FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Educhild – Dopracování a vydání android aplikace
Student:	Bc. Petr Šíma
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je analýza a dokončení aktuální mobilní aplikace Educhild a následná realizace potřebného automatizovaného procesu vydávání na Google Play. Po realizaci této práce by mobilní část aplikace měla být plně přístupná uživatelům.

Postupujte v těchto krocích:
1. Analyzujte současný stav android aplikace Educhild, zaměřte se na chybějící funkcionalitu.
2. Analyzujte možnosti automatického nasazení android aplikace na Google Play s ohledem na snadnou testovatelnost vývojové části aplikace.
3. Na základě analýzy navrhněte vhodná řešení.
4. Implementujte a realizujte navržené.
5. Výsledné řešení důkladně otestujte.
6. Shrňte získané zkušenosti.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. ledna 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Educhild – Dopracování a vydání android aplikace

Bc. Petr Šíma

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

4. května 2023

Poděkování

Poděkování bych chtěl věnovat hlavně svému vedoucímu práce Ing. Jiřímu Hunkovi za jeho pomoc a rady, které mi během vzniku této práci dával.

Prohlášení

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 4. května 2023 . . . . . .

Odkaz na tuto práci

Šíma, Petr. *Educhild – Dopracování a vydání android aplikace*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato práce se zabývá aplikací Educhild. Nejprve analýzou aplikace, kdy specifikuje funkci, kterou je potřeba do aplikace přidat. Následně pokrývá návrh rozšíření aplikace. Věnuje se tématu modularizace a přináší návrh rozdělení aplikace do modulů. Popisuje, jak strukturovat sestavovací skripty pro Gradle. Integruje Jetpack Compose do aplikace a vysvětluje, jak samotný framework funguje. Nová funkce je samozřejmě i implementována pomocí jazyka Kotlin a následně i otestována.

Klíčová slova Educhild, Kotlin, Android, Jetpack Compose, Gradle, modularizace

Abstract

This thesis is about Educhild application. First, it analyzes the application by specifying functionality that needs to be added. Then, the thesis covers design of the new feature. It adresses the topic of modularization and makes a proposal for dividing the application into modules. The thesis describes how to structure Gradle build scripts and integrates Jetpack Compose into the application. It also explains how Jetpack Compose works itself. Naturally, the new feature is implemented using Kotlin and subsequently tested.

Keywords Educhild, Kotlin, Android, Jetpack Compose, Gradle, modularization

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Aplikace Educhild se věnuje problematice vzdělávání dětí, a to za pomoci mobilní aplikace. To je v dnešní době velmi dobré téma, protože děti tráví na mobilních telefonech velké množství času. Proto je vhodné tento čas využít a vhodným způsobem je donutit, aby jej využívali i pro svůj prospěch.

Projekt je stále ve fázi vývoje, avšak obsahuje mobilní aplikaci pro operační systém Android, webovou aplikaci a backend. Hlavním bodem je ale mobilní aplikace. Ta na základě toho, jak dítě používá svoje zařízení, zobrazuje kvízy, které musí vyplnit, aby mohlo pokračovat v používání vybraných aplikací. To jak se aplikace pro dítě chová konfiguruje rodič, který může vybírat po jak dlouhé době se mají kvízy zobrazovat a samozřejmě jaké konkrétní kvízy.

Práce obsahuje čtyři kapitoly. První z nich se věnuje analýze mobilní a webové aplikace. Následně obě porovná a na základě toho zjistí, zda v mobilní aplikací chybí nějaká důležitá funkce. Pro ni nakonec specifikuje potřebné požadavky.

Druhá kapitola se věnuje návrhu toho, jak funkci implementovat a jak správně rozšířit aplikaci. Samozřejmostí je také návrh doménového modelu pro tuto funkci.

Třetí kapitola popisuje, jak je vše implementované. Nechybí v ní popis použitých technologií a jejich využití při implementaci. Věnuje se například i tomu, jak je definované uživatelské rozhraní. Na závěr této kapitoly je popsáno, jak funguje příprava aplikace pro publikování na Google Play.

Čtvrtá kapitola popisuje, jak probíhalo testování této nové funkce. Obsahuje jaké druhý testů byly použity a jak konkrétně jsou aplikovány.

Cílem této práce je dokončit Android aplikaci a to tím, že se přidá chybějící funkce, kterou webová aplikace nabízí. Je potřeba aplikaci korektně o tuto funkci rozšířit a zlepšit samotnou strukturu projektu aplikace. Proto je nutné zkontrolovat rozložení aplikace do modulů a popřípadě toto upravit. Následně zajistit, aby bylo rozšíření o další funkce jednodušším úkonem. Následně připravit automatizované vydávání na Google Play, a to za pomoci

Gitlab CI, které se již v aplikaci používá. Na závěr samozřejmě novou funkci řádně otestovat.

Kapitola 1

Analýza

Educhild je projekt, který obsahuje mobilní aplikaci pro zařízení s operačním systémem Android, dále webovou aplikaci a backend.

Primární je však mobilní aplikace. Základní funkcionalita této aplikace je spouštění kvízů dětem při používání vybraných aplikací.

Aplikace je určená pro rodiče, kteří určují, kdy se mají kvízy zobrazit. Konkrétně nastavují interval, po kterém se kvízy budou spouštět, nebo nastavují rozvrh, kdy nebude možné některé aplikace používat vůbec. Tyto aplikace je samozřejmě možné také vybírat ze seznamu nainstalovaných aplikací v daném zařízení. Dalším cílovým uživatelem je dítě. Aplikace sleduje jeho aktivitu a na základě toho, jak rodič aplikaci nakonfiguroval, se zobrazují kvízy. Aby však dítě bylo motivované kvízy vyplňovat, dostane za každý kvíz odměnu v podobě času, za který může používat vybrané aplikace. Další odměnou je pak měna, za kterou si může vyzvednout odměny, které rodič nastaví.

Tato kapitola se bude zabývat analýzou obou frontendových klientů, tedy Android a webové aplikace. Pro každou platformu budou sepsány jednotlivé případy užití spolu s aktéry, kteří je vykonávají.

Na základě analýzy bude možné všechny funkce porovnat a zjistit, která v Android aplikaci chybí. Protože už bude pomocí případů užití specifikované, jak se má chovat, je možné vydefinovat funkční a nefunkční požadavky pro tuto novou funkci.

Jedním z výstupů tedy budou požadavky, kterými se bude zabývat tato práce. Pro ověření, že požadavky pokrývají všechny případy užití, bude vytvořena i tabulka pokrytí.

Jelikož projekt obsahuje i backend, kde se spravují všechna data, která jsou pro aplikaci důležitá, bude provedena i analýza API, které backend vystavuje. Jako výstup této analýzy bude souhrn požadavků potřebných pro rozšíření aplikace. Díky tomu bude možné provést správný návrh v další kapitole.

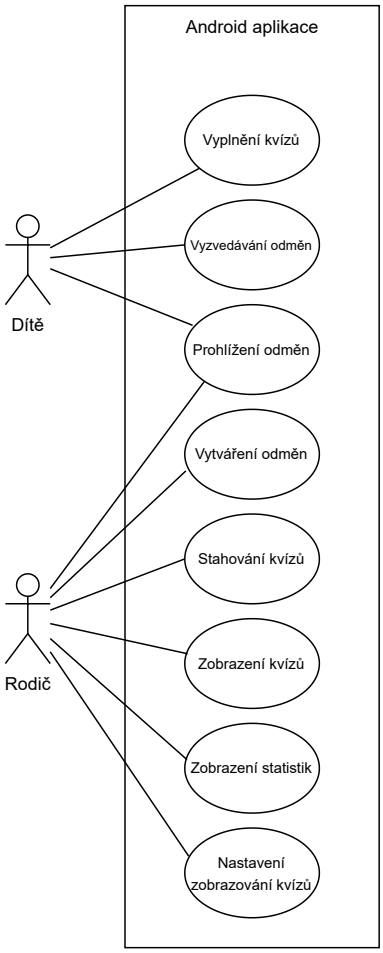
1.1 Android aplikace

Aktuální stav Android aplikace podporuje několik funkcí, které mohou provádět dva typy uživatelů. Ty můžeme rozdělit na rodiče a děti. Více si je pak specifikujeme v následující kapitole.

Rodič po instalaci aplikace nastaví chování aplikace, tedy jaké kvízy a kdy se mají zobrazovat. Poté přepne do dětského režimu a může mobilní telefon předat dítěti. To si pak může mobilní telefon používat jak chce. Avšak pokud se splní podmínka pro zobrazení kvízu, musí dítě kvíz vyplnit.

Tyto jednotlivé případy užití můžeme vidět na diagramu 1.1 a budou popsány v následujících podkapitolách.

Případy užití specifikují akce mezi uživatelem a systémem, díky kterým dosáhne určitého cíle. [1]



Obrázek 1.1: Diagram případů užití Android aplikace

1.1.1 Aktéři

1.1.1.1 Rodič

Rodič je uživatel, který má přístup do rodičovské části aplikace, ta je oddělená od té dětské PINem, který zná jen rodič.

V této části aplikace může spravovat všechno nastavení pro dětský režim. Tedy stahování kvízů, přiřazování kvízů, vytváření odměn, ale i zobrazení statistik, jak dítě mobilní telefon používá. Na základě toho pak může jednotlivé aplikace přidávat do monitorovaných a u nich se bude odčítat čas do zobrazení dalšího kvízu.

1.1.1.2 Dítě

Uživatel, který má přístup pouze do dětského režimu, do kterého mu aplikaci přepne rodič. Nemá možnost se do rodičovské části vrátit, protože je chráněná PINem.

Dítě si může v dětské části samo plnit kvízy a získávat tak čas na používání monitorovaných aplikací, popřípadě her.

Účelem zobrazování kvízů je dítě vzdělávat a zároveň ho motivovat k vyplňování kvízů odměnami, které mu rodič vytvoří.

1.1.2 Případy užití

1.1.2.1 UC1 Vyplnění kvízů

1. Systém dítěti zobrazí kvíz s daným počtem otázek, které mohou být třech typů a mohou obsahovat obrázky. A to buď v zadání, vysvětlení nebo v možnostech, které může dítě vybírat, popřípadě spojovat
2. Dítě postupně odpovídá na otázky
3. Systém po každé odpovědi zobrazí vysvětlení odpovědi, aby dítě pochopilo dané téma.

1.1.2.2 UC2 Vyzvedávání odměn

1. Systém dítěti nabízí seznam odměn
2. Dítě si může vybranou odměnu za měnu, v podobě lístečků, zakoupit.

1.1.2.3 UC3 Prohlížení odměn

1. Systém dítěti nebo rodiči nabízí seznam odměn
2. Ten si může tyto odměny prohlédnout

3. Systém u každé odměny zobrazuje obrázek, název, popis a ceny.

1.1.2.4 UC4 Vytváření odměn

1. Rodič má možnost vytvářet pro dítě odměny
2. Systém vyžaduje pro vytvoření odměn obrázek, název, popis a cenu.

1.1.2.5 UC5 Zobrazení kvízů

1. Systém nabízí rodiči seznam stažených kvízů
2. Rodič si u každého kvízu může zobrazit jejich název, popis, doporučený věk, kategorie, jazyk a seznam otázek
3. Systém umožňuje kvízy přiřadit
4. Rodič může přiřadit tyto kvízy svým dětem.

1.1.2.6 UC6 Stahování kvízů

1. Systém dovoluje stáhnout kvízy do lokálního úložiště zařízení
2. Rodič může následně s těmito kvízy, včetně otázek, pracovat i bez připojení k internetu
3. Systém tyto stažené kvízy zobrazuje v odděleném seznamu.

1.1.2.7 UC7 Zobrazení statistik

1. Systém zobrazuje rodiči zaznamenané statistiky o používání mobilního telefonu dítětem
2. Rodič si je může prohlížet a na základě toho může nastavit chování celé aplikace.

1.1.2.8 UC8 Nastavení zobrazování kvízů

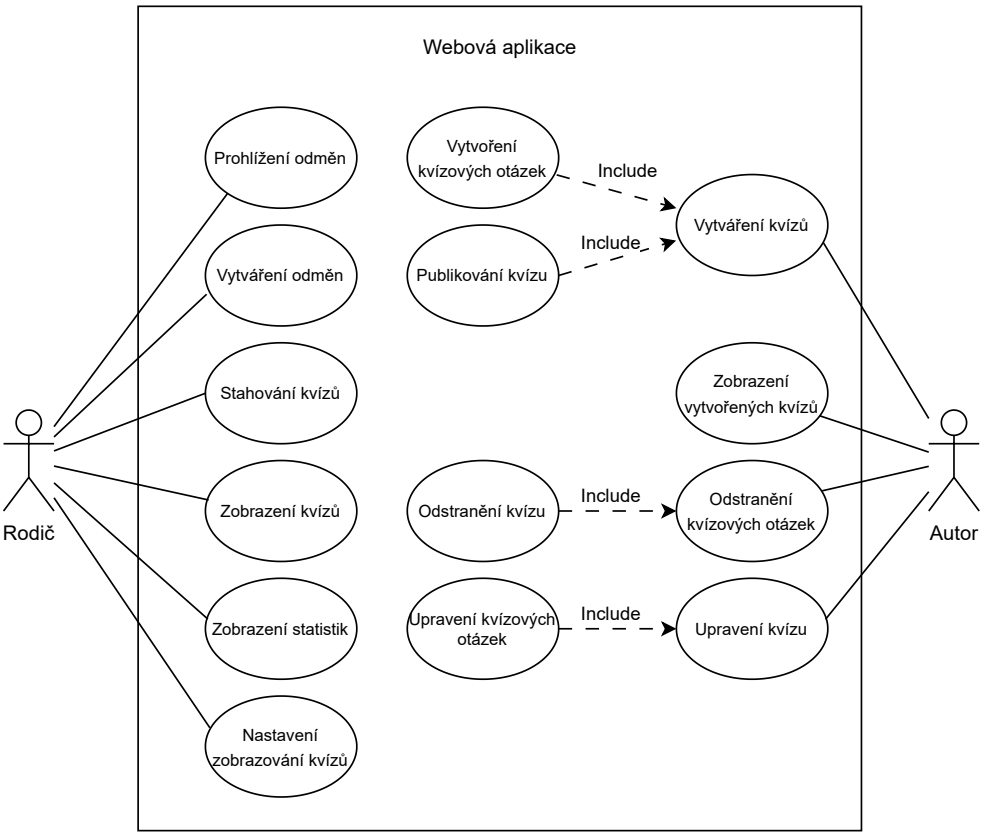
1. Systém umožňuje rodiči nastavit pravidla pro zobrazování kvízů
2. Rodič může nastavit, jak dlouho může dítě používat monitorované aplikace nebo nastavit rozvrh, kdy tyto aplikace nebude možné používat vůbec
3. Systém v dětském režimu kontroluje, zda jsou tato pravidla splněná.

1.2 Webová aplikace

Webová aplikace slouží jako nástroj, kde rodič může spravovat nastavení pro dítě, aby nemusel používat mobilní telefon a měl vše zobrazené více přehledněji.

Další velkou funkcionalitou je možnost kvízy vytvářet. Uživatele, který takové kvízy vytváří, můžeme označit jako autora. Je tedy možné si uživatele rozdělit na dva aktéry: rodič a autor.

Všechny případy užití jsou pak vidět na obrázku 1.2. Protože všechny případy použití, které provádí rodič jsou shodné s těmi u mobilní aplikace, shrneme si pouze ty, které jsou odlišné.



Obrázek 1.2: Diagram případů užití webové aplikace

1.2.1 Aktéři

1.2.1.1 Rodič

Rodič je uživatel, který podobně jako v mobilní aplikaci spravuje nastavení pro dětský režim.

1.2.1.2 Autor

Autor vytváří kvízy. Tyto kvízy může následně publikovat a díky tomu si je mohou stáhnout další uživatelé pro své děti.

1.2.2 Případy užití

1.2.2.1 UC9 Vytváření kvízu

1. Systém umožňuje autorovi vytvářet vlastní kvízy
2. Autor pro vytvoření kvízu vyplní název kvízu, popis, doporučený věk dítěte, kategorii a jazyk
3. Systém následně vytvoří tento kvíz se stavem *CONCEPT*
4. Autor jako jediný tento kvíz následně vidí v seznamu vytvořených kvízů

1.2.2.2 UC10 Vytvoření kvízových otázek

1. Systém během vytváření kvízu nabízí uživateli možnost přidávat kvízové otázky
2. Autor vybere jeden z typů:
   - s otevřenou odpovědí - číslo/text
   - výběr z více možností
   - spojování dvojic.

   Každá otázka obsahuje parametry zadání otázky a vysvětlení odpovědi. Pro jednotlivé typy se poté mění odpovědi. Otevřená otázka obsahuje pouze hodnotu odpovědi. Výběr z více možností pak seznam odpovědí spolu s označením, zda je správně. Spojování dvojic pak seznam těchto párů.

   Otázky mohou volitelně obsahovat obrázky, a to u zadání a vysvětlení. Dále pak u výběru z více možností a u spojování dvojic u jednotlivých položek.
3. Systém uloží vytvořenou otázku a přiřadí ji ke kvízu.

1.2.2.3 UC11 Publikování kvízu

1. Systém po vytvoření kvízu dovolí jeho publikování
2. Autor může svůj kvíz publikovat
3. Systém nastaví stav na *PUBLISHED* a tím se stane dostupný pro ostatní uživatele.

1.2.2.4 UC12 Zobrazení vytvořených kvízů

1. Systém zobrazuje uživateli kvízy, které vytvořil
2. Autor si může zobrazit u těchto kvízů detail, na kterém jsou mimo informací o kvízu i jednotlivé otázky.

1.2.2.5 UC13 Odstranění kvízových otázek

1. Systém na seznamu otázek umožňuje jejich smazání
2. Autor může smazat jednotlivé kvízové otázky
3. Systém odstraní tuto otázku z kvízu

1.2.2.6 UC14 Odstranění kvízu

1. Systém na seznamu kvízů umožňuje jejich smazání
2. Autor může smazat kvíz
3. Systém nastaví stav na *DELETED* a tím umožní jeho pozdější obnovení.

1.2.2.7 UC15 Upravení kvízu

1. Systém nabídne u jednotlivých kvízů autorovi úpravu
2. Autor upraví u kvízu libovolné parametry
3. Systém aplikuje tuto úpravu.

1.2.2.8 UC16 Upravení kvízových otázek

1. Systém nabídne u jednotlivých otázek autorovi úpravu
2. Autor jakkoliv změní otázku
3. Systém aplikuje tuto změnu.

1.3 Popis nové funkcionality

Z analýzy aktuálního stavu obou aplikací a jejich případů užití vyplývá, že Android aplikace postrádá možnost vytváření kvízů. V následující části si tedy shrneme jednotlivé požadavky.

Tyto požadavky můžeme rozdělit na dva typy, a to konkrétně na funkční a nefunkční. Funkční požadavky popisují, jak se má systém chovat za určitých podmínek a zahrnuje jednotlivé funkce, které systém musí umět. Nefunkční požadavky jsou pak obecné požadavky na systém. [2]

U funkčních požadavků bude kromě jejich náročnosti uvedena i jejich kategorie a to podle *MOSCOW*. Ta má následující kategorie.

- *Must have* požadavek, který je nezbytně nutný pro danou funkci
- *Should have*, opět důležitý požadavek, avšak není pro fungování nezbytně nutný
- *Could have* požadavky mající malý důsledek, pokud budou chybět
- *Will not have* označuje požadavky, které nejsou prioritou a proto nebudou zahrnuty.

[3]

1.3.1 Funkční požadavky

1.3.1.1 FR1 Vytváření kvízů

Popis: Systém uživateli s rolí Autor umožní vytvářet kvízy. Pro vytvoření kvízu musí uživatel zadat následující parametry.

- Název kvízu
- Popis
- Doporučený věk dítěte
- Kategorii
- Jazyk

Systém poté vytvoří kvíz se stavem *CONCEPT* a uživatel si tento kvíz bude moci dále upravovat.

Kategorie: Must have
Složitost: Střední

1.3.1.2 FR2 Správa vytvořených kvízů

Popis: Systém uživateli umožní zobrazit si své vytvořené kvízy. Tyto kvízy pak může uživatel používat jako rodič.

Kvízy může dále mazat, popřípadě upravovat. Po smazání kvízu, systém dovolí uživateli kvíz kdykoliv obnovit.

Kategorie: Must have
Složitost: Vysoká

1.3.1.3 FR3 Vytváření kvízových otázek

Popis: Uživatel může u každého kvízu vytvářet otázky.

Otázky mohou být třech typů.

1. S otevřenou odpovědí - číslo/text
2. Výběr z více možností
3. Spojování dvojic

Všechny otázky obsahují následující atributy.

- Textové zadání otázky
- Obrázkové zadání otázky
- Textové vysvětlení odpovědi
- Obrázkové vysvětlení odpovědi

Pro typ *S otevřenou odpovědí* obsahuje otázka navíc ještě.

- Textová správná odpověď

Typ *Výběr z více možností* otázka obsahuje seznam odpovědí s následujícími atributy.

- Textové znění odpovědi
- Obrázek odpovědi
- Příznak, zda je správně

Spojování dvojic pak má navíc seznam páru, kde každý položka obsahuje.

- Textové znění
- Obrázek

Kategorie: Must have
Složitost: Vysoká

1.3.1.4 FR4 Správa kvízových otázek

Popis: Systém dovolí uživateli zobrazovat ke každému kvízu jeho otázky. Zároveň systém umožní tyto otázky editovat nebo je smazat.

Kategorie: Should have
Složitost: Střední

1.3.1.5 FR5 Synchronizace vytvořených kvízů

Popis: Systém sám synchronizuje změny na vytvořených kvízech. Tedy když autor změní kvíz například ve webové aplikaci, v té mobilní, kde je uživatel přihlášený pod stejným účtem, se zobrazí aktuální kvíz s danou změnou.

Kategorie: Should have
Složitost: Střední

1.3.2 Nefunkční požadavky

1.3.2.1 NFR6 Offline podpora

Popis: Uživatel předpokládá, že kvízy, které už jednou zobrazil v mobilní aplikaci, tak budou dostupné pro prohlížení i bez připojení k internetu.

Tyto kvízy bude možné v offline režimu přiřazovat dětem, které je budou moci vyplňovat.

1.3.2.2 NFR7 Rozšiřitelnost

Popis: Je potřeba zachovat nebo zlepšit rozšiřitelnost aplikace o nové funkce.

1.3.3 Pokrytí případů užití

Funkční požadavky pokrývají všechny případy užití, které jsou pro novou funkcionalitu potřeba. Tyto případy užití byly definovány v kapitole 1.2.2.

Samotné pokrytí lze vidět v tabulce 1.1.

	FR1	FR2	FR3	FR4	FR5
UC9	x				
UC10			x		
UC11		x			x
UC12				x	
UC13				x	
UC14		x			
UC15		x			
UC16				x	

Tabulka 1.1: Tabulka pokrytí případů užití

1.4 API pro tvorbu kvízů

Na základě předchozí části, která specifikovala všechny požadavky pro novou funkci, můžeme analyzovat již připravené API. Díky tomu si zesumarizovat potřebné endpointy, které budou pro rozšíření aplikace potřeba.

Samotné API poskytuje svou dokumentaci, která je dostupná na cestě `api/swagger-ui.html`. Díky tomu je možné jednoduše specifikovat všechny potřebné požadavky, které bude aplikace volat.

1.4.1 Kvízy

Aby bylo možné kvízy tvořit, je potřeba získat roli `Author`.

S touto rolí je pak možné posílat požadavky, pro vytváření kvízů. To samozřejmě zahrnuje i jejich vylistování, úpravu a všechny další operace, které byly specifikované ve funkčních požadavcích.

1.4.1.1 Získání role Author

Popis

Tento požadavek přiřadí uživateli roli Author, pokud ji ještě nemá.

Request: POST authors

Tělo požadavku musí obsahovat pouze preferovaný jazyk autora.

Response

Odpověď obsahuje pouze informace o vytvořeném autorovi, tedy pouze jeho preferovaný jazyk.

1.4.1.2 Vytvoření kvízu

Popis

Vytvoří nový kvíz, který má za autora uživatele, který požadavek zaslal. Tento kvíz bude mít po vytvoření stav *CONCEPT* a díky tomu nebude viditelný pro ostatní uživatele.

Request: POST quizzes

Požadavek pro vytvoření obsahuje všechny potřebné informace o kvízu a můžeme ho vidět na ukázce 1.

Response

V odpovědi se vrátí identifikátor nového kvízu.

```
{
  "name": "Quiz name",
  "recommendedAge": 10,
  "description": "This is test quiz",
  "language": "en"
}
```

Výpis kódu 1: Ukázka requestu pro vytvoření kvízu

1.4.1.3 Seznam dostupných jazyků

Popis

Protože pro vytvoření kvízu je potřeba přiřadit jazyk a seznam dostupných jazyků je dostupný na serveru, je potřeba provést požadavek pro jejich vylistování.

Request: GET languages

Response

Vrací seznam dostupných jazyků definovaných jejich názvem.

1.4.1.4 Seznam dostupných kategorií

Popis

Obdobně jako u jazyků je seznam všech dostupných kategorií kvízu dostupná na backendu.

Request: GET categories

Response

Vrací seznam dostupných kvízových kategorií. Každá tato kategorie má identifikátor, jméno a barvu.

1.4.1.5 Přiřazení kategorie

Popis

K jednotlivým kvízům je po jejich vytvoření potřeba přiřadit kategorii.

Request: POST quizzes/{quizId}/categories/{categoryId}

Response

Vrací prázdné tělo v odpovědi.

1.4.1.6 Úprava kvízu

Popis

Vytvořené kvízy je možné po jejich vytvoření upravit. Pomocí tohoto endpointu lze kvízy také publikovat a to tím, že se změní stav na *PUBLISHED*.

Request: PUT quizzes/{quizId}

Na rozdíl od vytvoření kvízu, obsahuje tělo navíc identifikátor a stav. Ukázku můžeme vidět ve výpisu kódu 2.

Response

Vrací prázdné tělo v odpovědi.

```
{
  "id": "...",
  "state": "PUBLISHED",
  "name": "Editted quiz name",
  "recommendedAge": 10,
  "description": "This is test quiz",
  "language": "en"
}
```

Výpis kódu 2: Ukázka requestu pro úpravu kvízu

1.4.1.7 Smazání kvízu

Popis

Pomocí tohoto požadavku lze kvíz smazat, avšak smazání kvízu znamená, že se kvízu nastaví stav na *DELETED*. Díky tomu je dostupný pro uživatele, které ho mají už stažený, popřípadě je možné ho obnovit pro uživatele, který ho vytvořil.

Request: DELETE quizzes/{quizId}

Response

Vrací prázdné tělo v odpovědi.

1.4.1.8 Seznam vytvořených kvízů

Popis

Požadavek, který vrací seznam vytvořených kvízů pro přihlášeného uživatele. Tyto kvízy je možné filtrovat pomocí jejich stavu, kde stavy mohou být následující.

- CONCEPT
- PUBLISHED
- DELETED

Request: GET quizzes/created?quiz_state={quizState}

Response

Vrací seznam vytvořených kvízů, které mají stav daný v požadavku.

1.4.1.9 Získání aktuální verze kvízu

Popis

Protože je možné kvízy upravovat a poté je potřeba je aktualizovat, má každý kvíz verzi, která se při každé úpravě zvyšuje. Tento endpoint vrací tuto verzi, aby se nemuselo zbytečně získávat všechny informace o kvízu.

Request: GET quizzes/{quizId}/version

Response

Tělo odpovědi obsahuje pouze verzi pro daný kvíz.

1.4.2 Kvízové otázky

Po vytvoření kvízů je potřeba k němu přidat jednotlivé otázky. Samozřejmě je potřeba, aby kvíz vytvořil stejný uživatel, který otázky přidává.

1.4.2.1 Vytvoření kvízových otázek

Popis

Slouží k vytvoření kvízových otázek. Tento endpoint podporuje posílání seznamu otázek. Samozřejmostí je, že podporuje přidání všech typů otázek

Request: POST quizzes/{quizId}/questions

V těle požadavku je jak již bylo zmíněno seznam kvízových otázek.
Každý typ je rozlišený atributem *type*, ten může nabývat hodnot

- open
- multichoice
- pairs

Pro typ s otevřenou odpovědí je vidět ukázka objektu v ukázce 3. Typ výběru z více možností v ukázce 4 a typ spojování dvojic pak v ukázce 5.

Response

Vrací prázdné tělo v odpovědi.

```
{
  "type": "open",
  "question": {
    "text": "Question example?"
  },
  "explanation": {
    "text": "Explanation text"
  },
  "answer": "Answer"
}
```

Výpis kódu 3: Ukázka objektu pro otázku s otevřenou odpovědí

```
{
  "type": "multichoice",
  "question": {
    "text": "Question example?"
  },
  "explanation": {
    "text": "Explanation text"
  },
  "answers": [
    {
      "correct": true,
      "text": "Answer 1"
    },
    {
      "correct": false,
      "text": "Answer 2"
    }
  ]
}
```

Výpis kódu 4: Ukázka objektu pro otázku s výběrem z více možností

```
{
  "type": "pairs",
  "question": {
    "text": "Question example?"
  },
  "explanation": {
    "text": "Explanation text"
  },
  "pairs": [
    {
      "a": {
        "text": "Pair 1a"
      },
      "b": {
        "text": "Pair 1b"
      }
    },
    {
      "a": {
        "text": "Part 2a"
      },
      "b": {
        "text": "Part 2b"
      }
    }
  ]
}
```

Výpis kódu 5: Ukázka objektu pro otázku se spojování dvojic

1.4.2.2 Upravení kvízové otázky

Popis

Každou otázku je možné upravovat, a to změnou zadání, ale samozřejmostí je také změna jejího typu.

Request: PUT questions/{questionId}

Tělo požadavku je stejné jako u vytvoření nové otázky, tedy příkladu jsou stejné jako ve výpisech 3, 4 a 5.

Response

Vrací prázdné tělo v odpovědi.

1.4.2.3 Seznam kvízových otázek

Popis

Vrací seznam kvízových otázek k danému kvízu

Request: GET quizzes/{quizId}/questions

Response

Tělo odpovědi obsahuje již zmíněný seznam. Objekty pro jednotlivé otázky jsou velmi podobné těm z ukázek 3, 4 a 5, navíc obsahují u každého objektu pouze jeho identifikátor.

1.4.2.4 Smazání kvízové otázky

Popis

Tento požadavek slouží ke smazání kvízové otázky z kvízu.

Request: DELETE questions/{questionId}

Response

Vrací prázdné tělo v odpovědi.

1.4.2.5 Nahrání obrázku

Popis

Protože otázky mohou obsahovat obrázky, je potřeba je nějak nahrávat. Aby se nemusely posílat všechny v jednom požadavku.

Pro každý nahraný obrázek se na serveru vygeneruje název, podle kterého se obrázek identifikuje.

Tyto obrázky se poté přiřazují k otázkám pomocí jejich názvu, při vytváření nebo úpravě kvízových otázek.

Request: POST images

Obsahuje obrázek, který se má nahrát.

Response

Vrací název nahraného obrázku.

1.4.2.6 Získání obrázku

Popis

Díky tomuto požadavku je možné získávat nahrané obrázky se serveru.

Request: GET images/{imageName}

Response

Vrací daný obrázek definovaný jménem, které server vygeneroval.

1.5 Shrnutí

V průběhu této kapitoly byly podrobně zkoumány obě frontendové aplikace s cílem identifikovat možné nedostatky a příležitosti pro zlepšení. Na základě této analýzy bylo zjištěno, že Android aplikace postrádá funkci umožňující tvorbu kvízů.

Pro tuto chybějící funkci byly následně definovány funkční a nefunkční požadavky, které zahrnují popis očekávaného chování nové funkce. Tyto požadavky se stanou základem pro další kapitoly, kde budou sloužit pro návrh a následnou implementaci zmíněné funkce.

Kromě toho byla provedena analýza stávajícího API, které slouží k vytváření kvízů. Výsledkem toho je sada požadavků, které je nezbytné použít při implementaci nové funkce do aplikace. To zajistí, že funkce bude použitelná napříč všemi podporovanými platformami projektu.

Kapitola 2

Návrh

Tato kapitola diplomové práce je zaměřena na návrh rozšíření mobilní aplikace o novou funkci, která byla podrobně analyzována a popsána v předchozí kapitole.

Na úvod kapitoly se podrobněji seznámíme s celkovou koncepcí aplikace. Budeme se zaměřovat na to, jak navrhnout a vylepšit rozvržení aplikace do modulů. Součástí toho bude integrace nové funkcionality do této navržené struktury.

Následně se budeme zabývat návrhem doménového modelu, který bude zodpovědný za logiku nové funkce. V této části se budeme věnovat především návrhu pro část vytváření kvízů.

Výsledkem této kapitoly bude podrobný návrh, který bude základem pro následující kapitolu zaměřenou na implementaci nové funkce.

2.1 Modularizace

Modularizace je způsob, jak uspořádat jednotlivé celky kódu do samostatných částí, které jsou na sobě závislé. Každá tato část se nazývá modul. Každý tento modul je pak nezávislý a slouží k jasnému účelu. Rozdělení problému na menší a jednodušší podproblémy snižuje složitost návrhu a zajišťuje snazší údržbu celého systému.

Použití modularizace má kromě lepší udržitelnosti a lepší kvality kódu i dalších několik výhod.

První takovou výhodou je přepoužitelnost. Modularizace umožňuje sdílení kódu a vytváření více aplikací ze stejného základu. Moduly můžeme označit jako takové stavební bloky. Proto by aplikace měla být souhrnem svých funkcí, kde jsou funkce uspořádány jako samostatné moduly. Funkce, které pak určitý modul poskytuje mohou, ale nemusí být konkrétní aplikací povoleny.

Jako druhou výhodu můžeme zmínit striktní zapouzdření, kdy moduly umožní snadno kontrolovat, co se vystaví ostatním částem celého projektu.

Všechno kromě veřejného rozhraní se pak může označit jako `internal` nebo `private`, aby bylo zabráněno použití mimo modul.

Poslední výhodou je přizpůsobitelné doručení. Díky možnostem, které nabízí Google Play se mohou některé funkce doručovat do aplikace pod nějakou podmínkou nebo jen když je potřeba tyto funkce používat.

Tyto výhody jsou dosažitelné jen díky modularizaci. Ta však pomůže i u dalších, které jsou možné docílit pomocí jiných technik. Mezi tyto výhody můžeme jmenovat následující.

- Rozšiřitelnost
- Vlastnictví
- Zapouzdření
- Testovatelnost
- Čas sestavení projektu

Špatná modularizace však může přinést i nějaké nástrahy, pokud je aplikována špatně. Při návrhu by se měla zohlednit četnost modulů, tedy kolik projekt obsahuje modulů a jakou úroveň tato modularizace bude dosahovat. Toto ovlivňuje hlavně rozsah celého projektu.

Při špatném návrhu může dojít k příliš velké četnosti modulů. Každý modul přináší určitou režii v podobě větší složitosti sestavení a velkému opakování kódu. To způsobuje pomalejší práci na projektu a horší udržitelnost.

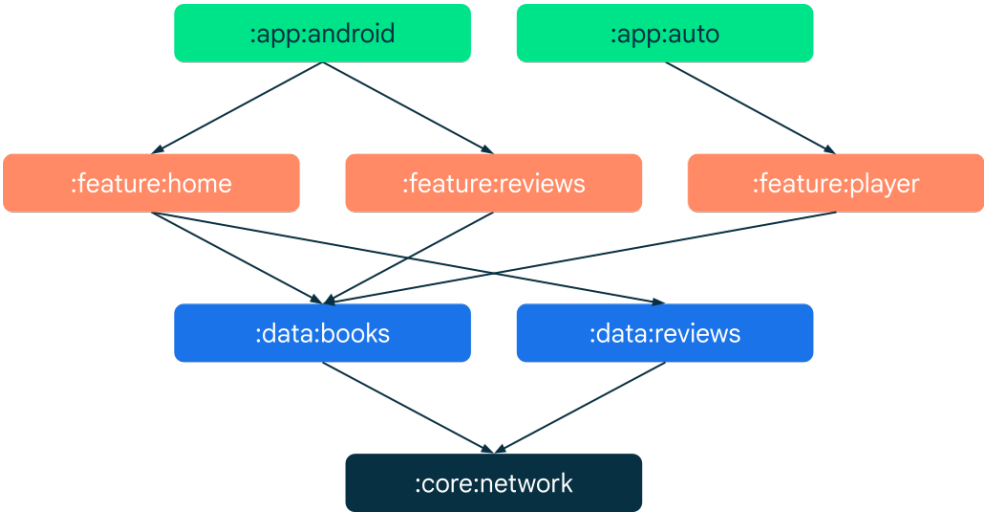
Na druhou stranu může dojít k malému počtu velkých modulů. To způsobí vznik dalších monolitů a všechny výhody, které modularizace nabízí se vytratí.

Je důležité zohlednit rozsah celého projektu při rozhodování o modularizaci, protože v případě menších projektů nemusí být modularizace vždy vhodná a efektivní. Výhody, jako je rozšiřitelnost, rychlost sestavení a snadnější údržba, se v malých projektech nemusí projevit tak výrazně, jako v rozsáhlejších projektech.

U menších projektů může modularizace vést k zbytečné komplexitě a zvýšeným nákladům na vývoj a údržbu, protože reorganizace kódu a rozdělení funkcionalit do samostatných modulů může způsobit zvýšenou režii a zhoršení přehlednosti celkové struktury projektu.

Vzhledem k těmto aspektům je tedy klíčové pečlivě zvážit, zda modularizace přináší skutečné výhody pro daný projekt, a zohlednit to při rozhodování o jeho architektuře a struktuře. [4]

Příklad jak může takové rozdělení do modulů vypadat je vidět na obrázku 2.1.



Obrázek 2.1: Příklad rozdělení aplikace do modulů [5]

2.1.1 Aktuální řešení

V současné době je aplikace rozdělena do několika modulů, jak je znázorněno na diagramu na obrázku 2.2. Toto rozdělení bylo původně navrženo tak, aby usnadnilo vývoj aplikace a samozřejmě přineslo i výhody zmíněné v předchozí části. Nicméně, s růstem projektu se aplikace rozrostla do poměrně velkého rozsahu. To způsobilo, že stávající rozdělení modulů se stalo neefektivním a nedostačujícím. Jednotlivé moduly se staly velkými a obtížně udržitelnými. To vede ke zpomalení vývoje a prodloužení času potřebného pro sestavení aplikace.

Aby bylo možné tyto problémy vyřešit, je nutné provést revizi současného rozdělení do modulů a zvážit reorganizaci aplikace do menších a více nezávislých jednotek. Při této reorganizaci by měl být kladen důraz na minimalizaci závislostí mezi jednotlivými moduly.

2.1.1.1 Model

Tento modul zahrnuje modely pro všechny modelové třídy používané v aplikaci. Nicméně toto současné rozdělení představuje problém, protože obsahuje jak modely, které definují entity pro databázi, tak i datové objekty, které slouží pro komunikaci s API.

Tímto způsobem se však poskytují všechny modelové třídy najednou, což není optimální, obzvláště pokud jsou některé z nich spjaty s databázovými či API třídami. Toto může způsobit komplikace při správě kódu a ovlivnit udržitelnost a rozšiřitelnost celé aplikace.

2.1.1.2 Network

Tento modul pokrývá všechnu logiku, která slouží k síťové komunikaci. Tedy definice jednotlivých požadavků, včetně jejich správné konfigurace. Dále obsahuje logiku, která tyto požadavky volá a zpracovává.

Tím, že modul obsahuje tuto veškerou logiku, stal se velkým a podle problémů s modularizací, která byly uvedena v předchozí části, není zahrnutí všech těchto komponentů do jednoho modulu správné.

2.1.1.3 Persistance

Podobně jako předešlý modul, tento obsahuje všechnu logiku, která souvisí s uložením na lokálním úložišti zařízení. Konkrétně tedy databáze a persistentní hodnoty dané svým klíčem.

Nevýhodou tohoto modulu je opět určitě jeho velikost a například logika, která s databází komunikuje. Měla by být extrahovaná do zvláštních modulů.

2.1.1.4 Data

Modul, který obsahuje veškerou logiku a je závislý na modulech *Network* a *Persistance*. Díky tomu modul obsahuje velký počet tříd. Probíhá zde veškerá synchronizace mezi lokální databází a backendem. Připravuje data pro jednotlivé obrazovky v aplikaci.

U tohoto modulu je problém s velikostí modulu ještě větší, proto je určitě ke zvážení jeho rozdělení do dalších modulů.

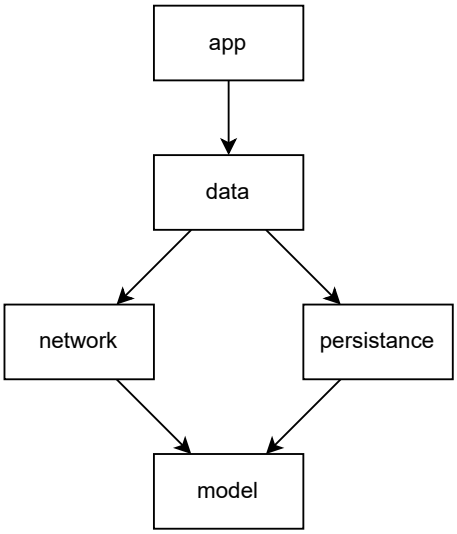
2.1.1.5 App

Modul, který slouží jako základ pro každou Android aplikaci. V případě této aplikace zahrnuje definici všech obrazovek, spolu s prezentační logikou. Dále modul obsahuje všechny komponenty, které definuje Android framework, jako jsou například třídy, které slouží ke spouštění synchronizace dat.

Protože je to modul, který je závislý na všech ostatních, tak při každé změně kdekoliv v projektu dochází k sestavení tohoto modulu. Protože je tento modul však velký, prodlužuje to čas pro sestavení aplikace. Další problém, který byl zmíněn i v předchozí kapitole o modularizaci je to, že modul obsahuje všechny funkce, které aplikace nabízí a díky tomu opět ztrácí modularizace své výhody.

2.1.2 Studie již existujících řešení

Aby bylo možné přijít s lepším návrhem pro modularizaci, tato část se bude věnovat studii projektů, které ji využívají. Prvním z nich je aplikace *Now in Android*, která se drží doporučeného postupu pro modularizaci. Druhou aplikací je *Tivi*, která pro to využívá trochu jiný přístup, avšak má o něco větší rozsah a tím se blíží k aplikaci, kterou popisuje tato práce.

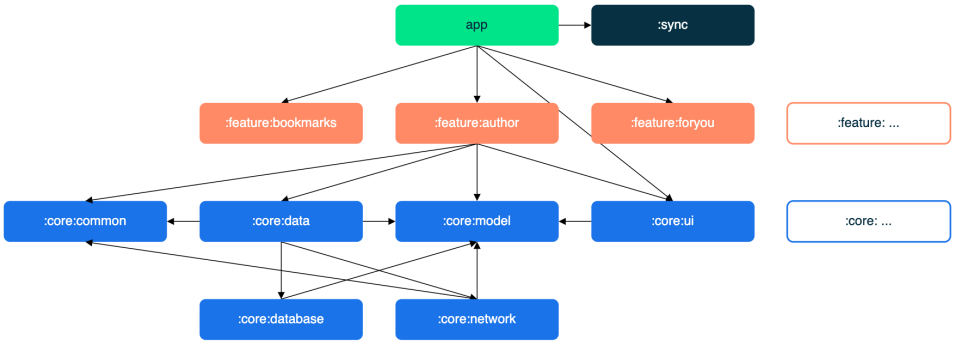


Obrázek 2.2: Aktuální řešení rozdělení do modulů v aplikaci

2.1.2.1 Now in Android

Now in Android je plně funkční aplikace pro Android vytvořená pomocí jazyka Kotlin a pomocí Jetpack Compose. Řídí se osvědčenými postupy pro návrh a vývoj Android aplikací a má sloužit jako reference pro vývojáře. Aplikace sama o sobě slouží jako prohlížeč novinek a zpráv ze světa Android vývoje. [6]

Aplikace není moc velkého rozsahu, ale jak již bylo zmíněno, řídí se oficiálním postupem pro strukturování Android aplikací. Jak jsou rozdělené moduly a jak na sobě závisí je vidět na obrázku 2.3.



Obrázek 2.3: Rozdělení do modulů v Now in Android [7]

Samotné rozdělení dělíme do několika úrovní. První takovou úrovní jsou *core* moduly. Ty obsahují pomocný kód a specifické závislosti, které je třeba sdílet mezi ostatní moduly v aplikaci. Tyto moduly mohou záviset na jiných *core* modulech, ale neměly by záviset na ostatních modulech.

Druhou úrovní jsou moduly *feature*, které jsou specifické pro danou funkci a jsou rozděleny tak, aby obsahovaly pouze jednu odpovědnost v aplikaci. Tyto moduly lze v případě potřeby opakovaně používat v libovolné aplikaci, včetně testovacích, nebo v jiných konfiguracích. Přitom se stále udržují oddělené a izolované od sebe. Pokud je třída potřebná pouze pro jeden modul, měla by v tomto modulu zůstat. Pokud ne, měla by být extrahována do příslušného *core* modulu. *Feature* moduly by však neměly mít žádné závislosti na jiných *feature* modulech. Závisí pouze na modulech *core*, které využívají.

Jako třetí úroveň je *app*. Ta obsahuje třídy samotné aplikace a k tomu potřebné třídy z Android frameworku. Dále podporuje všechny *feature* moduly, na kterých je závislá. Další závislostí jsou požadované *core* moduly. Tento modul je díky tomu velmi minimalistický a jeho sestavení trvá minimum času.

Ostatní moduly, například pro synchronizaci, benchmark nebo testování, jsou pouze listy v grafu modulů a tím pouze využívají výhod modularizace a nijak její strukturu nenarušují. [8]

I přesto, že tato aplikace je menšího rozsahu, je modularizace provedena mnohem lépe. Moduly jsou menší a je jasně dané, jak na sobě budou záviset. Díky tomu se při změně v nějakém modulu sestavuje pouze to, co je třeba. To je hlavní nevýhodou modularizace v aplikace Educhild, kde se ve většině případů sestavuje většina modulů, které jsou ještě k tomu dost rozsáhlé. Další výhodou je minimální modul *app*, který pouze slouží jako sjednocení všech modulů *feature*.

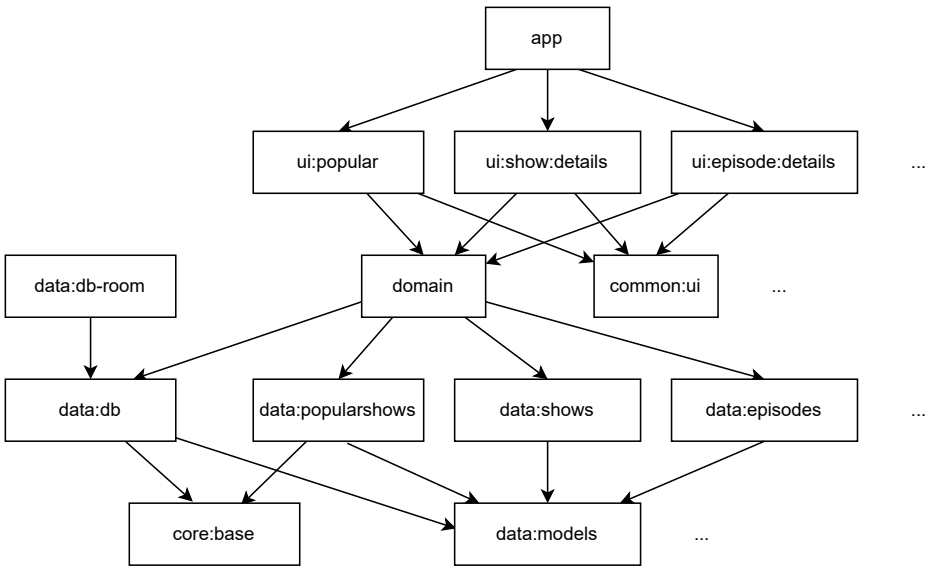
Nevýhodou pro větší aplikace by však byly *core* moduly. Každá změna v takovémto modulu by totiž způsobila sestavení téměř celého projektu. Moduly jsou sice rozděleny podle jejich účelu, nejsou však rozděleny podle jednotlivých funkcí. Jako příklad lze uvést modul *core:model*. Ten obsahuje všechny datové modely, které se v aplikaci používají a navíc na něm závisí mnoho dalších modulů. Úprava kódu v tomto modulu tedy způsobí dlouhý čas sestavení. Větší rozdělení by tedy přispělo k oddělení zodpovědností jednotlivých modulů a ještě by více podpořilo výhody samotné modularizace. To se však týká pouze větších aplikací. Aplikaci rozsahu jako je Now in Android, je s tímto návrhem modulů naprosto adekvátní a větší modularizace by v tomto případě ani nedávala smysl.

2.1.2.2 Tivi

Samotná aplikace je navržena pro prohlížení seriálů, což uživatelům umožňuje zobrazovat detaily o seriálech, procházet informace o jednotlivých seriích a dílech, a vytvářet si seznam oblíbených pořadů. Tato funkce přidávání do oblíbených usnadňuje sledování postupu uživatele v průběhu sledování seri-

álu. Umožňuje mu zachovat přehled o již zhlédnutých dílech a zobrazovat informace o následujících epizodách. Aplikace také nabízí možnost přihlášení pomocí účtu *Trakt*, a díky tomu propojuje data uživatelů s jejich servery. Tato integrace s *Trakt* zjednodušuje proces přihlášení a zajišťuje, že uživatel nemusí v aplikaci opakovaně vyplňovat své údaje. Dalším přínosem spojení s *Trakt* API je možnost zobrazování hodnocení jednotlivých seriálů a jejich epizod, což uživatelům poskytuje další informace při výběru, které seriály sledovat.

Protože je aplikace poměrně velkého rozsahu je rozdělena do mnohem více modulů než předchozí aplikace *Now in Android*. Samotné rozdělení do modulů je vidět na diagramu na obrázku 2.4. Toto je však pouze část celého diagramu a zachycuje pouze hlavní kostru celého stromu modulů. Ostatní moduly, které na diagramu chybí jsou pak do aplikace integrovány stejně jako typy modulů, které na diagramu znázorněny jsou. Typy nebo spíše úrovně modulů budou popsány v následujících odstavcích.



Obrázek 2.4: Rozdělení do modulů v Tivi

První takovou úrovní jsou opět *core* moduly. Jejich účel je však oproti předchozí aplikaci rozdílný. Tady to jsou menší moduly, které obsahují komponenty, které jsou potřebné napříč celou aplikací. Jako příklad je modul *core:base*, kde jsou pouze různé rozšiřující funkce a definované anotace, které usnadňují psaní kódu napříč celým projektem. Dalším příkladem může být modul *core:analytics*, ten zase obsahuje třídy pro logování různých událostí, tedy opět logika, která usnadňuje práci na celém projektu a je společná. Samotné *core* moduly na sobě mohou záviset, avšak nemají závislost na žádné

jiné moduly. Proto dochází k jejich sestavení jako první. Jejich velikost je velmi malá, proto jejich vzájemná závislost nepřináší žádný problém v době sestavení.

Další úrovní, a to konkrétně druhou, jsou moduly *data*. Ty však mohou sloužit pro různé účely. Modul *data:models* v sobě obsahuje všechny datové modely a díky tomu na něm závisí většina ostatních *data* modulů. Druhým účelem těchto modulů jsou moduly, které poskytují data z různých zdrojů. Většinou se jedná o zdroj API. Pro databázi je zde totiž zvláštní modul *data:db-room*, protože databáze *Room* vyžaduje, aby všechny třídy, které pro přístup k datům potřebuje, byly v jednom modulu, popřípadě modulech, na kterých tento modul závisí. V tomto případě jsou však všechny třídy v jednom modulu. Jak tedy vyplývá z popisu této úrovně modulů, moduly mohou na sobě záviset, avšak měla by být snaha, aby závisely co nejméně a zachovala se co nejnižší provázanost. Jinak moduly závisejí pouze na modulech první úrovně *core*.

Třetí úrovní je modul *domain*, ten je na této úrovni pouze jeden. Jeho úkolem je sjednocení většiny *data* modulů, které poskytují nějaká data a kombinovat je, popřípadě zde provádět nějakou doménovou logiku. Toto je možné uskutečnit, protože modul má k dispozici všechna data, která potřebuje. Avšak problémem toho modulu je, že při změně libovolného *data* modulu musí dojít k znovu sestavení modulu *domain*. V tomto případě modul obsahuje pouze 50 souborů `.kt`, který většinou obsahuje jednu třídu malé velikosti. Což pro dobu sestavení nebude mít takový velký dopad. Je však důležité brát v úvahu, že to v budoucnu může přivést problémy a bude třeba to řešit rozdělením do dalších modulů. Avšak to by neměl být problém a tato úroveň by mohla obsahovat několik modulů, které by na sobě nebyly závislé.

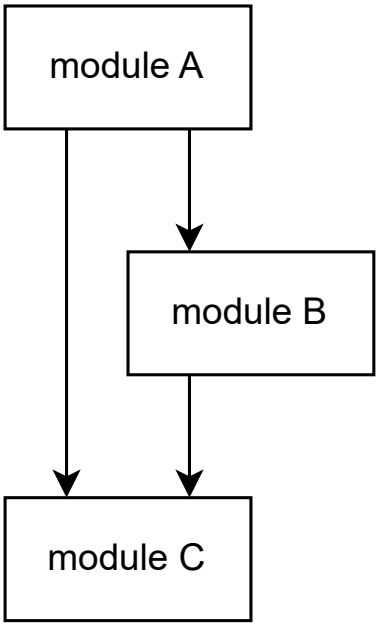
Jako další jsou zde moduly *common*, ty asi nelze nazvat jako úroveň, avšak podobně jako u aplikace *Now in Android* slouží jako místo, kde je nějaká společná logika. Jako například modul *common:ui*, kde je definice design systému a obsahuje společné UI komponenty, které se napříč celou aplikací používají. Tyto moduly nezávisejí na žádných jiných a díky tomu se nemusí sestavovat, pokud v nich nedojde k nějaké změně.

Další úroveň obsahuje moduly *ui*. Je možné je přirovnat k modulům *feature* z předchozí aplikace. Obsahují pouze definici UI a logiku, která je spojená s přípravou dat pro jednotlivé obrazovky. Platí pro ní to stejné jako pro *feature* moduly a to, že nejsou na sobě závislé. Závisí tedy pouze na modulech nižších úrovní. Vždy však na modulu *domain*.

Poslední úrovní je opět modul *app*, ten opět sjednocuje všechny moduly *ui* a nastavuje všechny potřebné komponenty pro funkčnost Android aplikace. Je tedy opět velmi minimalistický, protože k jeho znovu sestavení bude docházet téměř vždy.

Jednou ze strategií co se v projektu používá je *api–impl*. Díky ní dochází k optimalizaci času pro sestavení. Jako příklad si můžeme vzít tři úrovně modulů jako na obrázku 2.5. V tomto příkladě modul A závisí na modulech B a C.

Aby však bylo možné sestavit modul B, nejprve se musí sestavit modul C. To zapříčiní, že se sestavení nebude moci provádět paralelně a postupně se budou muset sestavovat v tomto pořadí: C, B a A.

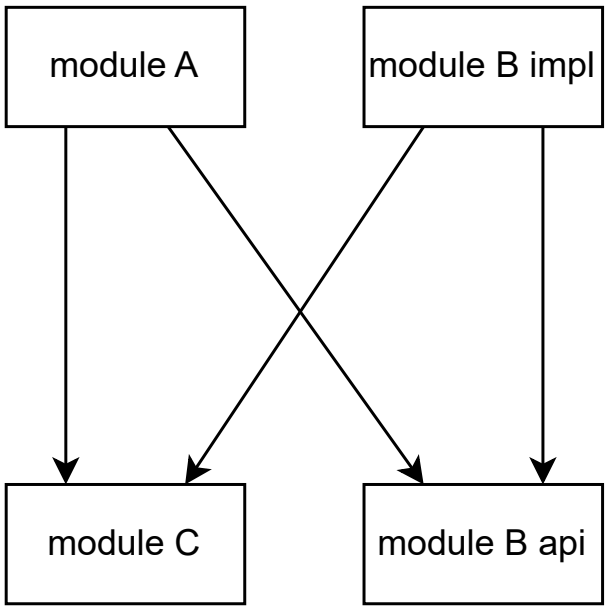


Obrázek 2.5: Ukázka strategie *api–impl* před optimalizací

To však lze optimalizovat a modul B rozdělit na dva, kde jeden pouze vystaví rozhraní. Takovému modulu se říká *api*. Ten druhý poté implementuje toto rozhraní. Samotné spojení se pak zajistí pomocí *dependency injection*. Na diagramu na obrázku 2.6 můžeme takovou optimalizaci vidět. Díky tomu se nám vytvořily dvě úrovně modulů. Pro jejich sestavení se nejprve paralelně sestaví moduly *C* a *B api* a poté opět paralelně *A* a *B impl*. Další výhodou je, že pokud se bude měnit implementace modulu *impl*, stačí sestavit pouze tento modul a zbytek se může při sestavení přeskočit, protože na modulu *impl* už žádný jiný nezávisí. Tato strategie však nelze používat ve všech případech. Musí být zajištěno, že modul *api* nepotřebuje záviset na žádném jiném modulu stejné úrovně, v tomto případě tedy není závislost na modulu *C*.

Tato strategie se například používá v projektu na modulech *data:db*, ten slouží jako modul *api*, a *data:db-room* a můžeme jej označit jako *impl*.

Tato aplikace v každém případě využívá více modulů než aplikace *Now in Android*. Je zde lépe řešené rozdělení *core* a *data* modulů, protože v předchozí aplikaci tyto moduly byly pouze označeny jako *core* a byly mnohem většího rozsahu. V této aplikaci jsou rozděleny podle jejich účelu. Například modul, který poskytuje pouze seriály atd. To zajišťuje oddělení jejich zodpovědností a zajišťuje, že se nemusí sestavovat znovu velké množství dalších modulů.



Obrázek 2.6: Ukázka strategie *api–impl* po optimalizaci

Projekt opět používal rozdělení do *ui* modulů, které jsou na sobě nezávislé a díky tomu je možné je sestavovat paralelně, popřípadě je přeskočit, pokud už sestavené byly. Tento vzor pro modularizaci je tedy něco, co by při návrhu mělo být zváženo a následně i použito.

Nevýhodou však může být modul *domain*, který závisí na velkém počtu modulů a z druhé strany na něm spoustu modulů závisí. Díky tomu se stává místem, které bude způsobovat zpomalení času sestavení. Ale na druhou stranu, jak již bylo řečeno, tento modul není velký a jediný větší dopad na sestavení bude způsoben moduly, které na něm závisí. Ty jsou však už na sobě nezávislé a budou se sestavovat paralelně. Závěrem by se tedy dalo říct, že toto řešení není ideální, ale rozhodně je přijatelné a všechny výhody modularizace jsou zachovány.

2.1.3 Návrh nového řešení

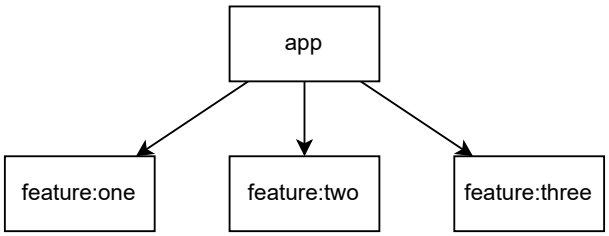
Na základě analýzy již existujících řešení se tato část bude věnovat návrhem a zlepšení modularizace pro aplikace Educhild. Protože aplikace již nějaké moduly obsahuje, musí se návrh provést tak, aby bylo možné ho jednoduše začít aplikovat a nemusela se aplikace celá od základu přepsat.

Základem každé aplikace je modul *app*. Pojďme tedy v návrhu začít s ním. Tento modul by měl být co nejmenší a pouze konfigurovat vše co je pro samotnou Android aplikaci nutné. Dále samozřejmě musí spojit všechny mo-

duly, které jsou pro aplikaci potřebné. Vydefinujme si tedy první krok návrhu modulů na obrázku 2.7.

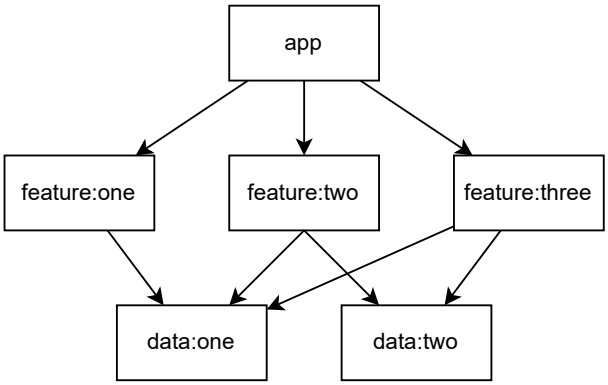
Obrázek 2.7: Návrh modularizace krok 1

Tak jako v obou aplikacích, se jednotlivé obrazovky definovaly v samostatných modulech. Ty nazvěme *feature* jako v aplikaci *Now in Android*. Tyto moduly tedy budou obsahovat pouze definici UI a logiky, které ke každé obrazovce přísluší. Moduly na sobě budou samozřejmě nezávislé, což je jednoduše splnitelné, když obsahují pouze tyto obrazovky. Součástí těchto modulů může být i navigace v rámci jednoho modulu. Pokud je potřeba komunikovat mezi těmito moduly, je potřeba to řešit před modul *app*. V kroku dva jsme tedy přidali moduly *feature* a můžeme ho vidět na obrázky 2.8.



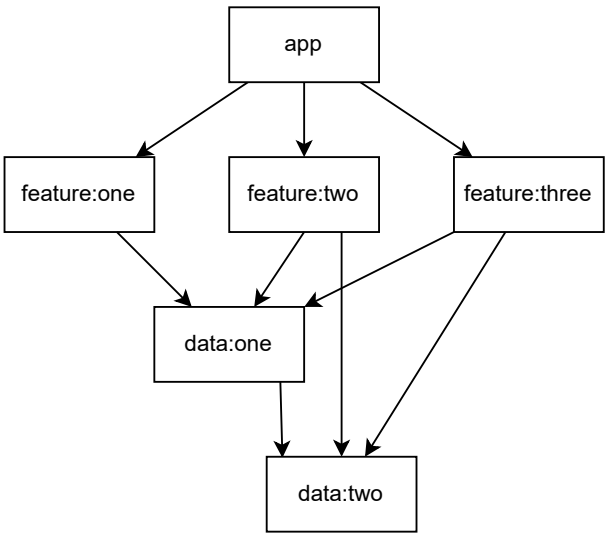
Obrázek 2.8: Návrh modularizace krok 2

Dalším krokem je přidat moduly, které budou poskytovat data, pro jednotlivé obrazovky. Tyto moduly nazvěme *data*, stejně jako v *Tivi* aplikaci. Moduly budou rozděleny podle jejich účelu, tedy například modul *data:quiz*, bude poskytovat všechna relevantní data ke kvízům. Obdobně by to fungovalo například pro moduly *data:questions* nebo *data:rewards*. Tyto moduly na sobě budou opět nezávislé a bude je tedy možné sestavovat paralelně. Stejně jako moduly *feature*. Diagram po tomto kroku je vidět na obrázku 2.9.



Obrázek 2.9: Návrh modularizace krok 3

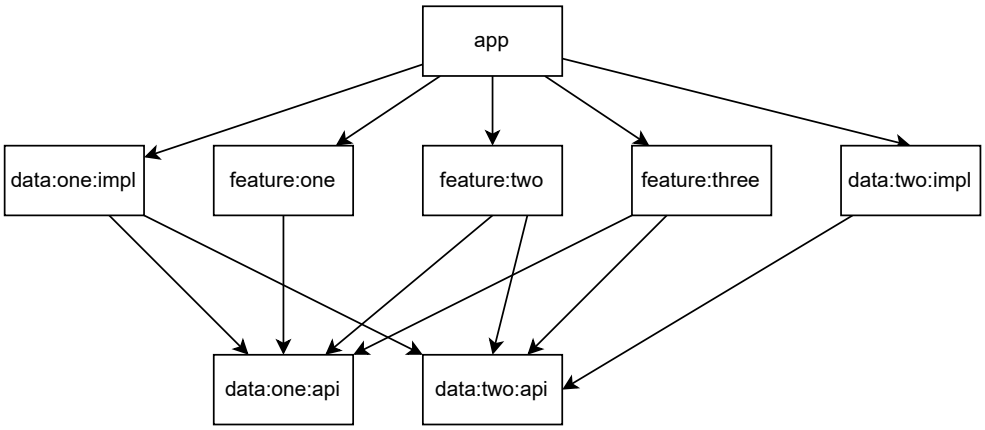
Stav po třetím kroku však není velice reálný. Jako příklad jak tento stav vyvrátit si můžeme vzít moduly *data:quiz* a *data:questions*. Pokud však budeme chtít vrátit kvízy spolu s otázkami najednou, budou muset tyto kvízy na sobě záviset, protože modul s kvízy neví, jak dostat kvízové otázky a je to zodpovědnost modulu *data:questions*. Dovolme tedy závislost mezi moduly *data*. Aplikace *tivi* tyto závislosti také povolovala. Po této úpravě dosáhneme stavu, který je no diagramu na obrázku 2.10.



Obrázek 2.10: Návrh modularizace krok 4

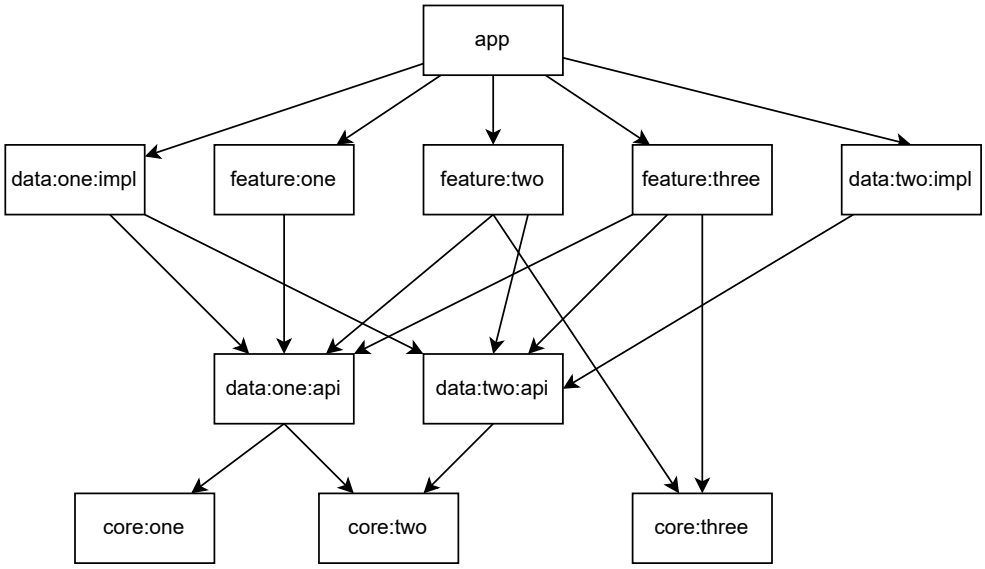
Po kroku 4 však můžeme aplikovat optimalizaci *api-impl*, která byla popsána v předchozí kapitole. Díky tomu se hloubka stromu zmenší a optimalizuje se i samotný čas pro sestavení aplikace. V tomto případě to dává naprostý

smysl a je to i umožněné, protože moduly *data* potřebují závislost na sobě pouze pro získání dat, avšak stačí jim na to znát pouze rozhraní a nic jiného. Na modulech *impl* bude pak záviset modul *app*, aby bylo možné konkrétně nastavit *dependency injection* a implementace rozhraní se správně spojila. Tato optimalizace je aplikována v kroku na obrázku 2.11. Díky tomu se zmenšila hloubka stromu a navíc moduly *data api* ani moduly *data impl* na sobě nejsou závislé.



Obrázek 2.11: Návrh modularizace krok 5

Jako poslední nám zbývá přidat základní modely *core*. Mezi ty budou patřit například moduly *core:design*, *core:network* nebo *core:persistance*. Tyto moduly budou konfigurovat design systém a jednotlivé komponenty sdílené napříč všemi obrazovkami. Dále budou nastavovat síťovou vrstvu aplikace, tedy komunikaci s API a to samé pro lokální databázi. V modulu pro databázi pak budou všechny entity a ostatní třídy, které knihovna pro databáze v Android aplikacích vyžaduje. U modulů *core* by opět mělo být možné, aby na sobě nezávisely. Výsledný návrh je vidět na diagramu na obrázky 2.12.



Obrázek 2.12: Návrh modularizace krok 6

2.1.3.1 Integrace nového návrhu do stávajícího řešení

Jak již bylo řečeno, aktuálně má aplikace několik modulů. Pro implementaci vytváření kvízů by tedy bylo ideální, aby se aktuální implementace nemusela celá měnit a dala se pouze o tuto novou funkci rozšířit již s novým návrhem modularizace.

Aby to odpovídalo návrhu z předchozí části, můžeme následovně přejmenovat moduly.

- *model* na *core:model*
- *network* na *core:network*
- *persistance* na *core:persistance*

Závislost mezi těmito moduly bude zachována, aby se nenarušila již existující implementace.

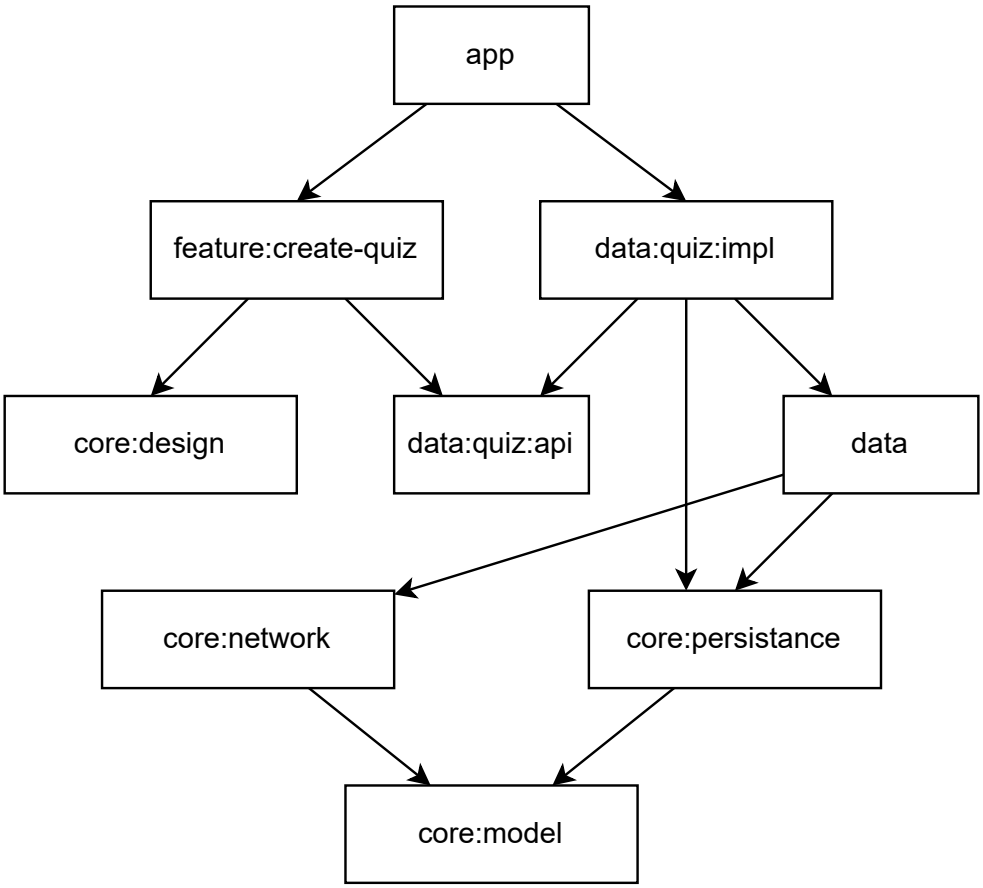
Pro novou funkci můžeme vytvořit dva nové moduly *data:quiz:api* a druhý *data:quiz:impl*, které budou poskytovat data týkající se kvízů a jejich otázek. Díky tomu bude odpovědnost týkající se kvízů pouze v těchto modulech a ostatní moduly, které budou tyto data potřebovat, stačí, když budou záviset na modulu *data:quiz:api*.

Stávající modul *data* zůstane zachován, avšak jeho obsah by měl být postupně nahrazován menšími moduly, jako je nový *data:quiz:**.

Dále určitě přibude nový modul *core:design*, který bude definovat společné UI komponenty pro všechny *feature* moduly.

Nakonec přidáme modul pro obrazovky, jenž slouží pro tvorbu nových kvízů *feature:create-quiz*.

Výsledný návrh pro rozšíření je vidět na obrázku 2.13. Toto rozšíření kromě přejmenování modulů, aby lépe odpovídaly návrhu, nevyžaduje žádný zásah do samotného kódu aplikace. Samozřejmě, aby to odpovídalo původnímu návrhu, je třeba eliminovat modul *data* a nahradit ho moduly menšími. Modulu *core:network* a *core:persistance* pak zredukovat, aby obsahovaly jen to nutné a přesunout je do *data* modulů pro správné oddělení zodpovědnosti. Nakonec úplně eliminovat modul *core:model* a všechny datové třídy taktéž přemístit do modulů *data*.



Obrázek 2.13: Finální návrh modularizace v projektu

2.2 Doménový model

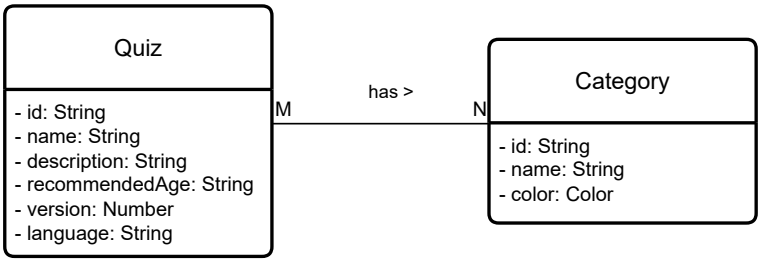
Tato část se bude zabývat návrhem doménového modelu se zaměřením na kvízy a jejich samotnou tvorbu. Protože se s kvízy už v aplikaci pracuje, projekt už třídy pro kvízy obsahuje. Avšak ty nejsou navrženy tak, aby bylo možné kvízy vytvářet, proto je bude nutné rozšířit o další atributy, které to umožní.

Vytvoření doménového modelu je metoda popisu a modelování entit a vztahů mezi nimi. Ty společně představují samotnou doménu a její problematiku. Zabývá se identifikací doménových entit a jejich vztahů, ta je odvozená z pochopení požadavků na systém. Díky tomu poskytuje základ pro pochopení a následný návrh systémů. [9]

Samotný doménový model bude rozdělen do dvou částí a to do části s kvízem a poté do části s kvízovými otázkami. Díky tomu budou diagramy více přehledné.

2.2.1 Část kvízu

Tato část doménového modelu se tedy bude zabývat pouze kvízem. Diagram je vidět na diagramu na obrázku 2.14.



Obrázek 2.14: Doménový model kvízu

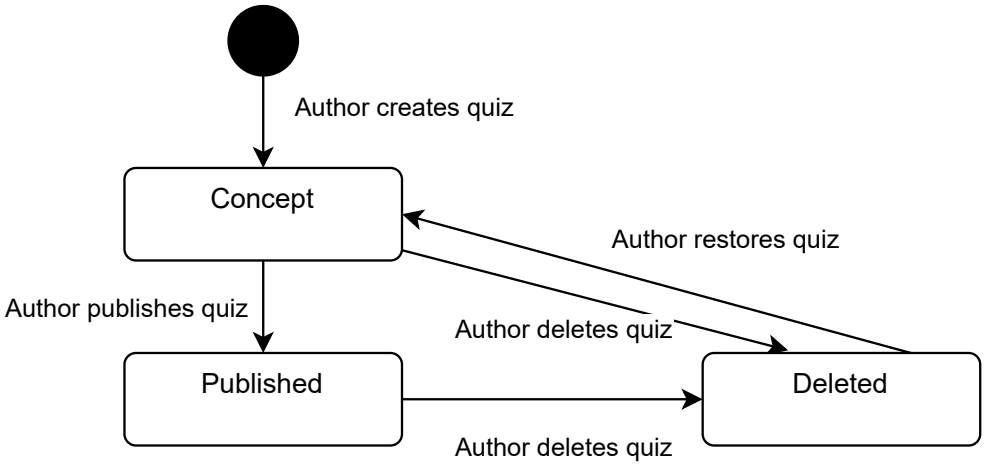
2.2.1.1 Kvíz

Entita reprezentuje kvíz. Ten může uživatel vytvářet, upravovat či mazat. Zároveň se zobrazuje dítěti, které ho může vyplňovat. Kvíz má několik atributů ty jsou popsané v tabulce 2.1.

Kvíz může nabývat dvou typů stavů. První typ určuje samotný jeho stav vytvoření. Stavový diagram lze vidět na obrázku 2.15. Kvíz po jeho vytvoření má stav koncept. Ten může jeho autor změnit na publikován, když si myslí, že je kvíz připraven, aby si ho mohli stáhnout ostatní uživatelé. V obou těchto stavech ho může ale i smazat. Smazaný kvíz je poté možné obnovit a tím se dostane zpět do stavu konceptu a uživatel ho může dále publikovat.

Název	**Popis**
id	Identifikátor kvízu
name	Jméno kvízu
description	Popis obsahu
recommendedAge	Doporučený věk pro dítě
version	Aktuální verze kvízu
language	Jazyk kvízu

Tabulka 2.1: Atributy kvízu

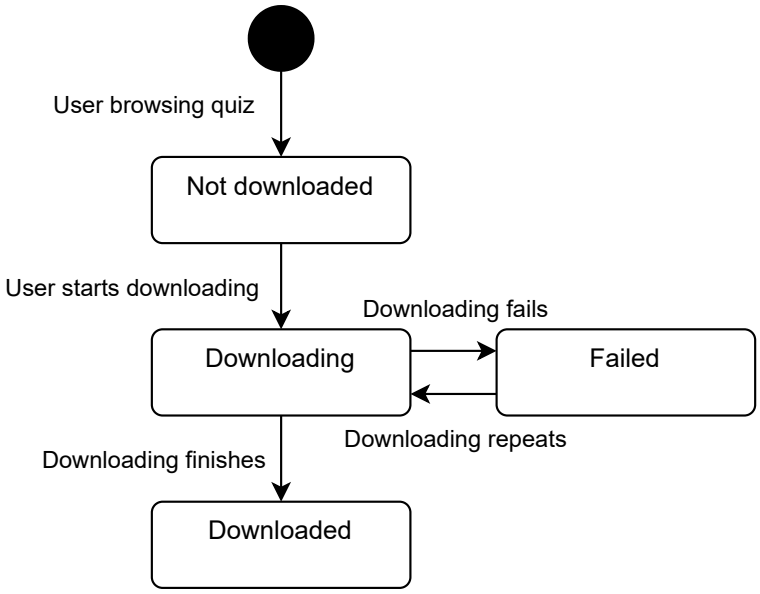


Obrázek 2.15: Stavový diagram vytvoření kvízu

Druhým typem stavu je stav stažení kvízu, protože je možné kvízy stahovat, aby bylo možné s nimi pracovat i v offline režimu. Proto je nutné reprezentovat stav tohoto stažení. Díky tomu poté například zobrazovat pouze úspěšně stažené kvízy. Uživatel si prohlíží dostupné kvízy, ty mají stav jako nestažené. Pokud se nějaký rozhodne stáhnout, změní se stav na právě stahované a podle výsledku stažení pak na úspěšně či neúspěšně stažené. Stavový diagram pro tento stav je vidět na obrázky 2.16.

2.2.1.2 Kategorie

Kategorie kvízů reprezentuje entita, která má pouze tři atributy. Jsou jimi identifikátor, jméno a barva, která danou kategorii reprezentuje.



Obrázek 2.16: Stavový diagram stažení kvízu

2.2.2 Část kvízové otázky

Druhá část doménového modelu popisuje kvízové otázky a je vidět na diagramu na obrázku 2.17.

2.2.2.1 Kvíz

Tato entita byla popsána v části 2.2.1. V diagramu je entita pouze, protože popisuje, že kvíz obsahuje více otázek. Nemusí obsahovat však otázku žádnou. To je validní stav, protože po vytvoření kvízu žádné otázky nemá a uživatel je musí nejprve vytvořit.

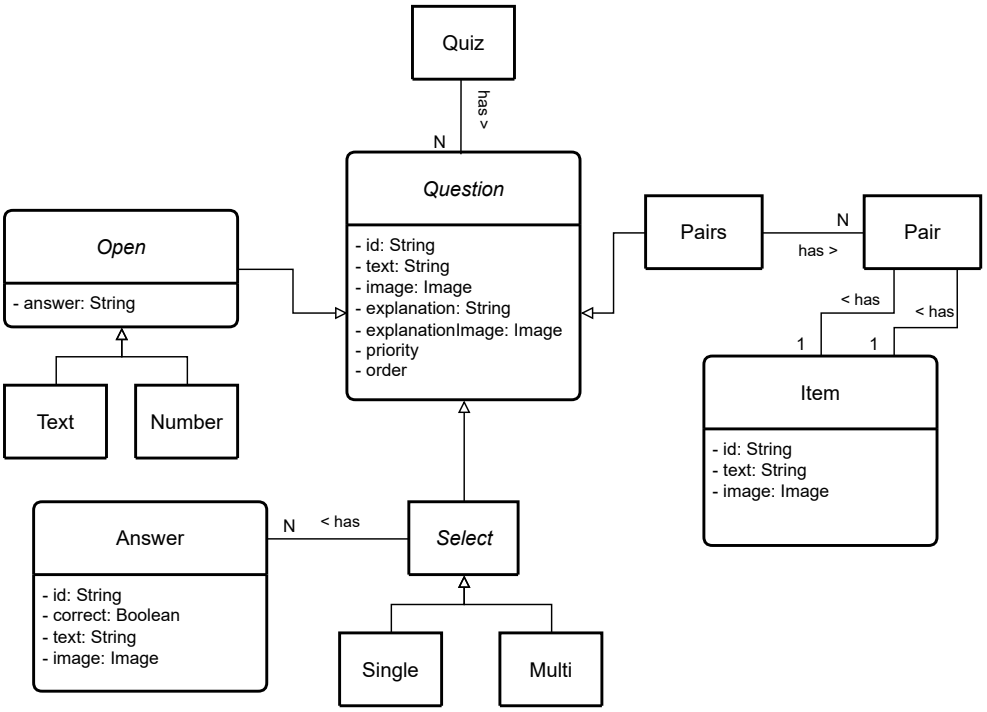
2.2.2.2 Otázka

Entita reprezentující kvízovou otázku. Jak je v diagramu naznačeno, ta může být třech typů: otevřená, výběr a spojování párů. Každý tento typ však má společné atributy, popsané v tabulce 2.2.

2.2.2.3 Otevřená

Podtyp otázky popisující otevřenou otázku. Ta má navíc jeden atribut a to `answer`, který obsahuje správnou odpověď.

Tyto entity můžeme ještě rozdělit na dva podtypy, kde podtyp text reprezentuje textovou odpověď, tedy odpověď, která může obsahovat libovolné



Obrázek 2.17: Doménový model kvízové otázky

Název	**Popis**
id	Identifikátor otázky
text	Textové zadání otázky
image	Obrázkové zadání otázky
explanation	Textové vysvětlení odpovědi
explanationImage	Obrázkové vysvětlení odpovědi
priority	Priorita, podle které se zobrazuje otázka dítěti. Mění se podle toho, zda dítě odpovídá na otázku špatně.
order	Pořadí v jakém byla otázka vytvořena

Tabulka 2.2: Atributy kvízové otázky

znaky. Více striktní je pak typ číselné odpovědi, ten rozlišuje, jaký vstup dovolí uživateli zadávat.

2.2.2.4 Výběr

Pro typ výběru z více možností slouží podtyp `Select`. Ten obsahuje více odpovědí, ta je popsána entitou `Answer`.

Entita pro samotné odpovědi obsahuje několik atributů, kromě identifikátoru je to textové znění odpovědi společně s obrázkovou reprezentací a nakonec příznak, zda je daná odpověď správně.

Výběr z více možností je možné opět oddělit na dva typy. Jeden, kde je pouze jedna možnost správně, a druhý, který povoluje více správných odpovědí. Díky tomu, je pak uživateli možné zobrazit správné UI komponenty.

2.2.2.5 Spojování párů

Spojování párů, popřípadě dvojic, definuje entita `Pairs`. Každá entita může obsahovat více párů `Pair`. To reprezentuje, jak správně spojené dvojice vypadají. Prvek této dvojice je pak popsán pomocí entity `Item`. Ta obsahuje opět identifikátor, textovou a obrázkovou reprezentaci. Jak z názvu vyplývá tyto prvky mohou být v páru pouze a vždy jen dva.

2.2.2.6 Obrázky

Protože otázky mohou obsahovat několik obrázků, je potřeba tyto obrázky nějak reprezentovat. Avšak obrázek může být několika typů.

Při tvorbě kvízových otázek uživatel vybírá obrázky z úložiště svého zařízení. K tomu slouží typ `Picked`, který má atribut `path` určující cestu k vybranému obrázku. Tato cesta je důležitá pro nahrání obrázku na server, poté se již stane irelevantní. Proto uživatel může obrázek v telefonu přesunout, popřípadě odstranit a na otázku to nebude mít žádný vliv.

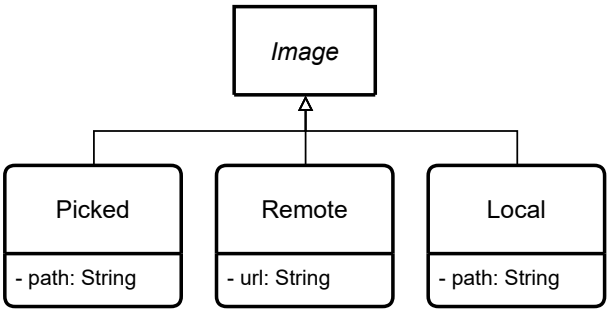
Po nahrání na serveru jsou u vrácených otázek obrázky typu `Remote`. Tedy obrázek je definovaný pomocí URL a pro jeho zobrazení je potřeba mít připojení k internetu. Tento stav je tedy před tím, než dojde s samotnému stažení otázky, protože je potřeba, aby byly dostupné i v offline režimu.

Třetím typem, který bude použit u všech stažených otázek a jsou dostupné offline, je `Local`. V tomto stavu je obrázek v interním úložišti aplikace a je dohledatelný pomocí atributu `path`.

Diagram pro všechny typy obrázky je vidět na obrázku 2.18.

2.3 Uživatelské rozhraní

Samotný návrh pro uživatelské rozhraní nové funkce, včetně jeho testování, probíhal v rámci předmětu *NI–NUR*, proto se jím nebudeme v této práci zabývat.



Obrázek 2.18: Doménový model obrázku

2.4 Shrnutí

Výstupy této kapitoly budou sloužit k následné implementaci.

Prvním výstupem je návrh modularizace s popisem, jak aktuální stav projektu rozšířit o novou funkci. Díky tomu by mělo dojít ke zlepšení kvality celého Android projektu. Samotné přidání funkce vytváření kvízů, by mělo být oddělené od zbytku celého projektu a díky tomu by mělo jít ověřit, zda je návrh správný a modularizace přináší všechny výhody, které byly definované.

Výstupem číslo dva je doménový model, který popisuje kvízy spolu s kvízovými otázkami. Na základě toho bude jednoduché tuto doménu převést do aplikace a implementovat tak všechny potřebné funkcionality definované na začátku této práce.

Kapitola 3

Implementace

Kapitola o implementaci se bude zabývat tím, jak je nová funkce implementována.

Nejprve krátce představí technologie, které jsou v projektu již použité. Avšak pro implementaci byly vybrané i technologie pro projekt zcela nové. Důvod jejich použití bude popsán v sekci, která se jim bude věnovat. V jejich popisu nebude chybět ani ukázka toho, jak je použitá pro konkrétní případy.

V této kapitole nebude chybět ani ukázka realizace modulů z předchozího návrhu.

Na závěr bude popsán postup pro automatické publikování na Google Play, které bude probíhat na *CI*.

Výstupem této kapitoly tedy bude hotová nová funkce, která bude připravená pro testování. Tomu se bude tato práce věnovat v následující kapitole.

3.1 Technologie

Android aplikace používá několik technologií nebo knihoven. Ty hlavní, které budou v rámci rozšíření o novou funkcionalitu použity, budou popsány v této sekci.

3.1.1 Kotlin

Na konferenci Google I/O v roce 2019 byl oficiálně Kotlin potvrzený jako primární jazyk pro vývoj na zařízení s operačním systémem Android.

To pro vývoj přináší hned několik výhod.

- Psaní méně kódu, které je zkombinované s lepší čitelností.
- Stabilní prostředí a vyspělý jazyk. Od roku 2011 se jazyk neustále vyvíjí a je postupně integrován do mnoha aplikací.

- Podpora v Android knihovnách. Ty využívají specifické funkce jazyka a usnadňují práci s nimi.
- Interoperabilita s Javou
- Podpora multiplatformního vývoje. Jazyk lze využít pro vývoj nejen Android aplikací, ale i iOS, backend a webové aplikace. Využívá sdílení společného kódu a díky tomu je možné psát nativní aplikace se stejným základem a logikou.

[10]

3.1.2 Coroutines

Korutiny jsou návrhový vzor pro souběžně běžící kód. Usnadňují tak použití a zjednodušují kód, který běží nezávisle na sobě.

V Android pomáhají spravovat dlouho trvající úlohy, které by mohly zablokovat hlavní vlákno a tím způsobit, že aplikace nebude reagovat. Použití však přináší více funkcí.

- Lehkost. Díky podpoře pozastavení, které neblokuje vlákno, v němž korutina běží, může spouštět více korutin na tom samém vlákně. Pozastavení tak šetří paměť oproti blokování a zároveň podporuje spouštění mnoho souběžných operací.
- Vystavěná podpora rušení. Zrušení se automaticky šíří hierarchií běžících korutin a díky tomu je zajištěno, že dojde k zrušení i jich. Díky tomu se zmenšuje šance na unik paměti.
- Integrace s Android knihovna, které poskytují plnou podporu korutin a díky tomu je možné je ve zbytku aplikace snadno používat.

[11]

3.1.3 Retrofit

Pomocí této knihovny je jednoduché reprezentovat API pomocí rozhraní, které je popsané pomocí anotací.

Retrofit poté vygeneruje implementaci tohoto rozhraní a díky tomu je možné odesílat jednotlivé požadavky.

Samozřejmostí je podpora korutin pro asynchronní zpracování a konvertorů, které například převádí datové objekty do JSON reprezentace. Tyto konvertory lze poskytnout vlastní nebo využít již připravených, které používají nejrozšířenější knihovny pro serializaci a deserializaci. [12]

Ukázka toho, jak reprezentovat API je vidět na ukázce kódu 6. V této ukázce dojde k definici API, které vytváří z uživatele nového autora. Obsahuje metodu požadavku, jeho cestu a jako tělo se posílá datový objekt, který obsahuje všechny potřebné atributy.

```
internal interface AuthorApiDescription {

    @POST("authors")
    suspend fun createAuthor(
        @Body apiAuthor: ApiAuthor
    ): ApiAuthor
}
```

Výpis kódu 6: Ukázka vytvoření API definice pomocí Retrofit

3.1.4 Room

Room je knihovna, která poskytuje abstrakční vrstvu nad SQLite, jenž umožňuje plynulý přístup k databázi a zároveň využívá jeho plný výkon. Mezi výhody této knihovny patří následující.

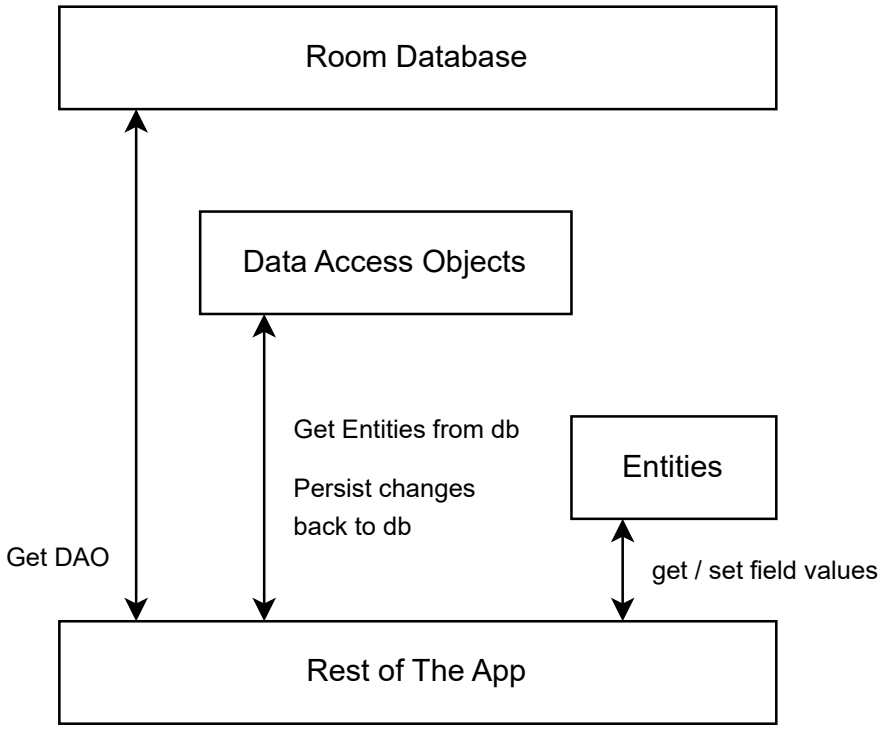
- Ověřování dotazů SQL v době kompilace.
- Anotace, které minimalizují opakující se kód.
- Zjednodušení migrací databází.

Hlavními komponenty pro přístup do databáze jsou následující.

1. Třída databáze, ta udržuje databázi a slouží jako hlavní přístupový bod k dalším vrstvám.
2. Entity jsou třídy, které reprezentují tabulky v databázi.
3. DAOs poskytují metody, které může aplikace používat k listování, vkládání, úpravě nebo vymazání dat v databázi.

[13]

Samotný diagram pro jednotlivé komponenty ilustruje obrázek 3.1.



Obrázek 3.1: Diagram architektury Room knihovny [14]

3.1.5 Koin

Koin je pragmatický a jednoduchý framework pro *dependency injection* v jazyce Kotlin. [15]

Tato knihovna používá vytváření definic v modulech. Jako modul se zde považuje prostor, kde se deklarují všechny komponenty. Tyto moduly jsou funkce.

Jednotlivé komponenty mohou být *Singletony*, to znamená, že kontejner bude udržovat unikátní instanci deklarovaného komponentu. Tyto instance se však vytváří až po prvním dotazu na komponentu.

Dalším příkladem komponenty je `factory`. Ta zaručuje, že při každém dotazu pro danou definici se vytvoří nová instance. Tyto instance pak nejsou uloženy uvnitř Koin kontejneru a nejsou tedy přepoužité znovu. [16]

Ukázka jak se vytváří modul, který definuje oba typy těchto komponent, je vidět na výpisu kódu 7.

Díky této knihovně bude následně jednoduché zajistit funkčnost návrhu modulů. Zejména pak optimalizaci *api–impl*. Modul *impl*, bude vystavovat

tento modul, který se přidá jako definice a Koin bude poté schopný dodat správnou instanci pro rozhraní z modulu *api*.

```
val dataAuthorModule = module {
    factoryOf(::ApiAuthorMapper)

    single {
        get<Retrofit>().create(AuthorApiDescription::class.java)
    }
    singleOf(::RetrofitAuthorDataSource) {
        bind<NetworkAuthorDataSource>()
    }

    singleOf(::AuthorRepositoryImpl) {
        bind<AuthorRepository>()
    }
}
```

Výpis kódu 7: Ukázka vytvoření modulu pomocí Koin

3.2 Gradle

Gradle se sice v projektu již používá, protože je to standardní cesta, jak sestavovat Android aplikace. Avšak, aby bylo možné implementovat návrh modularizace, je nutné tuto část značně změnit.

Gradle je nástroj pro automatizaci sestavení, který je dostatečně flexibilní pro sestavení téměř jakéhokoliv typu softwaru. Gradle má jen málo předpokladů o tom, co se snaží sestavit nebo jak to udělat. Díky tomu je však velmi flexibilní.

Samotný Gradle běží na JVM prostředí a díky tomu může běžet na různých platformách. A díky tomu že JVM používá i Java či Kotlin, je to jasná volba pro použití k sestavení Android aplikací.

Gradle používá následující terminologii k oddělení různých typů. Prvním typem je projet. To je věc, kterou gradle sestavuje. Každý tento projekt je popsán pomocí skriptu v souboru `build.gradle`. Tento skript definuje úlohu, závislosti, pluginy a další konfigurace pro daný projekt. Každé sestavení se však může skládat z jednoho nebo více projektů a každý takový projekt může obsahovat více podprojektů.

Druhým typem je úkol (Task), který obsahuje logiku, vykonávající nějakou práci. Jako příklad si můžeme uvést kompilaci kódu, spouštění testů nebo například deployment softwaru. Ve většině případů se používají již existující úkoly. Samotný Gradle poskytuje několik společných, jenž sestavovací systém potřebuje, jako například Java testovací úkol, která umí spouštět testy. Další úkoly pak poskytují pluginy. Každý úkol má následující strukturu.

- Akce, to jsou části práce, které něco vykonávají
- Vstupy (hodnoty, soubory, složky), se kterými akce pracují
- Výstupy (soubory, složky), které akce generují

Díky jednotlivým vstupům a výstupům lze úkoly řetězit a následovně i ověřovat, zda je nutné je spouštět v případě, kdy už jednou se stejnými parametry běžely.

Třetí typ, který Gradle nabízí, jsou pluginy. Ty umožňují přinést do sestavování nové koncepty nad rámec úloh, souborů a konfigurací. Například většina pluginů pro programovací jazyky přidává do sestavení koncept sad zdrojových kódů. Pluginy poskytují možnost přepoužití logiky napříč více projekty. S pluginy je možné napsat úlohu jednou a použít ji ve více sestavení, nebo například provést na jednom místě společnou konfiguraci a opět jí přepoužívat. Tím se sníží duplicita v jednotlivých skritech v každém modulu. Díky tomu se zjednodušuje použití a zvyšuje se efektivita.

Samotné sestavení má tři fáze životního cyklu.

1. Inicializace
2. Konfigurace
3. Exekuce

Během inicializace se nastavuje prostředí pro sestavení a určuje se, které projekty se mají do sestavení zahrnout. V této fázi je důležitý soubor se skriptem `settings.gradle`, který definuje, jaké projekty mají být součástí samotného sestavení. Ukázka toho, jak v projektu takový soubor vypadá, je vidět ve výpisu 8.

Fáze konfigurační sestavuje a nastavuje graf úkolů potřebných pro sestavení. Určuje, které potřebují být spouštěny a v jakém pořadí. To vše na základě toho co uživatel chce sestavit a na základě jednotlivých skriptů, které jsou v každém projektu.

Poslední fáze je exekuční, ta už pouze spouští jednotlivé úkoly na základě toho, co bylo výstupem z fáze předchozí. [17]

3.2.1 Konvenční pluginy

V případě, kdy jednotlivé projekty používají společnou logiku pro konfiguraci sestavení, je doporučené tuto logiku přesunout do společného pluginu. Díky tomu se sníží duplicita a pluginy se zjednoduší. Skripty by tedy měli ve finále obsahovat pouze specifickou konfiguraci a vše společné by mělo být v těchto konvenčních pluginech. [18]

```
pluginManagement {
    ...
}

dependencyResolutionManagement {
    ...
}

rootProject.name = "EduChild"

include ":app"

include ":core:architecture"
include ":core:design"

...
```

Výpis kódu 8: Ukázka souboru settings.gradle

3.2.2 Precompiled pluginy

Konvenční pluginy z předchozí části lze implementovat jako takzvané *precompiled pluginy*. Díky nim je možné psát Gradle pluginy a tím poskytovat logiku pro sestavení v *Groovy* nebo *Kotlin* skriptovacích jazycích.

Stačí tedy vytvořit soubory s koncovkou `.gradle` do struktury složek `src/main/groovy`. Tyto skripty jsou poté zkompilovány do *class* souborů a následovně zabaleny do *jar*. Poté jsou v ostatních projektech dostupné jako pluginy a lze je použít podle jejich unikátního identifikátoru. Tento identifikátor je určený podle názvu souboru, který skript obsahuje. [19]

Například, pokud se soubor se skriptem jmenuje `build-app.gradle`, tak výsledný plugin bude mít id `build-app`. Podle něj ho lze poté aplikovat do jednotlivých projektů, to lze vidět v ukázce 9.

```
plugins {
    id("build-app")
    ...
}
```

Výpis kódu 9: Ukázka aplikování pluginu

3.2.3 Composite build

Composite build je způsob, jak docílit toho, aby jeno sestavení mělo k dispozici jiné sestavení. Dalo by se přirovnat k sestavení, které obsahuje více projektů,

až na výjimku, že composite build už je zahrnut do všech ostatních.

Díky tomu je konfigurovaný a sestavený izolovaně. To přináší výhodu, že pokud v něm nedojde k žádně změně, nemusí se sestavení spouštět znovu.

Samotné nastavení takového sestavení je jednoduché, stačí přidat projekt, nazvěme ho například `build-logic`. Ten musí obsahovat `settings.gradle` a `build.gradle` soubory. Díky tomu se Gradle bude chovat k tomuto projektu jako k samostatnému. Pro přidání jako composite, je potřeba v kořenovém `settings.gradle` toto konfigurovat stejně jako na ukázce 10. Poté budou všechny pluginy z tohoto projektu dostupné ve všech ostatních. [20]

```
pluginManagement {
    includeBuild("build-logic")
    repositories {
        ...
    }
}
```

Výpis kódu 10: Nastavení composite build v settings.gradle

3.2.4 Version catalog

Version catalog je centrální seznam závislostí ze kterých může uživatel vybírat při deklarování závislosti v skriptech sestavení.

Výhody použití této funkce v Gradle jsou následující.

- Pro každý katalog se generují typově bezpečné přístupy, takže je možné snadno přidávat závislosti pomocí automatického doplňování v IDE
- Katalogy jsou viditelné ve všech projektech sestavení. Je to centrální místo, kde lze deklarovat verzi závislostí a zajistit, aby se změna této verze vztahovala na všechny podprojekty.
- Podpora balíčků závislostí, což je skupina, které se běžně používá současně

Pro vytvoření takového katalogu, stačí přidat soubor `libs.versions.toml` do složky `gradle`. Struktura a příklad takového katalogu je ve výpisu kódu 11. Přístup v jednotlivých skriptech je poté přes `libs.{název_závislosti}`. [21]

```
[versions]
kotlin = "1.8.20"
...

[libraries]
kotlin-stdlib = {
    module = "org.jetbrains.kotlin:kotlin-stdlib",
    version.ref = "kotlin"
}
...

[plugins]
kotlin-android = {
    id = "org.jetbrains.kotlin.android",
    version.ref = "kotlin"
}
...
```

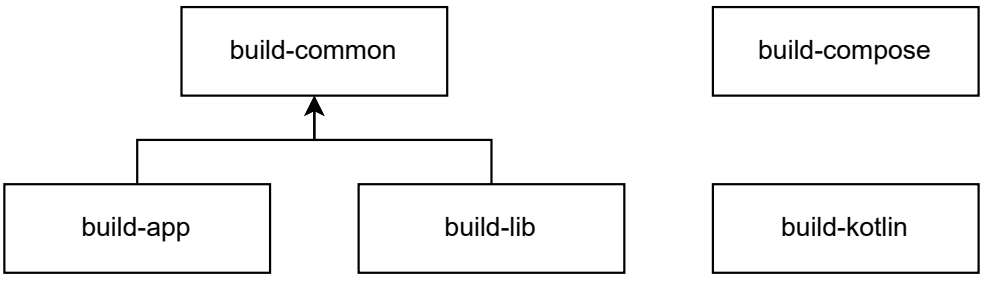
Výpis kódu 11: Ukázka version katalogu

3.2.5 Použití

Nejprve pro lepší strukturu skriptů byl vytvořen *version catalog* a všechny závislosti se přidávají pomocí něj. Díky tomu je jednodušší spravovat jednotlivé verze závislostí a snadno je aktualizovat.

Dále byly vytvořeny konvenční pluginy a to za pomocí precompiled pluginů. Ty se nacházejí v `build-logic` modulu, který je nastavený jako composite build, aby využíval všech vlastností, které byly dříve uvedeny.

Díky tomu bylo možné navrhnout strukturu pluginů tak, jak je vidět na diagramu na obrázku 3.2.



Obrázek 3.2: Diagram gradle pluginů

3.2.5.1 build-common

Obsahuje společnou logiku pro nastavení jakéhokoliv Android projektu. V případě této aplikace to je aplikace a knihovna. Tento plugin přidává další pluginy potřebné pro vývoj Android aplikace, například Kotliny pro Android.

Konfiguruje verze Androidu, pro které je aplikace určena, tedy minimální a cílená. Nastavuje verzi Javy, konfiguruje různé prostředí API a nakonec přidává závislosti potřebné v každém modulu.

3.2.5.2 build-app

Přidává plugin pro Android aplikaci a náš plugin `build-common`. Dále nastavuje další specifické věci pro aplikaci, jako například podepisování a typy sestavení.

3.2.5.3 build-lib

Obdobně jako předchozí, tento plugin přidává plugin pro knihovnu a plugin pro společnou logiku `build-common`. Víc tento plugin nekonfiguruje, avšak kdyby bylo třeba něco pro všechny knihovny změnit, je možné to udělat v tomto pluginu a ihned se to použije u všech projektů, které ho používají.

3.2.5.4 build-compose

Plugin, který konfiguruje Jetpack Compose. To vyžaduje nastavení verze kompilátoru a přidání několika závislostí. Všechny moduly, které pak budou používat *Jetpack Compose*, aplikují tento plugin a mohou jej využívat. Typicky se bude jednat o moduly typu *feature* z návrhu o modularizaci.

3.2.5.5 build-kotlin

Plugin konfigurující pouze samotný Kotlin. Ten je určený pro moduly, které nepotřebují závislost na Android knihovnách a tím pádem nemusí být považovány jako Android projekty. Díky tomu se sníží čas jejich sestavování. Tento plugin je nejjednodušší a nastavuje pouze příslušný plugin a verzi Javy. Ukázka tohoto pluginu je vidět na výpisu 12.

3.3 Jetpack Compose

Jetpack Compose je moderní sada nástrojů pro vytváření nativního UI v Androidu. Zjednodušuje a zrychluje samotný vývoj uživatelského rozhraní za pomocí intuitivního API v Kotlinu. Samozřejmostí Compose je jeho Interoperabilita s definicí UI pomocí XML. [22]

```
plugins {
    id("org.jetbrains.kotlin.jvm")
}

java {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

Výpis kódu 12: Ukázka pluginu build-kotlin

Compose byl pro tuto novou funkci vybrán obzvláště kvůli jeho popularitě a jednoduchosti pro psaní složitého UI. To bude velmi užitečné pro nové obrazovky. A díky své interoperabilitě ničemu nebrání ho do projektu integrovat.

V této části bude popsáno jak samotný Compose funguje a dále jak je použit pro implementaci funkce tvorby kvízů.

3.3.1 Popis

Rozdělení zodpovědností je dobře známý princip. Tento princip však není aplikován pro starší definování UI pomocí XML. To vyžaduje znalost samotného obsahu obrazovku už ve *ViewModelu*. Jako příklad toho může být referencování jednotlivých komponent pomocí `findViewById`.

Používání těchto API vyžaduje znalost toho, jak je XML rozvržení definováno a vytváří mezi nimi vazba. Pokud pak dojde ke změně v XML, může dojít k porušení těchto vazeb a vzniknou z toho problémy v podobě výjimek.

Pokud by však bylo UI definované pomocí stejného jazyka jako zbytek aplikace, mohlo by dojít ke zlepšení. Toto nabízí právě Compose, který pomocí jednotlivých *Composable* funkcí nabízí nástroj pro rozdělení těchto odpovědností.

Composable funkce může přijímat nějaké parametry a sama o sobě může volat další *Composable* funkce. Tato volání reprezentují uživatelské rozhraní v samotné hierarchii. Výhodou používání Kotlinu je možnost použití jakýchkoliv konstrukcí. To znamená, že můžeme například používat podmínky nebo cykly, pro vytvoření složitého UI.

Compose tedy využívá deklarativního psaní uživatelských rozhraní. To zajišťuje, že pokud se nějaké *Composable* funkce nezavolá, nebude její obsah vykreslen. Není tedy potřeba řešit, v jakém stavu byla předchozí fáze obrazovky a vždy se zobrazuje to, co je potřeba pro stav aktuální. Samotný přechod mezi stavy zajišťuje framework sám od sebe.

Všechny *Composable* funkce mohou být zavolány v jakýkoliv čas. Pokud je například použita velmi rozsáhlá hierarchie těchto funkcí a je potřeba část změnit, dojde k přepočítání pouze té části, která je potřeba. To je zajištěno tím, že funkce se můžeš libovolně přeskakovat nebo restartovat. To přináší

mnoho výhod z hlediska výkonnosti. O tom, kdy k přeskakování či restartování bude docházet rozhoduje samotný framework. [23]

3.3.2 Rekompozice

Ke *Composable* funkcím kompilátor během sestavení přidá objekt `Composer`. Ten obsahuje datovou strukturu *Gap Buffer*. Tato struktura reprezentuje kolekci s aktuálním indexem. Je implementována v paměti pomocí pole. Samotné pole je potom větší než data, které reprezentuje, zbytek je označený jako mezera (*Gap*).

Jednotlivá volání funkcí přidávají do této struktury hodnoty. Pokud je potřeba nějaké hodnoty změnit posune se tento index. Pokud je potřeba vložit hodnoty na nějaký určitý index, dojde k posunutí mezery a nové hodnoty se vloží. Všechny tyto operace jsou konstantní, až na operaci posunu mezery, ta má lineární časovou složitost.

Na následujícím příkladu je vidět, jak kompilátor a buffer pracuje.

Ve výpisu 13 je vidět jednoduchá funkce

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(
        text="Count: $count",
        onPress={ count += 1 }
    )
}
...
```

Výpis kódu 13: Příklad před zpracováním *Composable* funkcí kompilátorem

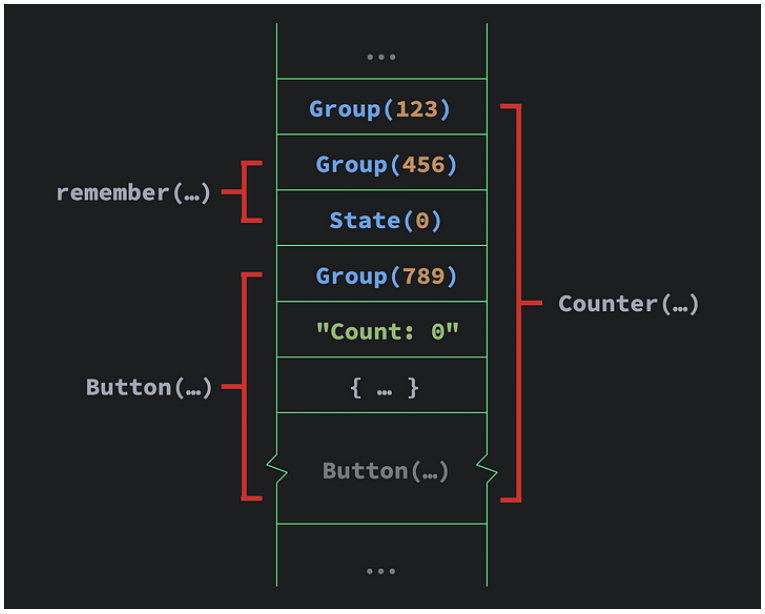
Kompilátor si poté vygeneruje kód 14. Tedy každé *Composable* funkci přidá objekt `Composer`. A pracuje s bufferem následujícím způsobem.

- funkce `start` se zavolá s vygenerovaným unikátním id
- `remember` přidá objekt `group` opět s vygenerovaným id
- Hodnota kterou vrací `mutableStateOf` je uložena jako instance třídy `State`
- `Button` vloží objekt `group`, který je následován každým dalším parametrem.

```
fun Counter(composer: Composer) {
    composer.start(123)
    var count by remember(composer) { mutableStateOf(0) }
    Button(
        composer,
        text="Count: $count",
        onPress={ count += 1 },
    )
    composer.end()
}
...
```

Výpis kódu 14: Příklad po zpracování *Composable* funkcí kompilátorem

Nakonec se zavolá `end`. Stav bufferu po této operaci je vidět na obrázku 3.3.



Obrázek 3.3: Buffer po zpracování ukázkového kódu

Rekompozice, neboli spuštění této funkce znovu, je možné pomocí hodnoty, které volitelně vrací funkce `end`, jak je vidět na ukázce 15.

```
composer.end()?.updateScope { nextComposer ->
    Counter(nextComposer)
}
...
```

Výpis kódu 15: Část spouštějící rekompozici

To říká, jak restartovat tuto funkci, pokud je to třeba. Funkce však vracet nic nemusí, to nastane v případě, že nečte žádnou hodnoty, která se může měnit. V takovém případě není nutné funkci restartovat a o restart, pokud je potřeba, se stará funkce, ze které je tato volána. [24]

3.3.3 Implementace design systému

Protože se v aplikaci používají různé barvy, písma a druhy komponentů, je potřeba je nějak definovat. Při definici pomocí XML se tyto barvy definovaly taktéž v XML. To však už není v *Jetpack Compose* potřeba.

Nejprve se musí specifikovat co téma aplikace bude obsahovat za prvky. V našem případě to budou barvy a písma. Poskytovat do jednotlivých funkcí se budou pomocí `CompositionLocal`.

`CompositionLocal` je nástroj pro implicitní předávání dat skrze kompozici. Je doporučení ho používat právě pro data jako jsou barvy atd. [25]

Nejprve je potřeba definovat třídy, které budou obsahovat všechny dostupné barvy a písma. Nazvěme je `Colors` a `Fonts`. O jejich nastavení se pak starají příslušné funkce, které jsou vidět na výpisech 16 a 17

```
@Composable
fun educhildColors(): Colors {
    return Colors(
        primary = Color(0xFFFFFFFF),
        onPrimary = Color(0xFF161D30),
        secondary = Color(0xFFFE6D72),
        onSecondary = Color(0xFFFFFFFF),
        ...
    )
}
```

Výpis kódu 16: Definování barev

```
@Composable
fun educhildFonts(): Fonts {
    return Fonts(
        headlines = Fonts.Headlines(
            h1 = TextStyle(
                fontSize = 36.sp,
                fontFamily = poppinsFontFamily,
                fontWeight = FontWeight.SemiBold
            ),
            ...
        ),
        ...
    )
}
```

Výpis kódu 17: Definování písma

Nyní je potřeba přidat objekt, který bude tyto hodnoty udržovat. O to se bude starat objekt `EduchildTheme` a jeho definice je vidět na výpisu 18. Jednotlivé hodnoty se berou ze `staticCompositionLocalOf`, která udržuje nastavenou hodnotu napříč celou *composable* hierarchií.

```
private val LocalColors = staticCompositionLocalOf { Colors() }
private val LocalTypography = staticCompositionLocalOf { Fonts() }

object EduchildTheme {

    val typography: Fonts
        @Composable
        @ReadOnlyComposable
        get() = LocalTypography.current

    val colors: Colors
        @Composable
        @ReadOnlyComposable
        get() = LocalColors.current
}
```

Výpis kódu 18: Definování téma aplikace

Hodnoty je však potřeba ještě nastavit na ty správné, protože nyní kompozice ještě neobsahuje správná data. To je třeba udělat v kořenové *Composable* funkci. Díky tomu se tyto hodnoty už budou správně předávat skrze další funkce. Ukázka toho je vidět na výpisu 19, kde je ještě navíc použité *An-*

droid MaterialTheme, které se stará o správné nastavení některých animací, například efekt po kliknutí.

```
CompositionLocalProvider(
        LocalTypography provides educhildFonts(),
        LocalColors provides educhildColors()
) {
    MaterialTheme(content = content)
}
```

Výpis kódu 19: Aplikování téma aplikace

Nyní je tedy v rámci všech funkcí referencovat barvy a písma například pomocí: `EduchildTheme.colors.primary`. Zároveň pokud se nějaká barva změní, dojde ke změně v rámci celé aplikace a nemusí se ručně hledat její výskyty. To je výhodou definování design systému a následného použití tímto způsobem.

Samozřejmě aplikace využívá i stejné UI komponentu napříč celou aplikací. Příkladem mohou být tlačítka. Ty jsou poté definované v modulu *core:design*, stejně jako samotné téma aplikace. Ukázka toho, jak je definované primární tlačítko, je na výpisu kódu 20 a jeho podoba je na obrázku 3.4.

Obrázek 3.4: Primární tlačítko

Na ukázce je vidět, že se používá již dostupné tlačítko, které se pouze styluje. Tedy nastavují se odsazení, barvy atd. Většina takových dostupných komponent dovoluje si nastavit obsah jakýkoliv. Tedy není problém udělat například tlačítko s ikonou uvnitř navíc. V tomto případě však stačí pouze text, který má správné písmo a barvu.

3.3.4 Tvorba UI

Po nastavení téma a přípravě komponent je možné začít implementovat samotné UI. Tomu se budou věnovat následující sekce.

3.3.4.1 Navigace

Protože nová funkce obsahuje několik obrazovek, je potřeba se mezi nimi nějak navigovat. Na to byla použita knihovna *Compose Destinations*, která je postavena nad oficiální navigací pro Compose.

Compose Destinations je KSP knihovna, která zpracovává anotace a generuje kód, který interně používá oficiální *Jetpack Compose Navigation*. Díky

```
@Composable
fun Primary(
    text: String,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    onClick: () -> Unit
) {
    Button(
        onClick = onClick,
        modifier = modifier,
        contentPadding = PaddingValues(vertical = 16.dp),
        enabled = enabled,
        colors = ButtonDefaults.buttonColors(
            containerColor = EduchildTheme.colors.secondary
        )
    ) {
        Text(
            text.toUpperCase(Locale.current),
            style = EduchildTheme.typography.button,
            color = EduchildTheme.colors.onSecondary
        )
    }
}
```

Výpis kódu 20: Definování primárního tlačítka

tomu zjednodušuje komplexní, opakující se kód, který navíc není typově bezpečný. Avšak tím, že využívá stejné API pro navigaci, všechny principy jsou stejné. Mezi výhody tedy patří následující.

- Typově bezpečné argumenty pro navigace
- Jednoduché a typově bezpečné navigování zpět s výsledkem
- Jednoduché vytváření deeplinků

[26]

3.3.4.2 Komunikace mezi UI a ViewModelem

Každý ViewModel, který přísluší obrazovce poskytuje jednu třídu, která reprezentuje stav a vystavuje jej pomocí `StateFlow`. Tento stav je většinou reprezentovaný pomocí `sealed interface` a jeho konkrétní implementace určují jaký stav může nastat. Například stav, který vystavuje obrazovka se seznamem vytvořených kvízů, nabízí následující stavy.

- `Loading`, úvodní stav, reprezentující načítání kvízů
- `Empty` stav, kdy žádný kvíz ještě není vytvořen
- `Loaded` stav, kdy se načtou vytvořené kvízy a seznam není prázdný

Díky tomu muže UI reagovat na tyto stavy a zobrazovat příslušné komponenty. Jak bylo řečeno v kapitole 3.3, je *Compose* deklarativní a nemusí se starat o předchozí stavy. Stačí tedy kontrolovat, jaký je stav a vykreslit to co pro něj přísluší.

`StateFlow` slouží jako držitel stavu, který emituje aktuální a další každou změnu tohoto stavu do všech kolektorů. Hodnotu lze samozřejmě kdykoliv číst, či měnit (to však pokud se jedná o instanci `MutableStateFlow`). [27]

`Sealed interface` představují omezené hierarchie tříd, které poskytují větší kontrolu nad dědičností. Všechny podtřídy jsou známé již při kompilaci. Žádné další podtřídy se nesmí objevit mimo modul nebo aktuální balíček v němž je rozhraní definováno. Díky tomu je možné kontrolovat jaký je stav a vždy si být jistý, jaké možnosti, respektive podtřídy, mohou nastat. [28]

Jednotlivé obrazovky vždy mají vlastní *Composable* funkci, ve které se předají její argumenty, samozřejmě pokud jsou nějaké potřeba, ViewModel a třída starající se o navigaci. Tato funkce má anotaci `Destination`, to zaručuje, že se pro ní vygeneruje destinace a poté lze na tuto obrazovku navigovat. Uvnitř této funkce se začne poslouchat stav z ViewModelu. S tímto stavem se poté zavolá *Composable* funkce, která už uvnitř není závislá na ViewModelu a zobrazuje uživatelské rozhraní na základě tohoto stavu. Díky tomu první z těchto funkcí řeší rekompozici, tak jak bylo popsáno v sekci 3.3.2, ta druhá, která už jen přebírá stav, je poté ve většině případů bezstavová a nemusí to řešit. To vede ke zlepšení výkonu a lepší oddělení logiky v UI, protože druhá funkce nemá ani referenci na ViewModel, ale vystavuje pouze funkce, které podporuje. Ukázka tohoto rozdělení je vidět na výpisu 21.

V ukázce 21 je navíc ještě vidět, jak UI nemá povědomí o tom, jak reagovat na různé akce. Pouze vystavuje rozhraní, které ho zpracovává pomocí lambda funkcí. Stavová *Composable* až řeší tuto logiku a díky tomu je vše na jednom místě. To má výhodu pro budoucí potencionální změny, v tomto případě například změna akce po kliknutí na tlačítko přidat, může vést na jinou obrazovku, popřípadě zobrazit nějakou hlášku a UI se nebude muset vůbec měnit.

V některých případech je však nutné komunikovat z ViewModelu směrem k UI. Příkladem může být například potřeba zobrazit hlášku, když selže nějaká operace. Na to ale může posloužit samotný stav, který ViewModel poskytuje.

Díky tomu na to může UI správně zareagovat. Avšak po reakci na událost, je potřeba tento stav opět resetovat. To lze ale už udělat způsobem, který byl popsán v předchozích odstavcích, a to pomocí lambda funkcí a následném zavoláním funkce ve ViewModelu. Tento způsob přináší výhodu toho, že je zaručeno, aby událost byla zpracována správně a atomicky. Pokud by uživatel

```
@Composable
@RootNavGraph(start = true)
@Destination
fun QuizListScreen(
    navigator: DestinationsNavigator,
    viewModel: QuizListViewModel = koinViewModel(),
    ...
) {
    val state by viewModel.screenStateStream.collectAsStateWithLifecycle()
    QuizListScreen(
        state = state,
        onAddClick = {
            navigator.navigate(CreateQuizScreenDestination())
        },
        onQuizClick = {
            navigator.navigate(QuizDetailScreenDestination(it.id))
        },
        ...
    )
    ...
}
```

Výpis kódu 21: Rozdělení funkcí pro jednotlivé obrazovky

například změnil orientaci obrazovky, kdy se musí překreslit celé UI, a došlo by k tomu během ukazování nějaké hlášky na obrazovce. Ta by však byla zobrazena pouze na krátkou dobu a uživatel si jí nemusel všimnout. Díky tomuto resetování stavu po konci zpracování událost, se ale hláška po změně orientace zobrazí znovu a uživatel je informovaný o tom, co se v aplikaci děje. [29]

3.3.4.3 Obrazovka pro tvorbu otázky

Tato sekce bude popisovat, jak je vytvořena obrazovka pro tvorbu otázky, protože ta je ze všech vzniklých obrazovek nejsložitější.

Důvodem této složitosti je skutečnost, že musí reagovat na různé typy otázek.

Protože je obsah dynamický a u typů otázek jako je výběr z více možností a spojování dvojic může být více řádek, jsou jednotlivé komponenty vkládány do kontejneru `LazyColumn`.

To je scrolovací kontejner, který vykresluje pouze potomky, které jsou aktuálně vidět na obrazovce. Díky tomu se šetří prostředky zařízení. [30]

První část obrazovky obsahuje komponenty, které jsou společné pro všechny typy otázek.

1. Menu pro výběr typu otázky
2. Zadání otázky
3. Vysvětlení odpovědi

Tyto komponenty jsou zobrazeny vždy a zbytek se ukazuje na základě toho, jaký je aktuálně zvolený typ otázky. Díky možnostem *Compose* a jeho deklarativnímu způsobu je jednoduché tohoto docílit a ukázka, jak je to v aplikaci řešené je vidět ve výpisu 22. Opět je zde využita vlastnost, že nezáleží na předchozím stavu, ale pouze na stavu aktuálním.

```
private fun LazyListScope.itemQuestion(
    question: QuizQuestion,
    ...
) {
    when (question) {
        is QuizQuestion.Open -> itemOpenQuestion(
            question = question,
            ...
        )

        is QuizQuestion.Select -> itemSelectQuestion(
            question = question,
            ...
        )

        is QuizQuestion.Pairs -> itemPairQuestion(
            question = question,
            ...
        )
    }
}
```

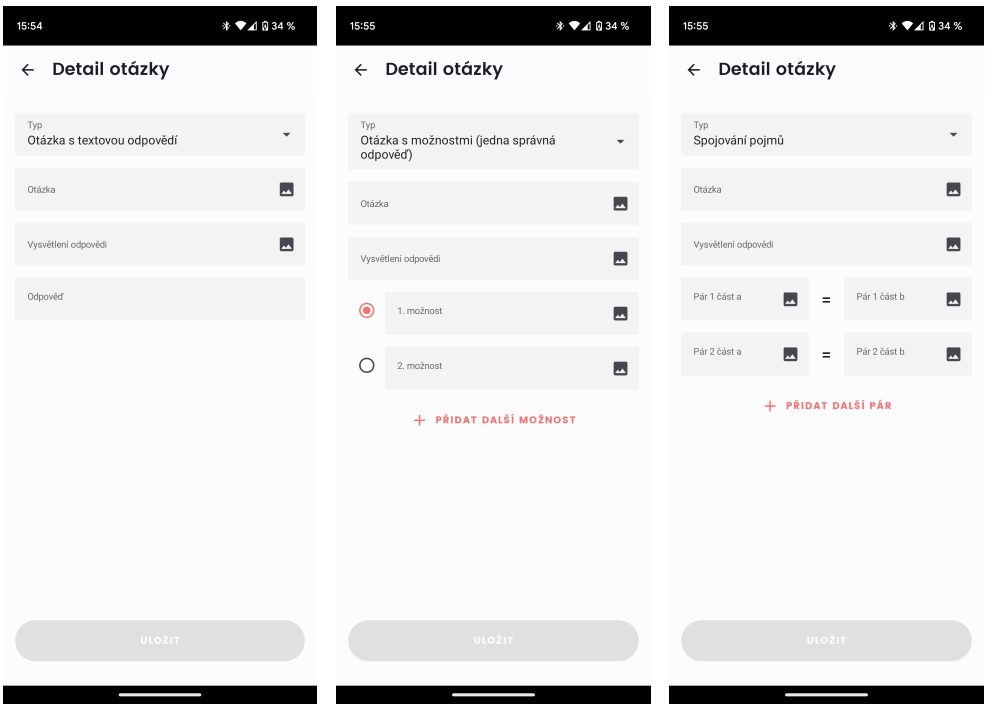
Výpis kódu 22: Zobrazení obsahu na základě typu otázky

V případě otevřené otázky se zobrazí pouze jedno vstupní pole pro zadání této odpovědi. Samozřejmostí je výběr, zda se bude jednat o textovou, či číselnou odpověď a podle toho se nastaví, jaké hodnoty půjde do tohoto pole zadat.

Typ pro výběr z více možností poté zobrazuje řádky pro jednotlivé možnosti. Každý řádek má možnost zvolit, zda je správně a textové pole pro znění této možnosti. Je zde samozřejmě možnost přidat další řádek, popřípadě ho odstranit.

Stejně jako předchozí typ funguje i typ spojování dvojic. Ten ale zobrazuje dvě textové pole pro zadání obou částí dvojice.

To jak obrazovky vypadají je znázorněno na obrázcích 3.5.



Obrázek 3.5: Obrazovky pro tvorbu otázky

U jednotlivých položek otázky, lze přidávat obrázky. Android na to nabízí nástroj zvaný `Photo picker`. Ta zobrazuje obrazovku, se všemi obrázky v zařízení a uživatel si může vybrat jakou chce. Do aplikace se poté vrátí objekt, který obsahuje cestu k tomuto souboru a je zajištěno, že k němu bude mít aplikace přístup i bez žádných dalších oprávnění. [31]

V hlavní funkci pro obrazovku (ta která má referenci na ViewModel) poté stačí zavolat funkci, která otevírá obrazovku pro vybrání obrázku. Následně přijde výsledek a ten se přiřadí k otázce. Proto, aby se obrázek přiřadil na správné místo, slouží ve stavu atribut `imageDestination`. Ten se před otevřením výběru obrázků nastaví na správnou hodnotu. Tyto destinace jsou definované opět pomocí `sealed interface` a mohou být následující.

- `Question` určující zadání otázky
- `Explanation` patřící k vysvětlení odpovědi
- `Answer`, který obsahuje navíc index o kterou odpověď se jedná
- `PairItemA` taktéž obsahuje index o který pár se jedná
- `PairItemB` opět spolu s indexem

Díky tomu je poté ViewModel schopný upravit otázku tak, aby obsahovala přidaný obrázek.

3.4 Publikování

V této sekci bude popsáno, jak probíhá samotné automatizované publikování na Google Play.

Nejprve se bude věnovat procesu podepisování nahrávaného souboru a v druhé části poté samotné nahrávání pomocí CI.

3.4.1 Podepisování

Android vyžaduje, aby všechny instalační soubory `.apk` byly podepsány certifikátem ještě před tím, než jsou nainstalované do zařízení.

Samotné Google Play spravuje a uchovává klíč k podpisu aplikace a následně s ním podepisuje instalační soubory pro distribuci. Pro samotné nahrání na Google Play se však používají soubory `.aab`. Díky nim je možné odložit sestavení a podepisování souborů `.apk` až na Google Play obchod. To přináší několik výhod.

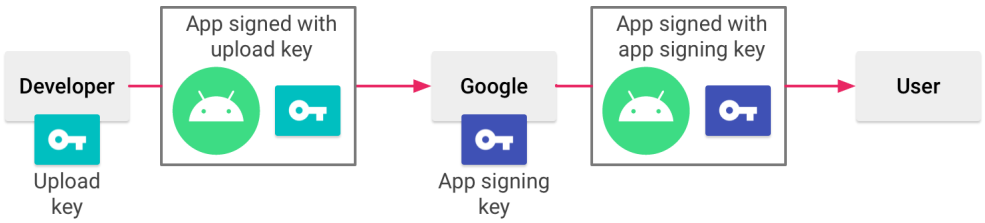
- `.aab` soubor podporuje pokročilé způsoby distribuce v Google Play. Díky tomu je výsledný instalační soubor mnohem menší a používá pouze části, které jsou pro dané zařízení potřeba. Například na základě rozlišení obrazovky zařízení.
- Zvyšuje bezpečnost samotného podpisového klíče a díky tomu umožňuje použít samostatný klíč pro nahrání do služby
- Umožňuje změnit podpisový klíč v případě jeho kompromitace

Celý proces podepisování tedy vyžaduje dva různé klíče.

1. *Upload key*
2. *Signing key*

První je spravován uživateli, kteří aplikaci nahrávají, a slouží při nahrávání aplikace k ověření identity aplikace. Následně služba Google Play podepíše aplikaci klíčem číslo dva. Tento klíč si spravuje sama. Následně je aplikace připravena k nainstalování u koncových uživatelů. [32]

Samotný proces podepisování je znázorněn na obrázku 3.6.



Obrázek 3.6: Proces podepisování aplikace [33]

3.4.2 Nahrávání na Google Play

Po podpisu aplikace je možné aplikaci nahrát na Google Play. O to se stará *Gradle* plugin.

Plugin se jmenuje *Gradle Play Publisher* a slouží k automatizovanému nahrání souborů na Google Play. V konfiguraci pluginu je možné nastavit všechny možnosti, které jsou u nahrání možné. Příkladem může být to, zda se aplikace má nahrát pouze do interního testování, do uzavřeného, otevřeného nebo rovnou do produkce. Samozřejmostí je možnost toto měnit až po nahrání. Aby toto bylo možné, je potřeba vygenerovat klíč, který slouží k autorizaci na Google Play. Postup pro vygenerování je však popsán v dokumantaci pluginu. [34]

Po přidání pluginu je poté možné zavolat samotné publikování na CI. Protože projekt již má Gitlab CI nastavené, stačilo pouze přidat jeden úkon, který se bude o publikování starat. Část kódu přidaná do souboru `.gitlab-ci.yml` je na ukázce 23. Stará se o to, že každý commit, který obsahuje tag, se automaticky nahraje na Google Play. Součástí toho je také přidání souboru s klíčem pro nahrání, který se nachází pouze v Gitlabu.

```
deploy:googlePlayDeploy:
  <<: *deploy
  only:
    - /^release\/[0-9]+[.][0-9]+[.][0-9]+$/
  dependencies:
    - build:release
  script:
    - echo $PUBLISHER_KEY > publisher-key.json
    - ./gradlew publishProdApiReleaseBundle
    - rm -f publisher-key.json
```

Výpis kódu 23: Rozšíření Gitlab CI o publikování na Google Play

3.5 Shrnutí

Výstup této kapitoly je hotová implementace, která dodržuje návrh z předchozí kapitoly. Samozřejmostí je také napojení na backend, díky čemu jsou data synchronizovaná a dostupná ve všech zařízeních, kde se uživatel přihlásí. Věnovala se zejména vyřešení modularizace pomocí *Gradle* a následně implementaci UI za pomoci *Jetpack Compose*. A na závěr shrnula, jak funguje publikování aplikace. Celá implementace je poté dostupná na `https://gitlab.fit.cvut.cz/simapet4/educhild-android`.

Aplikace byla následně vydaná na Google Play a to do interního testování. Pro každého uživatele, který si bude chtít stáhnout aplikaci, je potřeba přidat jeho email a zaslat mu odkaz s pozvánkou k testování. Takových uživatelů může být pozváno 100. Pro přidání aplikace do dalších fází testování, je potřeba doplnit dokument o zpracování osobních údajů, doplnit popisy k aplikaci a vyplnit potřebné formuláře o obsahu.

Kapitola 4

Testování

Aby bylo možné zjistit, zda implementace z předchozí kapitoly funguje správně, je potřeba aplikovat testy.

Právě tomuto se bude zabývat tato kapitola. Ta nejprve popíše druhy testů, následně popíše jejich aplikaci a jako výstup bude pokrytí ḱódu testy.

4.1 Unit testování

Pro ověření funkčnosti implementace byly napsány Unit testy.

Unit tetování je typ testování softwaru, kde jsou testovány jednotlivé jednotky či komponenty. Účelem toho je uvěření, že každá tato jednotka funguje správně a tak, jak je očekáváno. Unit testy izolují sekce kódu a ověřují je. Tato sekce může být funkce, objekt nebo celý modul. [35]

Pro realizaci těchto testů bylo použito několik technologií, které budou v následujících sekcích popsány.

4.1.1 JUnit

Pro testování Android aplikací jsou jedním z typů, které lze použít, takzvané lokální testy. Ty běží přímo na pracovní stanici, nikoliv na Android zařízení. K jejich spouštění se používá JVM. Díky tomu jsou testy rychlejší. Tyto lokální testy tedy mohou provádět Unit testování. Pro vytvoření těchto lokálních testů se standardně používá *JUnit* framework. [36]

4.1.2 Test double

Při testování jednotlivých částí systému je důležitá jejich izolace, zejména pak při Unit testování. Avšak jednotlivé komponenty mají většinou závislost na další části systému. Proto je běžná praxe pro ně vytvořit *test double*. To jsou objekty, které vypadají a chovají se stejně jako komponenty v aplikaci, ale

jsou vytvořeny, aby při testování poskytovaly určité chování či data. Díky tomu jsou testy jednodušší a rychlejší. Patří mezi ně několik typů.

- `Fake` má fungující implementaci třídy, ale je implementována tak, aby byla vhodná pro testování. Příkladem může být in–memory databáze.
- `Mock` chová se tak, jak je naprogramovaný a má očekávání ohledně interakcí, tedy může mít jiné chování než skutečná implementace. Jsou většinou vytvářeny pomocí mockovacích frameworků.
- `Stub` je podobný jako `Mock`, avšak neočekává žádné interakce. Je opět většinou vytvářen pomocí mockovacích frameworků.

[37]

V projektu je již použitý mockovací framewok pro mockování `MockK`, proto bude pro implementaci testů využit také.

4.1.3 Robolectric

Robolectric je framework, který přináší rychle a spolehlivě Unit testy na Android. Testy poté běží uvnitř JVM on pracovní stanici. Hlavní výhodou je, že tyto testy dovolují nakonfigurovat Android prostředí. U samostatných JUnit testů totiž není možné používat komponenty z Android frameworku. To však s použitím *Robolectric* možné je. [38]

4.1.4 Přístup do databáze

Jak již bylo zmíněno aplikace si udržuje lokální databázi a přistupuje k ní. Aby mohlo být toto chování korektně otestováno, byl použit právě framework *Robolectric*, který byl zmíněn v předchozí sekci. Díky němu je možné si vytvořit totožnou databázi, která se používá v aplikaci, avšak při testování je pouze jako in–memory.

Pro testování databáze byla vytvořena abstraktní třída `DatabaseTest`, která je vidět na ukázce kódu 24.

Tato třída se stará nejen o nastavení *Robolectic*, pomocí anotací, ale také o vytvoření databáze pomocí JUnit pravidla `DatabaseTestRule`. Implementace tohoto pravidla je na ukázce 25.

JUnit pravidlo implementováno pomocí třídy `TextWatcher` zajišťuje, že budou zavolány metody jako je `starting` respektive `finished` před respektive po vykonání samotného testu. [39]

A díky tomu je možné vytvořit databázi před tím, než se vykoná samotný test a poté ji uzavřít. Pro vytvoření databáze je však potřeba `Context` z Android frameworku, ten je ale dostupný díky *Robolectic*.

Pomocí těchto testů je tedy možné ověřit nejen to, že se volají správné funkce a dochází ke korektnímu mapování, ale i jednotlivé dotazy na samotnou databázi.

```
@RunWith(RobolectricTestRunner::class)
@Config(manifest = Config.NONE)
abstract class DatabaseTest {

    @get:Rule
    val databaseTestRule = DatabaseTestRule()

    val database
        get() = databaseTestRule.educhildDatabase
}
```

Výpis kódu 24: Ukázka třídy DatabaseTest pro testování lokální databáze

```
class DatabaseTestRule : TestWatcher() {

    lateinit var educhildDatabase: EduChildDatabase

    override fun starting(description: Description) {
        educhildDatabase = Room.inMemoryDatabaseBuilder(
            ApplicationProvider.getApplicationContext(),
            EduChildDatabase::class.java
        ).build()
    }

    override fun finished(description: Description) {
        educhildDatabase.close()
    }
}
```

Výpis kódu 25: Ukázka pravidla pro vytvoření databáze

4.1.5 ViewModel

Protože se ve ViewModely spouští jednotlivé korutiny a po zavolání jednotlivých funkcí dochází k asynchronnímu vykonání a následnému nastavení stavu do streamu, jak bylo popsáno dříve v této práci. Proto je potřeba správně nastavit, aby se vše spouštělo postupně a bylo tak vhodné pro testy.

Na to slouží další pravidlo `MainDispatcherRule`, které nastaví popsané chování pomocí třídy `UnconfinedTestDispatcher`, která je na to přímo určená. Implementace tohoto pravidla je na ukázce 26.

4.1.6 Moduly pro testování

Protože se některá logika sdílí napříč více moduly, bylo vhodné vytvořit nový modul *core:testing*, který obsahuje vše, co je potřeba pro testování napříč celou

```
class MainDispatcherRule(
    val testDispatcher: TestDispatcher = UnconfinedTestDispatcher(),
) : TestWatcher() {
    override fun starting(description: Description) {
        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description) {
        Dispatchers.resetMain()
    }
}
```

Výpis kódu 26: Ukázka pravidla pro nastavení spouštění korutin pro testování

aplikací. Příkladem mohou být pravidla pro testování databáze či ViewModelů.

Druhým modulem, který během testování vznikl, je modul *data:quiz:test*, který obsahuje pomocné funkce pro vytvoření modelů potřebných pro testování. Jako například funkce `createQuiz(...)` vytvářející objekt kvízu s testovacími daty. Vytvoření tohoto modulu bylo potřeba, protože tyto funkce jsou nutné v testech v modulech *feature:create-quiz* a *data:quiz:impl*.

Tyto moduly jsou do ostatních přidány však pouze při spouštění testů a samotné sestavování aplikace to nijak neomezuje.

4.2 Akceptační testování

Akceptační testování je typ testování prováděný koncovým uživatelem s účelem ověření funkčnosti systému. [40]

Při akceptačním testování byly zjištěny tři problémy, které budou uvedeny v následujících odstavcích a jsou seřazeny podle jejich závažnosti.

Prvním problémem je přidávání obrázků k jednotlivým otázkám. Implementace podporuje pouze vybírání obrázků z galerie mobilního zařízení. Uživatel ale nemá možnost si vyfotit novou fotku. Pokud tedy fotku nemá, musí otevřít externí aplikaci pro pořizování fotek, fotku pořídit, vrátit se zpět a fotku přidat z galerie.

Dalším problémem je chybějící proklik do úpravy kvízu z obrazovky detailu, kde se kvízy přiřazují dítěti. Uživatel tedy nemá možnost v této fázi jak kvíz jednoduše upravit, musí se vrátit zpět a otevřít část aplikace pro tvoření kvízů.

Třetím problémem je nemožnost zrušit publikaci kvízu. Pokud je kvíz jednou publikován, je potřeba pro zrušení této akce kvíz smazat a následně obnovit.

Po tomto testování byly opraveny první dva nalezené problémy. U prvního problému, kdy nebyla možnost pořizovat nové obrázky přímo z aplikace, byl přidán dialog, který uživateli nabízí možnost vyfocení nebo vybrání z galerie. Uživatel už tedy nemusí otevírat externí aplikaci. Druhý problém byl vyřešen tak, že pro vytvořené kvízy přihlášeným uživatelem se zobrazí tlačítko pro upravení kvízů, které vede na příslušnou obrazovku.

4.3 Shrnutí

Tato kapitola se věnovala ověření správnosti implementace, která vznikla.

Nejprve popsala typ použitých testů, následně technologie a nakonec jejich použití pro co nejlepší testování.

Výsledkem je tedy otestovaná nově vzniklá implementace. Celkem bylo napsáno 148 testů na třídy typu *Repository*, *DataSource* a *ViewModel*. Tyto třídy byly testovány z důvodu, že je zde veškerá logika pro novou funkci. Pokrytí testy, které bylo vygenerováno nástrojem, který nabízí vývojové prostředí *Android Studio*, lze vidět v tabulce 4.1.

Následně bylo provedeno akceptační testování, po kterém byly opraveny dva ze tří nalezených problémů.

ViewModel	
Třída	**Pokrytí**
QuizListViewModel	100 %
QuizDetailViewModel	100 %
CreateQuizViewModel	100 %
CreateQuestionViewModel	85 %

Repository	
Třída	**Pokrytí**
AuthorRepositoryImpl	100 %
QuizCategoryRepositoryImpl	100 %
QuizImageRepositoryImpl	100 %
LanguageRepositoryImpl	100 %
QuestionRepositoryImpl	100 %
QuizRepositoryImpl	100 %

DataSource	
Třída	**Pokrytí**
RetrofitQuizImageDataSource	75 %
FileImageDataSource	100 %
RetrofitQuizCategoryDataSource	100 %
RetrofitLanguageDataSource	100 %
RetrofitQuizQuestionDataSource	100 %
RoomQuizQuestionDataSource	100 %
RetrofitQuizDataSource	100 %
RoomQuizDataSource	100 %

UseCase	
Třída	**Pokrytí**
CreateOrUpdateQuizQuestionUseCaseImpl	100 %
DownloadQuizQuestionsUseCaseImpl	100 %
GetQuizzesWithQuestionsUseCaseImpl	100 %
SyncCreatedQuizzesUseCaseImpl	100 %

Tabulka 4.1: Pokrytí jednotlivých tříd testy

Závěr

Celý projekt se sice skládá z Android aplikace, webové aplikace a backendu, ale tato práce rozšiřuje pouze aplikaci pro operační systém Android.

Cílem práce bylo zjistit, zda v aplikaci chybí nějaká zásadní funkce. Tu následně doplnit a zajistit automatizované vydávání na Google Play.

V první kapitole se tedy tato práce věnovala právě analýze funkcí, které už mobilní aplikace uměla. Následně se porovnaly s funkcemi, co nabízí webová aplikace. Tyto funkce byly popsány pomocí případů užití. Na základě toho bylo zjištěno, že v mobilní aplikaci chybí možnost kvízy vytvářet. Proto byla ještě provedena analýza API, které vystavuje backend.

Druhá kapitola se následně věnovala návrhu. V úvodu kapitoly je popsaná problematika modularizace a její stávající použití v aplikaci. Na základě toho bylo zjištěno, že je potřeba tento přístup k modularizaci změnit. Proto byly prozkoumány již existující aplikace, které modularizaci využívají. Ty sloužily jako podklad pro nový návrh rozložení do modulů spolu s návrhem, jak ji správně aplikovat s novou implementovanou funkcí. V druhé polovině této kapitoly byl popsán doménový model, který popisuje kvízy a jednotlivé kvízové otázky.

Ve třetí kapitole byla popsána implementace. Nejprve popsala použité technologie a následně se věnovala změnám ve skriptech, které slouží k sestavení celé aplikace. Tyto změny bylo potřeba udělat pro zpřehlednění celé logiky a jednodušší aplikaci nového návrhu modularizace z předchozí kapitoly. Poté se věnovala frameworku pro definování UI. Ten byl použit jiný než ve stávající aplikaci je. Jak však práce zmiňuje, použití tohoto frameworku přináší mnoho výhod a díky jeho interoperabilitě nebylo jeho přidání do projektu žádný problém. Kapitola se tedy věnuje tomu, jak framework funguje a následně jak je pomocí něj uživatelské rozhraní implementováno. Na závěr této kapitoly je popsán samotný proces potřebný pro nahrání na Google Play společně s postupem, jak funguje automatizované nahrávání pomocí *Gitlab CI*.

Čtvrtá a poslední kapitole se věnuje testování hotové implementace. Po-

pisuje druhy testů, které byly použity. Taktéž zmiňuje, jak jsou testovány Android komponenty pomocí lokálních Unit testů.

Výsledkem je tedy aplikace, která obsahuje chybějící funkci tvorby kvízů. Vytvořené kvízy jsou spojené se svým autorem, tedy přihlášeným uživatelem a jsou nahrávané na server, aby byly dostupné na všech zařízeních uživatele. Vytvořené kvízy jsou taktéž ukládány do lokální databáze, aby je bylo možné prohlížet, popřípadě následně i vyplňovat, v režimu bez připojení k internetu. Tato aplikace je momentálně nahraná na Google Play a poskytnutá k internímu testování.

Zadání práce bylo splněno, ale i tak se nabízí několik možností pro zlepšení. Jedním z nich je aplikovat návrh modularizace pro celou aplikaci, tak jak bylo popsáno v kapitole návrhu. A samozřejmě také vydat aplikaci do produkce, aby byla dostupná pro všechny uživatele.

Literatura

1. VISUAL PARADIGM. *What is Use Case Diagram?* [online]. 2023. [cit. 2023-04-07]. Dostupné z: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/.
2. ALTEXSOFT. *Functional and Non-functional Requirements: Specification and Types* [online]. 2023. [cit. 2023-04-07]. Dostupné z: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/.
3. TECHTARGET. *MoSCoW method* [online]. 2023. [cit. 2023-04-30]. Dostupné z: https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method.
4. GOOGLE. *Guide to android app modularization* [online]. 2023. [cit. 2023-04-09]. Dostupné z: https://developer.android.com/topic/modularization.
5. GOOGLE. *Dependency graph of a sample multi-module codebase* [online]. 2023. [cit. 2023-04-09]. Dostupné z: https://developer.android.com/static/topic/modularization/images/1_sample_dep_graph.png.
6. GOOGLE. *Now in Android* [online]. 2023. [cit. 2023-04-09]. Dostupné z: https://github.com/android/nowinandroid.
7. GOOGLE. *Types of modules in Now in Android* [online]. 2023. [cit. 2023-04-09]. Dostupné z: https://raw.githubusercontent.com/android/nowinandroid/main/docs/images/modularization-graph.drawio.png.
8. GOOGLE. *Now in Android, Modularization learning journey* [online]. 2023. [cit. 2023-04-09]. Dostupné z: https://github.com/android/nowinandroid/blob/main/docs/ModularizationLearningJourney.md.
9. SCALED AGILE. *Domain Modeling* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://scaledagileframework.com/domain-modeling/.

10. JETBRAINS. *Kotlin for Android* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://kotlinlang.org/docs/android-overview.html.

11. GOOGLE. *Kotlin coroutines on Android* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://developer.android.com/kotlin/coroutines.

12. SQUARE. *Retrofit* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://square.github.io/retrofit/.

13. GOOGLE. *Save data in a local database using Room* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://developer.android.com/training/data-storage/room.

14. GOOGLE. *Diagram of Room library architecture* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://developer.android.com/static/images/training/data-storage/room_architecture.png.

15. KOIN & KOTZILLA. *What is Koin?* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://insert-koin.io/docs/reference/introduction.

16. KOIN & KOTZILLA. *Definitions* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://insert-koin.io/docs/reference/koin-core/definitions.

17. GRADLE INC. *What is Gradle?* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.gradle.org/current/userguide/what_is_gradle.html.

18. GRADLE INC. *Sharing build logic between subprojects Sample* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.gradle.org/current/samples/sample_convention_plugins.html.

19. GRADLE INC. *Developing Custom Gradle Plugins* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.gradle.org/current/userguide/custom_plugins.html.

20. GRADLE INC. *Composing builds* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.gradle.org/current/userguide/composite_builds.html.

21. GRADLE INC. *Sharing dependency versions between projects* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.gradle.org/current/userguide/platforms.html.

22. GOOGLE. *Why adopt Compose* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://developer.android.com/jetpack/compose/why-adopt.

23. RICHARDSON, Leland. *Understanding Jetpack Compose — part 1 of 2* [online]. 2020. [cit. 2023-04-19]. Dostupné z: https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050.

24. RICHARDSON, Leland. *Understanding Jetpack Compose — part 2 of 2* [online]. 2020. [cit. 2023-04-19]. Dostupné z: https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd.

25. GOOGLE. *Locally scoped data with CompositionLocal* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://developer.android.com/jetpack/compose/compositionlocal.

26. COSTA, Rafael. *Compose Destinations* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://composedestinations.rafaelcosta.xyz/.

27. GOOGLE. *StateFlow and SharedFlow* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://developer.android.com/kotlin/flow/stateflow-and-sharedflow.

28. JETBRAINS. *Sealed classes and interfaces* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://kotlinlang.org/docs/sealed-classes.html.

29. GOOGLE. *UI events* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://developer.android.com/topic/architecture/ui-layer/events.

30. GOOGLE. *Lists and grids* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://developer.android.com/jetpack/compose/lists.

31. GOOGLE. *Photo picker* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://developer.android.com/training/data-storage/shared/photopicker.

32. GOOGLE. *Sign your app* [online]. 2023. [cit. 2023-04-21]. Dostupné z: https://developer.android.com/studio/publish/app-signing.

33. GOOGLE. *Signing an app with Play App Signing* [online]. 2023. [cit. 2023-04-21]. Dostupné z: https://developer.android.com/static/studio/images/publish/appsigning_googleplayappsigningdiagram_2x.png.

34. TRIPLE-T. *Gradle Play Publisher* [online]. 2023. [cit. 2023-04-21]. Dostupné z: https://github.com/Triple-T/gradle-play-publisher.

35. HAMILTON, Thomas. *Unit Testing Tutorial – What is, Types & Test Example* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://www.guru99.com/unit-testing-guide.html.

36. GOOGLE. *Build local unit tests* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://developer.android.com/training/testing/local-tests.

37. GOOGLE. *Use test doubles in Android* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://developer.android.com/training/testing/fundamentals/test-doubles.

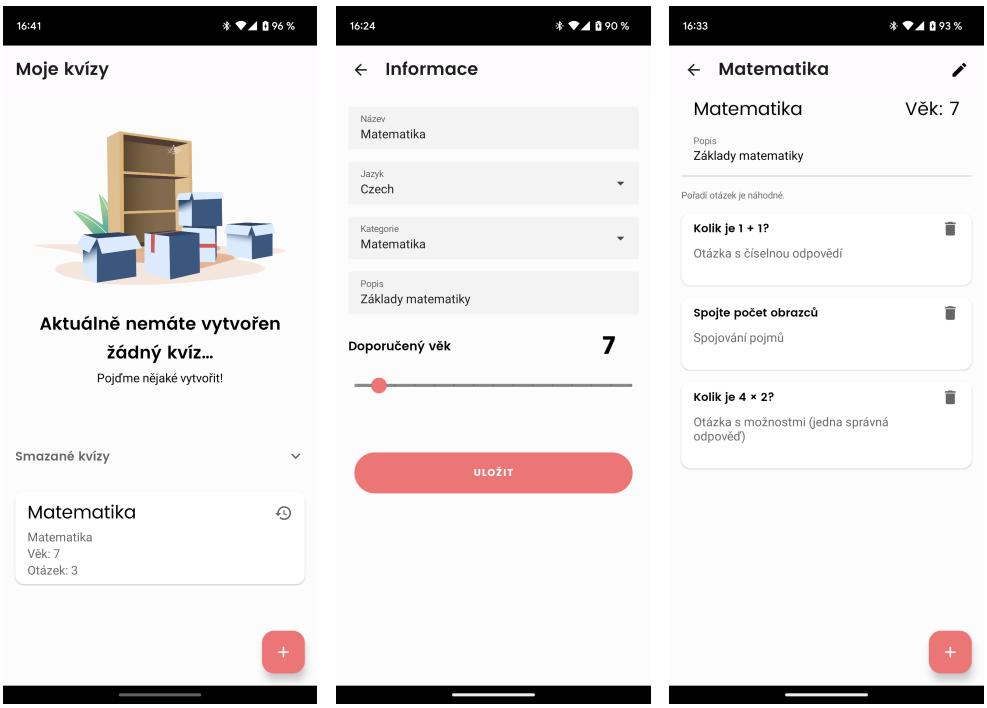
38. ROBOLECTRIC. *Robolectric* [online]. 2023. [cit. 2023-04-29]. Dostupné z: `https://robolectric.org/`.

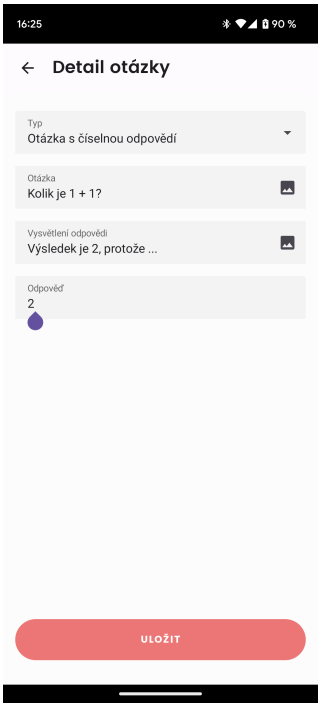
39. JUNIT. *TestWatcher* [online]. 2023. [cit. 2023-04-29]. Dostupné z: `https://junit.org/junit4/javadoc/4.12/org/junit/rules/TestWatcher.html`.

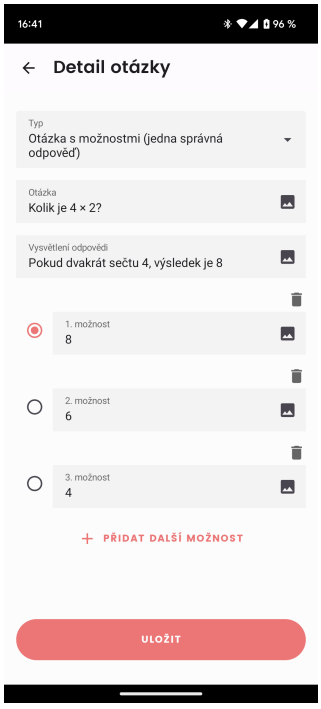
40. GURU99. *What is User Acceptance Testing (UAT)?* [online]. 2023. [cit. 2023-04-30]. Dostupné z: `https://www.guru99.com/user-acceptance-testing.html`.

Příloha A

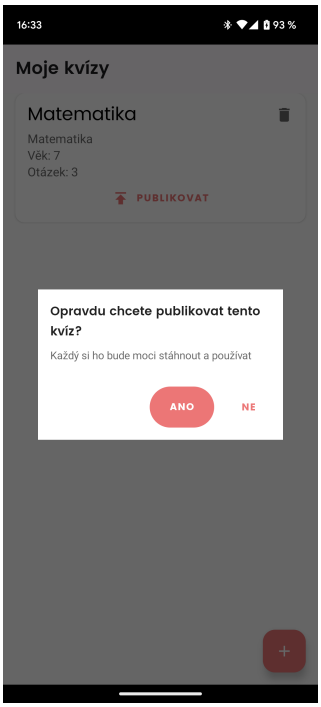
Snímky obrazovky z aplikace




16:25
90 %
Detail otázky
Typ
Otázka s číselnou odpovědí
Otázka
Kolik je 1 + 1?
Vysvětlení odpovědi
Výsledek je 2, protože ...
Odpověď
2
ULOŽIT


16:41
96 %
Detail otázky
Typ
Otázka s možnostmi (jedna správná odpověď)
Otázka
Kolik je 4 × 2?
Vysvětlení odpovědi
Pokud dvakrát sečtu 4, výsledek je 8
1. možnost
8
2. možnost
6
3. možnost
4
PŘIDAT DALŠÍ MOŽNOST
ULOŽIT


16:32
93 %
Detail otázky
Otázka
Spojte počet obrazců
Vysvětlení odpovědi
Počet udává kolik je kruhu na obrázku
Pár 1 část a
=
Pár 1 část b
3
Pár 2 část a
=
Pár 2 část b
4
ULOŽIT


16:33
93 %
Moje kvízy
Matematika
Matematika
Věk: 7
Otázek: 3
PUBLIKOVAT
Opravdu chcete publikovat tento kvíz?
Každý si ho bude moci stáhnout a používat
ANO
NE

Příloha B

Seznam použitých zkratek

API Application Programming Interface

CI Continuous integration

DAO Data access object

FIT Fakulta informačních technologií

FR Functional requirement

IDE Integrated development environment

JSON JavaScript Object Notation

JVM Java virtual machine

NFR Non-functional requirement

NI–NUR Návrh uživatelského rozhraní, předmět vyučovaný na ČVUT FIT

SQL Structured query language

Příloha C

Obsah přiloženého CD

- `README.md` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Petr_Sima_2023.pdf` ..... text práce ve formátu PDF
- `assets`
  - `docs` ..... dokumentace
  - `tests` ..... výstup testů