však velmi pravděpodobně implementovány v další verzi aplikace.

Dále aplikace neumožňuje streamování videí, a tak pro zobrazení lekce nutné nejprve načíst celé video. Hostování videí je sama o sobě velice datově náročná záležitost, a proto je vhodné v budoucích verzích implementovat streamování videí, aby byly co nejvíce sníženy náklady spojené s poskytováním aplikace.

6.4.2 Výsledný vzhled aplikace



Obrázek 6.3. Výsledný vzhled aplikace

Kapitola 7
Testování

Testování software je velmi důležité, jelikož během vývoje vnikají chyby, které je nutné před vydáním produkční verze identifikovat a včas opravit. Řádně otestovaný software zajišťuje kvalitu, vede k úspoře času, snížení nákladů a spokojenosti koncových uživatelů.

Tato kapitola se bude nejprve zabývat automatizovaným testováním pomocí statické analýzy a unit testů. A poté bude zbytek kapitoly věnován testováním použitelnosti, jehož cílem je ověřit použitelnost a celkovou funkčnost aplikace.

7.1 Statická analýza kódu

Statická analýza je způsob testování zdrojového kódu, který není závislý na běhu software. Jinak řečeno, statické testování probíhá čistě na úrovni zdrojového kódu bez potřeby spuštění programu. V dnešní době málokdy statická analýza probíhá manuální kontrolou kódu a naopak je často automatizována specializovaným software vyhledávající části kódu, které by mohly vést k chybám či bezpečnostním slabinám. Největší výhodou tohoto způsobu testování spočívá v tom, že ho lze zavést již v raných fázích softwarového vývoje a zároveň dokáže pokrýt celý rozsah zdrojového kódu.

Technologie Flutter poskytuje intergovaný nástroj `Dart Analyzer` pro realizaci statické analýzy kódu dle definovaných pravidel. Tento nástroj sestává ze dvou na sobě nezávislých modulů: `analyzer` a `linter`. Modul `analyzer` kontroluje správnost kódu a hledá chyby, jako jsou nedosažitelné části kódu, chybějící návratové hodnoty, nevyužité proměnné apod. Oproti tomu modul `linter` slouží ke kontrole stylu psaní kódu. Analýzu lze spustit pomocí příkazu `flutter analyze` a pomocí platformy GitLab ji lze provádět při každé integraci změny. Nastavení statické analýzy kódu je možné měnit pomocí souboru `analysis_options.yaml`, který leží v kořenovém adresáři projektu.

Při vývoji aplikace je použita taková konfigurace, která se snaží co nejvíce minimalizovat množství redudantního kódu a udržovat ho v přehledném stavu. Například pomocí pravidla `avoid_print` lze detekovat přítomnost zapomenutých print metod v produkčním kódu či podle pravidla `camel_case_types` lze dohlížet na pojmenování datových typů dle konvence *camel case*. Kromě zachování pořádku v kódu je možné pomocí statické analýzy také hledat potencionálně chybné části a nastavit jim závažnost. Při vývoji je například použito pravidlo `todo` se závažností `warning`, které při integraci nových změn vypíše varování v případě detekce TODO komentářů. Přísnější pravidla se závažností `error` při jejich porušení způsobí zastavení statické analýzy a nahlášení chybového stavu. Jedním z použitých pravidel se závažností `error` je například `missing_return`, které kontroluje absenci návratových hodnot.

Celkově, díky vysokému pokrytí, je statická analýza velice jednoduchým a mocným testovacím nástrojem, který při vývoji pomáhá detekovat jednoduché chyby a umožňuje zachovat kód konzistentní. Celou konfiguraci statické analýzy v souboru `analysis_options.yaml` lze pozorovat na následující stránce.

```
analyzer:
  errors:
    missing_required_param: error
    missing_return: error
    todo: warning
    unnecessary_null_comparison: warning

linter:
  rules:
    - always_declare_return_types
    - always_use_package_imports
    - annotate_overrides
    - avoid_annotating_with_dynamic
    - avoid_catching_errors
    - avoid_empty_else
    - avoid_init_to_null
    - avoid_print
    - avoid_relative_lib_imports
    - avoid_renaming_method_parameters
    - avoid_return_types_on_setters
    - avoid_unnecessary_containers
    - avoid_unused_constructor_parameters
    - await_only_futures
    - camel_case_extensions
    - camel_case_types
    - cancel_subscriptions
    - cast_nullable_to_non_nullable
    - close_sinks
    - empty_constructor_bodies
    - empty_statements
    - exhaustive_cases
    - hash_and_equals
    - unnecessary_await_in_return
    - unnecessary_brace_in_string_interps
    - unnecessary_const
    - unnecessary_getters_setters
    - unnecessary_new
    - unnecessary_null_aware_assignments
    - unnecessary_null_checks
    - unnecessary_null_in_if_null_operators
    - unnecessary_nullable_for_final_variable_declarations
    - unnecessary_overrides
    - unnecessary_parenthesis
    - unnecessary_statements
    - unnecessary_string_escapes
    - unnecessary_string_interpolations
    - unrelated_type_equality_checks
    - valid_regexps
    - void_checks
```

7.2 Unit testování

Unit testování je metoda, která ověřuje funkčnost nejmenších komponent systému nezávisle na ostatních. Podle tvůrce *Clean Architecture* Roberta C. Martina by každý správný unit test měl splňovat principy FIRST, které jsou následující:

- **Fast** – Unit testy by měly být rychlé. Pokud budou testy pomalé, nemohou se spouštět často, což má za následek, že v kódu se začnou akumulovat chyby.
- **Independent** – Všechny unit testy by měly na sebe navzájem nezávislé. V opačném případě selhání jednoho testu může způsobit selhání všech následujících.
- **Repeatable** – Unit testy by měly být opakovatelné ve všech prostředích bez rozdílných výsledků. Pokud bude toto pravidlo dodrženo, selhání testu znamená pouze špatnou funkčnost testované komponenty či nesprávně napsaný test a nic jiného.
- **Self-Validating** – U každého testu lze automaticky ověřit, zda-li proběhl úspěšně či nikoliv. Jinými slovy, k vyhodnocení testů nesmí být požadované žádné subjektivní rozhodnutí a výsledek každého testu musí být jednoznačně rozhodnutelný.
- **Timely** – Všechny unit testy je potřeba psát včas před vydáním produkční verze software. Pokud jsou testy psány až po vydání produkční verze, může se stát, že bude kód velice obtížně testovatelný. [33, str. 132–133]

K realizaci unit testování poskytuje Flutter balíčky `test` a `flutter_test`, které vystavují užitečné metody `test` a `expect`. Metoda `test` jako parametr přijímá název testu a funkci, ve které běží testy. Výsledky testu lze zkontrolovat pomocí metody `expect`, která přijímá jako první parametr očekávanou hodnotu a dále skutečnou hodnotu.

Někdy však existují mezi třídami závislosti, kvůli kterým nelze provést unit testy. K odstranění těchto závislostí během testování lze využít koncept zvaný mockování. Mockování je technika, která je založena na nahrazování závislostí speciálními mock objekty, které implementují rozhraní dané závislosti a zároveň je možné předem definovat chování pro konkrétní vstupy. Tímto způsobem je možné imitovat chování závislostí, a tak během testování odstranit pevné vazby mezi třídami. Balíček `mocktail` umožňuje velmi jednoduše implementovat mock objekty, které lze následně pomocí *dependency injection* vložit do závislých tříd.

Jelikož unit testů může být v projektu poměrně veliké množství, je vhodné zvolit jednotný formát strukturování a udržovat testy konzistentní. Proto je každý unit test navržen podle vzoru AAA, který dělí testy do tří fází:

1. **Arrange** – V této části testování jsou definovány všechny předpoklady a vstupy pro test, což zahrnuje inicializaci hodnot a nastavení mock objektů.
2. **Act** – Poté dochází k provedení všech akcí, které jsou předmětem testování.
3. **Assert** – V závěrečné části unit testování dochází k ověřování, že testované akce byly provedeny a chovaly se podle očekávání.

Hlavní výhodou tohoto modelu je, že vytváří jasné oddělení mezi nastavením testu, operacemi a zhodnocením výsledků, což usnadňuje čtení a porozumění kódu. [9, str. 204]

7.2.1 Ukázka unit testování

Všechny představené koncepty si lze ukázat na jednoduchém příkladu, ve kterém se testuje funkčnost *use case* třídy pro autentizaci uživatele představené v sekci 6.2.4. Následující ukázka testuje chování třídy `EmailAuthentication` při zadání korektní e-mailové adresy a hesla.

Nejprve je ve fázi *Arrange* nutné inicializovat všechny proměnné a definovat chování mock třídy `UserRepositoryMock`. Jak lze vidět v ukázce, pomocí metody `when` lze definovat chování mock objektu `userRepoMock` tak, aby při zadání e-mailové adresy `john.doe@gmail.com` a hesla `John123456` navracel předem definovanou entitu `mockResponse`. Poté dojde ve fázi *Act* k zavolání testované *use case* třídy `EmailAuthentication` a navrácená data jsou uloženy do proměnné `user`. V závěrečné fázi *Assert* je pomocí metody `expect` ověřeno, zda-li hodnoty navrácené entity v proměnné `user` jsou skutečně shodné s očekávanými hodnotami.

```
@LazySingleton(as: IUserAuthRepository, env: [Environment.prod])
class UserRepositoryMock extends Mock implements IUserAuthRepository {}

void main() {

  // Run test
  test("Should return user data when correct credentials are given",
          () async {

        // ----------------- Arrange -----------------
        final uid = "0";
        final name = "John Doe";
        final email = "john.doe@gmail.com";
        final password = "John123456";

        // Get initialized instances with injected dependencies
        final emailAuth = testServiceLocator<EmailAuthentication>()
        final userRepoMock = testServiceLocator<IUserAuthRepository>();

        // Create a predefined response
        final mockResponse = User(uid: uid, name: name, email: email);

        // Mock repository
        when(() => userRepoMock
                  .getUserWithEmailAndPassword(email, password))
                  .thenAnswer((_) async => mockResponse));


        // ------------------ Act --------------------
        final user = await emailAuth(email, password);


        // ----------------- Assert -----------------
        expect(user.email, email);
        expect(user.name, name);
        expect(user.uid, uid);
      }
  );

  ...
}
```

7.2.2 Pokrytí testů

Jak již bylo předtím zmíněno v sekci 6.1.4 o kontinuální integraci, unit testy se spouštějí při každé integraci nového kódu, čímž snižují riziko zanesení chyb během vývoje a zároveň ověřují funkčnost jednotlivých komponent. Testy pokrývají všechny prvky aplikační a infrastrukturní vrstvy, přičemž vynechány jsou balíčky doménové a prezentační vrstvy. Doménová vrstva je z testování vynechána, neboť se jedná pouze o sadu entit a rozhraní, jejichž testování nemá příliš velký význam. Co se týče prezentační vrstvy, její testování pomocí unit testů je možné, avšak unit testy jsou čistě mechanické a nedokáží odhalit chyby návrhu uživatelského rozhraní.

Z toho důvodu je lepší prezentační vrstvu validovat pomocí metod, které jsou založené na lidském faktoru. Uživatelské rozhraní je tedy během vývoje validováno formou *exploratory testing*, která spočívá ve volném průzkumu uživatelského rozhraní a jeho manuálním testování. Tato metoda je velice efektivní díky funkci *hot-reload*, která umožňuje během testování paralelně opravovat nalezené defekty.

Jelikož *exploratory testing* je prováděno pouze účastníky vývoje, je nutné prezentační vrstvu otestovat také i na osobách mimo vývojářský tým, a tím získat nezkreslené zhodnocení uživatelského rozhraní. Toto bude realizováno pomocí testování metody použitelnosti, čímž se bude zabývat následující podkapitola.

7.3 Testování použitelnosti

Celkovou funkčnost a jednoduchost použití aplikace lze ověřit pomocí testování použitelnosti, které je založeno na pozorování uživatelů pokoušejících se dokončit připravené scénáře. Na základě testování použitelnosti lze posoudit, jak dobře je výsledná aplikace využitelná a do jaké míry budou koncoví zákazníci spokojeni.

7.3.1 Průběh testování

Testování proběhne na menším okruhu lidí, z nichž malá část bude mít zkušenosti s výukovými aplikacemi a zbytek nikoliv. Poté moderátor* rozdá všem osobám testovací scénáře s úkoly a nechá čas, aby všichni účastníci porozuměli úkolům. Moderátor následně připraví aplikaci, aby splňovala předpoklady testovacího scénáře, a následně si bude volat účastníky po jednom, kteří se pokusí o splnění úkolů. Během plnění úkolů moderátor zapisuje výsledky a do průběhu zasahuje pouze v nutných případech.

Všechny testovací scénáře lze najít v příloze C. Testování je celkově zaměřené na nejvíce běžné činnosti v uživatelské a administrátorské části. Formát jednotlivých testovacích scénářů je následující:

1. **Identifikátor** – Identifikační kód a název testovací scénáře.
2. **Kontext** – Při jaké situaci může dojít k testovacímu scénáři.
3. **Předpoklady** – Předpoklady, které musí moderátor před začátkem testování zařídit.
4. **Úspěch** – Jaký by měl být stav systému, aby byl test vyhodnocen jako úspěšný.
5. **Postup** – Posloupnost kroků, kterými se budou účastníci řídit.

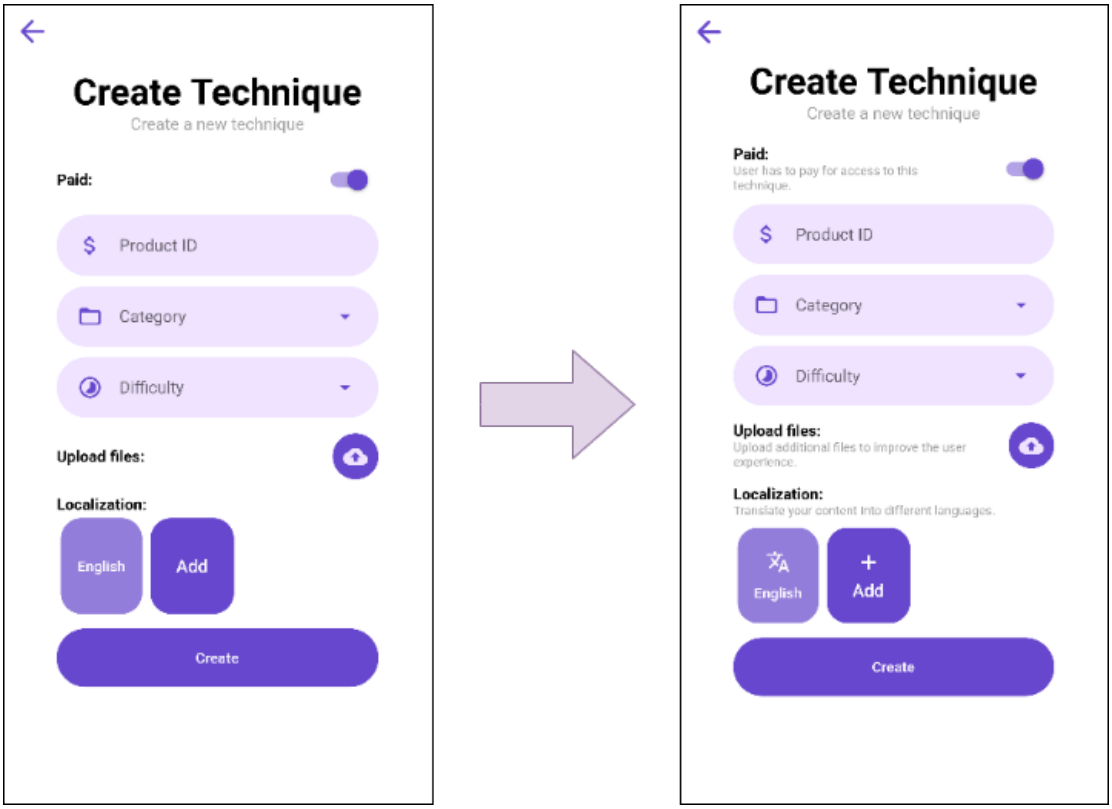
* Moderátorem se rozumí osoba, která řídí testování a zapisuje výsledky.

7.3.2 Výsledky testování

Testování proběhlo s pěti účastníky ve věkovém rozmezí 19–30 let, z nichž dva měli předchozí zkušenost s výukovými mobilními aplikacemi. Jeden z účastníků dokonce předtím používal mobilní aplikace TrueFire, která byla představena v sekci 2.2.1.

Účastník, který měl zkušenosti s aplikací TrueFire, se během testů orientoval v uživatelské části aplikace velmi rychle a bez jakékoliv pomoci, což není překvapivé, jelikož UI aplikace má velice podobný styl jako u TrueFire. Ostatní účastníci také zvládli testy uživatelské části bez větších problému, přičemž u prvních úkolů uživatelé potřebovali chvílí čas pro zorientování se v aplikaci a poté další úkoly již probíhaly hladce. Celkově při testování uživatelské části nebyly objeveny žádné zásadní chyby ve funkčnosti a účastníci hodnotili uživatelské rozhraní pozitivně.

Co se týče testů administrátorské části, někteří účastníci potřebovali při testování drobnou pomoc, neboť žádný z účastníků neměl s administrací jakékoliv zkušenosti. Proto na základě výsledků testování administrátorské části byly do formulářů k jednotlivým položkám přidány textové nápovědy, což se bude hodit k začlenění nových lektorů.



Obrázek 7.1. Změna formulářů po zhodnocení výsledků testování

Na závěr testování byly všichni účastníci poptáni, co by se dalo zlepšit na aplikaci. Jediné, co účastníkům nejvíce chybělo na aplikaci je více interaktivity. Tudíž v příštích verzích by mohla být aplikace obohacena o interaktivní cvičení či gamifikační prvky, které koncové uživatele při učení více zaměstnají.

Závěr

Cílem bakalářské práce bylo analyzovat, navrhnout a implementovat specializovanou výukovou mobilní aplikaci pro hru na brumle.

V úvodní části práce byla provedena analýza trhu mobilních aplikací s cílem identifikovat a analyzovat obdobná řešení, díky čemuž byla utvořena základní představa o mobilní výukové aplikaci, která posloužila jako podklad k analýze požadavků a návrhu aplikace. V další fázi byla provedena komplexní analýza funkčních a nefunkčních požadavků, která byla následně podpořena případy užití a analytickým doménovým modelem. Poté proběhla podrobná analýza technologie Flutter, která byla jedním z nefunkčních požadavků, a poté zhodnocení jejich silných a slabých stránek. Na základě analýzy požadavků byl proveden návrh architektury, serverové části a uživatelského rozhraní aplikace. Návrh vlastní architektury je založen na konceptu Clean Architecture, který zlepšuje zejména udržitelnost a rozšiřitelnost aplikace. Jako serverová část byla zvolena platforma Firebase poskytující backendové služby, které značně zrychlily vývoj a zároveň pokryly řadu funkčních i nefunkčních požadavů najednou. Tvorba uživatelské rozhraní byla inspirována konkurenčními aplikacemi, které nabízely velmi intuitivní a přívětivě UI. Celý návrh byl poté implementován a jako ukázka je v práci zvolena funkcionalita přihlášení uživatele, na které lze pochopit styl implementace a význam jednotlivých komponent. Výsledná implementace byla v průběhu vývoje automaticky testována pomocí statické analýzy, unit testů a na závěr proběhlo testování použitelnosti, které mělo ověřit celkovou funkčnost aplikace a jednoduchost použití. Na základě výsledků uživatelského testování byly provedeny změny v administrátorské sekci.

Všechny hlavní i dílčí cíle byly splněny a výsledkem této práce je funkční multiplatformní aplikace, která je připravená na nasazení do produkce. Do té doby jsou ve spolupráci se zadavatelem připravovány výukové lekce, které budou publikovány spolu s první verzí aplikace.

Literatura

[1] Manoharan, Raja. *Guide to Perform Competitive Analysis for your Next Mobile App Idea* [online]. 2. května 2018 [vid. 2021-03-30]. Dostupné na https://www.dotcominfoway.com/blog/how-to-perform-competitive-analysis-for-your-app-idea/amp/.

[2] TrueFire. *TrueFire Guitar Lessons* [software]. 1. března 2021 [vid. 2021-04-01]. Dostupné na https://apps.apple.com/us/app/truefire-guitar-lessons/id690143001.

[3] flowkey. *flowkey* [software]. 31. března 2021 [vid. 2021-04-01]. Dostupné na https://apps.apple.com/us/app/flowkey-learn-piano/id1020357408.

[4] Musora Media. *Drumeo* [software]. 7. května 2021 [vid. 2021-04-01]. Dostupné na https://apps.apple.com/us/app/drumeo/id1460388277.

[5] TrueFire. *TrueFire Guitar Lessons* [software]. 22. března 2021 [vid. 2021-04-01]. Dostupné na https://play.google.com/store/apps/details?id=com.truefire.android3.

[6] Musora Media. *Drumeo* [software]. 7. května 2021 [vid. 2021-04-01]. Dostupné na https://play.google.com/store/apps/details?id=com.drumeo.

[7] flowkey. *flowkey* [software]. 15. dubna 2021 [vid. 2021-04-01]. Dostupné na https://play.google.com/store/apps/details?id=com.flowkey.app.

[8] Sommerville, Ian a Pete Sawyer. *RE: a good practice guide*. John Wiley and Sons, 1997. ISBN 978-0-471-97444-4.

[9] Ingeno, Joseph. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018. ISBN 978-1788624060.

[10] Stephens, Rod. *Beginning software engineering*. John Wiley & Sons, 2015. ISBN 978-8126555376.

[11] Robichaud, Mitchel. *Functional vs Non-Functional Requirements: The Definitive Guide* [online]. 25. února 2021 [vid. 2021-04-07]. Dostupné na https://qracorp.com/functional-vs-non-functional-requirements/.

[12] Kim Hamilton, Russell Miles. *Learning UML 2.0*. O'Reilly, 2006. ISBN 978-0596009823.

[13] Larman, Craig. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012. ISBN 978-0131489066.

[14] Google Inc. *Flutter architectural overview* [online]. 13. dubna 2021 [vid. 2021-04-12]. Dostupné na https://flutter.dev/docs/resources/architectural-overview.

[15] Nikolaev, Sergey. *Flutter: a full introduction to the framework* [online]. 22. června 2020 [vid. 2021-04-13]. Dostupné na https://www.axon.dev/blog/flutter-a-full-introduction-to-the-framework.

[16] Google Inc. *Writing custom platform-specific code* [online]. 12. února 2021 [vid. 2021-04-17]. Dostupné na https://flutter.dev/docs/development/platform-integration/platform-channels.

[17] Schweiger, Frederik. *The Layer Cake* [online]. 21. prosince 2018 [vid. 2021-04-17]. Dostupné na https://medium.com/flutter-community/the-layer-cake-widgets-elements-renderobjects-7644c3142401.

[18] Martin, Robert C. *The Clean Architecture* [online]. 13. srpna 2012 [vid. 2021-04-03]. Dostupné na https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[19] Martin, Robert C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2018 . ISBN 978-0-13-449416-6.

[20] Sande, Jonathan. *CLEAN ARCHITECTURE FOR THE REST OF US* [online]. 4. ledna 2019 [vid. 2021-04-03]. Dostupné na https://pusher.com/tutorials/clean-architecture-introduction.

[21] Bixlabs Team. *An introduction to Clean Architecture* [online]. 15. února 2021 [vid. 2021-04-03]. Dostupné na https://www.blog.bixlabs.com/post/clean-architecture.

[22] Google Inc. *Firebase Authentication* [online]. 17. listopadu 2020 [vid. 2021-04-05]. Dostupné na https://firebase.google.com/docs/auth.

[23] Google Inc. *Cloud Firestore* [online]. 16. prosinec 2020 [vid. 2021-04-05]. Dostupné na https://firebase.google.com/docs/firestore.

[24] Google Inc. *Cloud Firestore Data model* [online]. 4. února 2021 [vid. 2021-04-05]. Dostupné na https://firebase.google.com/docs/firestore/data-model.

[25] Google Inc. *Cloud Storage for Firebase* [online]. 30. dubna 2021 [vid. 2021-04-05]. Dostupné na https://firebase.google.com/docs/storage.

[26] Fowler, Martin. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004. ISBN 978-0321193681.

[27] Apple Inc. *Play Console Help* [online]. [vid. 2021-04-06]. Dostupné na https://support.google.com/googleplay/android-developer/answer/9858738.

[28] Apple Inc. *App Store Review Guidelines: Business* [online]. 1. února 2021 [vid. 2021-04-06]. Dostupné na https://developer.apple.com/app-store/review/guidelines/.

[29] Red Hat Inc. *What is an IDE?* [online]. 8. ledna 2019 [vid. 2021-04-23]. Dostupné na https://www.redhat.com/en/topics/middleware/what-is-ide.

[30] GitLab Inc. *What is version control?* [online]. [vid. 2021-04-23]. Dostupné na https://about.gitlab.com/topics/version-control/.

[31] Wu, Victor. *4 ways to use GitLab Issue Boards* [online]. 2. srpna 2018 [vid. 2021-04-23]. Dostupné na https://about.gitlab.com/blog/2018/08/02/4-ways-to-use-gitlab-issue-boards/.

[32] GitLab Inc. *GitLab CI/CD docs* [online]. [vid. 2021-04-23]. Dostupné na https://docs.gitlab.com/ee/ci/.

[33] Martin, Robert C. *Clean code: A handbook of agile software craftsmanship*. Prentice Hall, 2009. ISBN 978-0-132-35088-4.

Příloha A
Seznam použitých zkratek

AOT	Ahead Of Time
API	Application Programming Interface
CA	Clean Architecture
CI	Continuous Integration
CRUD	Create, Read, Update, and Delete
DTO	Data Transfer Object
IDE	Integrated Development Environment
IO	Input Output
JIT	Just In Time
Mbps	Megabits per second
OOP	Object Oriented Programming
SDK	Software Development Kit
UI	User Interface
UX	User Experience
VM	Virtual Machine

Příloha B
Případy užití

B.1 Registrace

UC1	Registrace
Související požadavky:	FP1, FP2
Kontext:	Uživatelovi se zobrazí přihlašovací obrazovka, ale nemá účet.
Předpoklady:	žádné
Úspěch:	Uživatel má ověřený účet.
Neúspěch:	Byly zadány nevalidní údaje nebo účet byl již vytvořen.
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel klikne na tlačítko registrace. 2. Systém přesměruje uživatele k registračnímu formuláři. 3. Uživatel vyplní povinné údaje a zašle data dál ke zpracování. 4. Systém zvaliduje zaslané údaje, vytvoří nový účet a přesměruje uživatele na ověření účtu. 5. Uživatel stiskne tlačítko pro zaslání ověřovacího emailu. 6. Systém zašle ověřovací email s omezenou dobou platností. 7. Uživatel se přihlásí na svůj email a potvrdí ověřovací zprávu. 8. Systém změní status účtu na ověřený a přesměruje uživatele do uživatelské části.

B.2 Přihlášení

UC2	Přihlášení
Související požadavky:	FP3, FP4
Kontext:	Uživatel má účet a chce se přihlásit do aplikace.
Předpoklady:	Uživatel má účet s ověřenou emailovou adresou.
Úspěch:	Uživatel je přihlášen do aplikace.
Neúspěch:	Účet neexistuje nebo byly špatně vyplněny přihlašovací údaje.
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel spustí aplikaci. 2. Systém uživateli zobrazí přihlašovací obrazovku. 3. Uživatel vyplní přihlašovací údaje. 4. Na základě přihlašovacích údajů systém zkontroluje existenci účtu a následně přesměruje uživatele do uživatelské sekce.
Alternativní scénář:	1. Uživatel se spustí aplikaci. 2. Systém uživateli zobrazí přihlašovací obrazovku. 3. Uživatel klikne na logo Facebooku. 4. Systém přesměruje uživatele na přihlašovací stránku Facebooku. 5. Uživatel vyplní přihlašovací údaje a souhlasí s poskytnutím osobních údajů nutných k přihlášení. 6. Systém přesměruje uživatele do uživatelské sekce.

B.3 Odhlášení

UC3	Odhlášení
Související požadavky:	FP5
Kontext:	Uživatel se chce odhlásit z aplikace.
Předpoklady:	Uživatel je přihlášen do aplikace.
Úspěch:	Uživateli je odhlášen z aplikace.
Neúspěch:	žádný
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel klikne na tlačítko odhlášení v uživatelské sekci. 2. Systém odhlásí uživatele a přesměruje ho následně na přihlašovací obrazovku.

B.4 Zobrazení lekce

UC4	Zobrazení lekce
Související požadavky:	FP6, FP8
Kontext:	Uživatel chce zobrazit obsah lekce.
Předpoklady:	Uživatel se nachází v seznamu lekcí.
Úspěch:	Uživatel je přesměrován na detail lekce.
Neúspěch:	Uživatel odmítne zaplatit požadovanou částku.
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel klikne na požadovanou lekci. 2. Systém ověří, zda-li má uživatel oprávnění otevřít lekci, a přesměruje uživatele na obrazovku s obsahem lekce.
Alternativní scénář:	1. Uživatel klikne na požadovanou lekci. 2. Systém ověří, zda-li má uživatel oprávnění otevřít lekci, a přesměruje uživatele na odemykací obrazovku s náhledem lekce a její cenou. 3. Uživatel klikne na tlačítko pro odemknutí lekce. 4. Systém požádá uživatele o zaplacení stanovené finanční částky. 5. Uživatel zaplatí požadovanou částku. 6. Systém přesměruje uživatele na obrazovku s obsahem lekce.

B.5 Zobrazení kategorie

UC5	Zobrazení kategorie
Související požadavky:	FP7
Kontext:	Uživatel chce najít nové lekce na základě kategorií.
Předpoklady:	Uživatel je přihlášen do aplikace.
Úspěch:	Uživateli je zobrazen obsah kategorie.
Neúspěch:	žádný
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel klikne na tlačítko pro zobrazení kategorií. 2. Systém přesměruje uživatele na obrazovku se seznamem všech kategorií. 3. Uživatel vybere požadovanou kategorii. 4. Systém zobrazí uživateli seznam lekcí, které spadají do vybrané kategorie.

B.6 Stažení lekce

UC6	Stažení lekce
Související požadavky:	FP9
Kontext:	Uživatel chce stáhnout lekci do lokálního úložiště.
Předpoklady:	Uživatel se nachází v detailu lekce.
Úspěch:	Lekce je uložena v úložišti zařízení.
Neúspěch:	Lekci není možné uložit do zařízení kvůli nedostatečným oprávněním.
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel stiskne tlačítko pro stažení lekce. 2. Systém upozorní uživatele o procesu stahování a následně uloží lekci do paměti zařízení.

B.7 Změna uspořádání lekcí

UC7	Změna uspořádání lekcí
Související požadavky:	FP10, FP11
Kontext:	Uživatel chce změnit uspořádání lekcí pro snažší vyhledání.
Předpoklady:	Uživatel se nachází v seznamu lekcí.
Úspěch:	Seznam lekcí je seřazen dle zvolených kritérií.
Neúspěch:	žádný
Aktéři:	koncový uživatel
Hlavní scénář:	1. Uživatel klikne na tlačítko pro filtrování lekcí, následně nastaví pořadí a kritérium filtru. 2. Systém změní uspořádání seznamu dle zadaných parametrů.

B.8 Správa obsahu

UC8	Správa obsahu
Související požadavky:	FP12, FP13, FP14
Kontext:	Uživatel s administrátorským oprávněním chce měnit obsah aplikace.
Předpoklady:	Uživatel má administrátorské oprávnění.
Úspěch:	Obsah aplikace je aktualizován.
Neúspěch:	Uživatel nemá dostatečná oprávnění.
Aktéři:	Administrátor
Hlavní scénář:	1. Administrátor přejde do uživatelské sekce. 2. Systém zkontroluje, zda-li uživatel má dostatečná oprávnění, a zobrazí administrátorovi navíc administrátorské menu. 3. Administrátor zvolí příslušný formulář pro přidání či editaci lekce nebo kategorie, vyplní povinné údaje a uloží změny. 4. Systém uloží změny do databáze a změny se projeví při dalším spuštění.

Příloha C
Testovací scénáře

C.1 Registrace pomocí e-mailové adresy

TS1	Registrace pomocí e-mailové adresy	
Kontext:	Dostali jste chuť se naučit hrát na nový hudební nástroj a narazili jste na tuto aplikaci. K použití aplikace je však nutné mít ověřené účet, a tak se chcete zaregistrovat.	
Předpoklady:	Moderátor Vám připraví volnou e-mailovou adresu, pomocí které se můžete zaregistrovat.	
Úspěch:	Úspěšně jste se zaregistrovali a nacházíte se v domovské obrazovce.	
Postup:	1.	Zapněte aplikace a vyčkejte na načtení přihlašovací obrazovky.
	2.	Stiskněte text s nápisem registrace.
	3.	Vyplňte všechny povinné údaje. Na volnou e-mailovou adresu se můžete zeptat moderátora, pokud nechcete zadávat vlastní.
	4.	Přihlaste se na zadanou e-mailovou adresu a potvrďte ověřovací zprávu.
	5.	Vraťte se do aplikace a stiskněte pokračovat.

C.2 Přihlášení pomocí e-mailové adresy

TS2	Přihlášení pomocí e-mailové adresy
Kontext:	Jste vášnivý brumlista a výuková aplikace se Vám velmi líbila. Při posledním používání aplikace jste však byl odhlášen a chcete se znovu přihlásit.
Předpoklady:	Moderátor Vám připraví ověřený účet, pomocí kterého se lze do aplikace přihlásit.
Úspěch:	Úspěšně jste se přihlásili a nacházíte se v domovské obrazovce.
Postup:	1. Zapněte aplikace a vyčkejte na přihlašovací obrazovku. 2. Zeptejte se moderátora na přihlašovací údaje a zadejte je do přihlašovacího formuláře. 3. Potvrďte formulář.

C.3 Zobrazení vybrané lekce

TS3	Zobrazení vybrané lekce
Kontext:	Od svých přátel jste slyšel(a) o nové lekci, kterou chtete vyzkoušet, přičemž Vám byla prozrazen pouze její název a kategorie. Pokoušíte se danou lekci v aplikaci nalézt.
Předpoklady:	Moderátor připraví více kategorií a lekcí, mezi které patří i zadaná lekce a kategorie. Moderátor dále zařídí, aby jste začínal(a) v domovské stránce.
Úspěch:	Podařilo se Vám nalézt požadovanou lekci a nacházíte se v detailu lekce.
Postup:	1. Zeptejte se moderátora na název kategorie a pokuste se ji nalézt. 2. Otevřete danou kategorii. 3. Zeptejte se na název požadované lekce a pokuste se ji nalézt. 4. Stiskněte políčko požadované lekce.

C.4 Stažení lekce

TS4	Stažení lekce
Kontext:	Zjistili jste, že jedete na dovolenou, kde nebudete mít přípojení k internetu. Poslední lekci však nemáte dokončenou a chtěli byste ji dodělat, proto si ji stáhnete do úložiště zařízení pro offline zobrazení.
Předpoklady:	Moderátor zařídí, aby jste začínal(a) v detailu jakékoliv lekce.
Úspěch:	Podařilo se Vám stáhnout danou lekci a nacházíte se v obrazovce stažených lekcí.
Postup:	1. Stiskněte tlačítko s nápisem pro stažení. 2. Vyčkejte dokud nebudete upozorněni o dokončení stahování. 3. Přesuňte se do obrazovky s uživatelským obsahem a nalezněte staženou lekci.

C.5 Vytvoření nové lekce

TS5	Vytvoření nové lekce
Kontext:	Nyní jste v roli správce aplikace a dostal(a) jste nápad na novou lekci, kterou chcete přidat.
Předpoklady:	Moderátor zařídí, aby jste začínal(a) v uživatelském profilu, a máte dostatečná administrátorská oprávnění. Dále dostanete od moderátora předlohu nové lekce, kterou budete napodobovat. Všechny potřebné soubory se již nacházejí v zařízení.
Úspěch:	Vytvořená lekce je stejná jako předloha, což zkontroluje moderátor.
Postup:	1. Stiskněte v administrátorském menu tlačítko pro vytvoření lekce. 2. Podle předlohy vyplňte všechna textová pole. 3. Stiskněte tlačítko pro nahrání souborů. 4. Nahrajte obrázek a video dle předlohy, následně potvrďte volbu. 5. U položky lokalizace vyberte angličtinu, vyplňte dle předlohy a potvrďte změny. 6. Stiskněte na tlačítko pro přidání další lokalizace a podle předlohy vytvořte českou jazykovou verzi. 5. Uložte změny.

C.6 Editace lekce

TS6	Editace lekce
Kontext:	Nově byla vytvořena lekce, u které byly zjištěny chyby. Jako správce aplikace chcete chyby napravit.
Předpoklady:	Moderátor zařídí, aby jste začínal(a) v uživatelském profilu, a máte dostatečná administrátorská oprávnění. Dále dostanete od moderátora předlohu nové lekce, podle které budete danou lekci upravovat. Všechny potřebné soubory se již nacházejí v zařízení.
Úspěch:	Modifikovaná lekce je stejná jako předloha, což zkontroluje moderátor.
Postup:	1. Stiskněte v administrátorském menu tlačítko pro zobrazení seznamu všech lekcí. 2. Moderátor vám sdělí název lekce, kterou otevřete. 3. Vyplňte formulář dle předlohy. 4. Zkontrolujte, zda-li je obrázek a video stejné jako v předloze. 5. Zkontrolujte, zda-li je lokalizace stejná jako v předloze. 6. Uložte změny.

C.7 Vytvoření skryté kategorie

TS7	Vytvoření skryté kategorie	
Kontext:	Dostal(a) jste nápad na sadu nových lekcí, který zatím není tolik promyšlený. Prozatím chcete jen vytvořit jejich kategorii, která nebude uživatelům viditelná, a v ní připravovat nový obsah.	
Předpoklady:	Moderátor zařídí, aby jste začínal(a) v uživatelském profilu, a máte dostatečná administrátorská oprávnění. Dále dostanete od moderátora předlohu nové kategorie, kterou budete napodobovat. Všechny potřebné soubory se již nacházejí v zařízení. Dávejte pozor na viditelnost kategorie.	
Úspěch:	Nová kategorie je stejná jako předloha a zároveň je skrytá, což zkontroluje moderátor.	
Postup:	1.	Stiskněte v administrátorském menu tlačítko pro vytvoření nové lekce.
	2.	Ve formuláři vypněte přepínač viditelnosti.
	3.	Nahrajte obrázek a video dle předlohy, následně potvrďte volbu.
	4.	Přidejte lokalizovaný obsah dle předlohy.
	5.	Uložte změny.

C.8 Odhlášení

TS8	Odhlášení
Kontext:	Už je pozdě večer a čas jít spát. Z aplikace se proto chcete odhlásit.
Předpoklady:	Moderátor zařídí, aby jste začínal(a) v domovské stránce.
Úspěch:	Odhlášení proběhlo úspěšně a nacházíte se v přihlašovací obrazovce.
Postup:	1. Přesuňte se do uživatelského profilu. 2. Stiskněte tlačítko pro odhlášení.

Příloha D
Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `demo` ..... adresář s video ukázkami aplikace
  - `admin.mp4` ..... video ukázka administrátorské části
  - `auth.mp4` ..... video ukázka autentizační části
  - `user.mp4` ..... video ukázka uživatelské části
- `doc` ..... adresář s HTML vývojářskou dokumentací
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zrojové kódy práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF