Zadání diplomové práce

Název:	Integrace a nasazení mobilní aplikace pro studenty znakového jazyka v organizaci Tichý svět
Student:	Bc. Jindřich Žák
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Seznamte se s bakalářskou prací Alžběty Gogolákové, která navrhla a implementovala prototyp mobilní aplikace pro výuku znakového jazyka pro účastníky kurzů organizace Tichý svět.
Zároveň se seznamte s existujícím prostředím pro podporu výuky Tichého světa a jejich požadavky na autentizaci a autorizaci uživatelů mobilní aplikace
Navrhněte a implementujte vhodnou webovou službu poskytující data pro mobilní aplikaci, která autentizaci a autorizaci provede.
Upravte mobilní aplikaci tak, aby konzumovala data poskytovaná výše zmíněnou webovou službou.
Zdrojové kódy aplikace nasaďte na fakultní GitLab a implementujte pipelines pro build vývojové, testovací a produkční verze pro varianty Android a iOS.
S využitím školních účtů nebo účtů Tichého světa aplikaci nasaďte na Google Play a AppStore.
Celý postup zdokumentujte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 18. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Integrace a nasazení mobilní aplikace pro studenty znakového jazyka v organizaci Tichý svět

Bc. Jindřich Žák

Katedra softwarového inženýrství
Vedoucí práce: Ing. Michal Valenta, Ph.D.

6. května 2021

Poděkování

Chtěl bych poděkovat Ing. Michalovi Valentovi, Ph.D. za vřelý přístup a cenné rady, které mi poskytl během tvorby této práce. Chci také poděkovat své rodině za možnost studovat vysokou školu.

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. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Jindřich Žák. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Žák, Jindřich. *Integrace a nasazení mobilní aplikace pro studenty znakového jazyka v organizaci Tichý svět.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Práce se zabývá vývojem multiplatformní mobilní aplikace pro výuku znakového jazyka a její integrací s existujícím e-learningovým systémem. Mobilní aplikace staví na funkčním prototypu vytvořeném v rámci bakalářské práce Alžbety Gogolákové s použitím nástroje Flutter. Pro integraci s webovým e-learningovým portálem postaveném na systému Moodle vznikla webová služba s použitím PHP frameworku Symfony. Webová služba čte data z existující databáze se studijními materiály a provádí autentizaci uživatelů s využitím JWT tokenů. Dokončená mobilní aplikace s webovou službou komunikuje a je připravena na produkční použití. Prostřednictvím služby GitHub Actions byly implementovány automatizované skripty kontrolující kvalitu kódu aplikováním přístupu Continuous Integration. Dále došlo k implementaci Continuous Delivery skriptů, které provádějí sestavení aplikace pro systémy iOS a Android a její automatické doručení do obchodů s aplikacemi Google Play a AppStore. Pro zjednodušení a zrychlení distribuce testovacích verzí aplikace byla zřízena možnost jejího běhu a nasazení do webového prostředí. Práce přináší užitečné know-how týkající se automatizace opakujících se úkolů během vývoje mobilních aplikací.

Klíčová slova Flutter, vývoj mobilních aplikací, Symfony, Json Web Token, autentizace, GitHub Actions, Continuous Integration, Continuous Delivery

Abstract

The thesis describes development of a multi-platform mobile application for sign language education and its integration with an existing e-learning system. The mobile application is based on a functional prototype created during Alžbeta Gogoláková's bachelor thesis using the Flutter framework. For integration with the web-based e-learning portal built on top of Moodle, a web service was created using the PHP framework Symfony. The web service reads data from an existing database with study materials and performs user authentication using JWT tokens. The completed mobile application communicates with the web service and is ready for production use. Automated quality control scripts using GitHub Actions have been implemented by applying the Continuous Integration approach. In addition, Continuous Delivery scripts were implemented to build the iOS and Android app and automatically deliver it to the Google Play and AppStore. To simplify and speed up the distribution of test versions of the app, the ability to run and deploy it to a web environment was established. The thesis provides useful know-how regarding the automation of repetitive tasks during mobile application development.

Keywords Flutter, mobile app development, Symfony, Json Web Token, authentication, GitHub Actions, Continuous Integration, Continuous Delivery

Obsah

Úvod **1**

1 Analýza **3**
- 1.1 Současný stav . . . 3
  - 1.1.1 Výuka znakového jazyka . . . 3
  - 1.1.2 E-learning pro kurzy Tichého jazyka . . . 4
  - 1.1.3 Role mobilní aplikace pro Tichý jazyk . . . 5
- 1.2 Prototyp aplikace od Alžbety Gogolákové . . . 5
- 1.3 Potřebné úpravy pro dokončení aplikace . . . 7
- 1.4 Analýza dat . . . 9
  - 1.4.1 Struktura studijních materiálů v systému Moodle . . . 9
  - 1.4.2 Uživatelé v databázi Moodle . . . 11
- 1.5 Funkční a nefunkční požadavky na webovou službu . . . 12
- 1.6 Analýza technologií . . . 13
  - 1.6.1 Moodle Web API . . . 13
  - 1.6.2 Symfony . . . 16
  - 1.6.3 Bezpečnost komunikace . . . 16
  - 1.6.4 Automatizace procesů . . . 21

2 Návrh řešení **23**
- 2.1 Webová služba . . . 23
  - 2.1.1 Volba frameworku . . . 24
  - 2.1.2 Autentizace a autorizace . . . 24
  - 2.1.3 REST API . . . 25
- 2.2 Mobilní aplikace . . . 28
  - 2.2.1 Offline mód . . . 28
  - 2.2.2 Vyhledávání . . . 29
- 2.3 Strategie větví v nástroji Git . . . 29
  - 2.3.1 GitFlow . . . 30

2.3.2 Trunk-Based Development . . . 30
2.3.3 Volba strategie . . . 31
2.4 Continuous Integration a Delivery . . . 32

3 Implementace webové služby **35**
3.1 Zprovoznění vývojového prostředí . . . 35
3.1.1 Docker . . . 35
3.1.2 Docker Compose . . . 36
3.1.3 Databáze . . . 37
3.1.4 Prostředí pro běh PHP . . . 37
3.1.5 Moodle . . . 40
3.2 Vývoj Symfony aplikace . . . 41
3.2.1 Modelové třídy . . . 41
3.2.2 Controller . . . 42
3.2.3 Datová vrstva . . . 43
3.2.4 Autentizace uživatelů . . . 45
3.2.5 Vývojový HTTP klient a monitoring chyb . . . 46
3.2.6 Continuous Integration . . . 48

4 Implementace aplikace **51**
4.1 Aktualizace Flutter závislostí . . . 52
4.2 Tvorba uživatelského rozhraní . . . 54
4.3 State management . . . 55
4.3.1 Pomíjivý a sdílený stav . . . 55
4.3.2 Přístupy pro správu stavu . . . 55
4.3.3 Business Logic Component . . . 56
4.3.4 Implementace s použitím BLoC patternu . . . 57
4.4 Videoslovník . . . 61
4.5 Komunikace s API . . . 62
4.5.1 Klientské třídy . . . 62
4.5.2 HTTP cache . . . 63
4.6 Nasazení aplikace na web . . . 64
4.6.1 Nekompatibilní závislosti . . . 65
4.6.2 Definice odlišných prostředí s .env . . . 67
4.6.3 Nasazení testovacích verzí . . . 68
4.7 Přechod na null safety . . . 69
4.7.1 Výhody a změny v jazyce . . . 69
4.7.2 Migrace projektu . . . 70
4.8 Změny v průběhu vývoje . . . 70
4.8.1 Odstranění podpory offline módu . . . 71
4.8.2 Grafické úpravy . . . 71
4.9 Testy . . . 71
4.9.1 Jednotkové testy . . . 73
4.9.2 Widget testy . . . 73

4.9.3 Uživatelské testy 75
4.9.4 Flutter driver 75

5 Sestavení aplikace a nasazení **77**
5.1 GitHub Actions 77
5.1.1 Optimalizace doby běhu 78
5.2 Automatizace sestavení a nasazení 79
5.2.1 Fastlane 79
5.2.2 iOS 80
5.2.3 Android 82
5.2.4 Skrytí tajemství v CI/CD systému 84
5.3 Monitoring chyb 86
5.3.1 Mobilní aplikace 87
5.3.2 Webová služba 88

Závěr **89**

Seznam literatury **91**

A Seznam použitých zkratek **97**

B Testovací scénáře **99**

C Obsah přiloženého USB flash disku **101**

Seznam obrázků

1.1 Uživatelské rozhraní prototypu mobilní aplikace pro Tichý jazyk . 6
1.2 Struktura studijních materiálů Tichého jazyka v systému Moodle . 10

2.1 Sekvenční diagram znázorňující interakci mobilní aplikace a webové služby během autentizace 26

3.1 Repozitář Docker obrazů v systému GitLab 40
3.2 ER diagram znázorňující entity v systému Moodle potřebné pro čtení dat studijních materiálů 44

4.1 Nasazení mobilní aplikace na web 65
4.2 Snímky obrazovek aplikace v novém grafickém designu 72

5.1 Průběh GitHub Actions *jobu* sestavujícího iOS aplikaci s nahráním do AppStore Connect 81
5.2 Průběh GitHub Actions *jobu* sestavujícího Android aplikaci s nahráním do Google Play Console 85

Seznam tabulek

1.1 Pokrytí funkčních požadavků dostupnými funkcemi z Moodle Web services API 15

2.1 Návrh REST API endpointů 27

Úvod

Metody pro tvorbu multiplatformních mobilních aplikací zaznamenaly v posledních letech rychlý vývoj. Vznikají nástroje, které si kladou za cíl vývojářům usnadnit práci při jejich tvorbě a ušetřit čas strávený jejich vývojem. Jedním z takových nástrojů je i Flutter, který do světa vývoje mobilních aplikací přináší řadu nových přístupů a myšlenek.

Právě v nástroji Flutter vznikl prototyp aplikace pro výuku českého znakového jazyka v organizaci Tichý svět. Prototyp byl vytvořen studentkou FIT ČVUT Alžbetou Gogolákovou v rámci její bakalářské práce.

Mobilní aplikace však často splňují pouze funkci prezentační vrstvy v rámci nějakého většího systému. Ani v případě mobilní aplikace pro výuku znakového jazyka tomu není jinak. Proto se tato práce věnuje její integraci s již existujícím systémem pro výuku znakového jazyka v organizaci Tichý svět. Důraz při integraci je kladen na zabezpečenou komunikaci a autentizaci uživatelů.

V práci se věnuji nejdříve analýze celého problému. Konkrétně se seznamuji s činností organizace Tichý svět a jejím současným prostředím pro výuku znakového jazyka. Kromě analýzy domény se věnuji i technickým aspektům současného řešení a vhodným technologiím pro realizaci integrace mobilní aplikace s existujícím systémem Tichého světa.

Na základě analýzy sestavuji návrh celého řešení tak, aby byly splněny všechny požadavky na funkčnost aplikace a její zabezpečenou komunikaci se serverem. Kapitoly věnující se implementaci jsou rozděleny zvlášť pro webovou službu a mobilní aplikaci. Popisuji v nich důležitá rozhodnutí, která bylo potřeba v průběhu vývoje řešit.

Kromě integrace mobilní aplikace s webovou službou se práce zabývá i jejím nasazením. V posledních letech je trendem automatizovat co nejvíce procesů při vývoji software pro dosažení vyšší efektivity práce a minimalizace chyb způsobených člověkem. Otázce automatizace v kontextu vývoje mobilních aplikací se věnuji v předposlední kapitole.

Kapitola **1**

Analýza

1.1 Současný stav

Tichý svět je nezisková organizace působící v České republice již řadu let. Jejím cílem je pomoci začleňovat neslyšící osoby do světa slyšících různými prostředky a zároveň šířit osvětu o jejich životě a kultuře. Organizace poskytuje pro neslyšící všestrannou podporu. Kromě sociálního poradenství provozuje řadu souvisejících služeb a projektů jako například online tlumočení znakového jazyka, kavárnu s neslyšící obsluhou či kurzy znakového jazyka pod názvem Tichý jazyk. Právě Tichého jazyka se týká tato diplomová práce.

1.1.1 Výuka znakového jazyka

Kurzy Tichý jazyk poskytují výuku českého znakového jazyka. Výuka probíhá standardně prezenčně a pro její podporu existuje webový portál s e-learningem postavený na systému Moodle, který nabízí možnost domácího studia. Do e-learningového portálu se studenti mohou přihlásit pomocí účtu, který jim byl vytvořen.

Struktura výukových materiálů v e-learningu je hierarchická. Materiály jsou v nejvyšší vrstvě rozděleny do několika modulů, které představují větší celky výuky znakového jazyka. Studenti mají zpočátku přístup pouze k omezenému počtu modulů a s tím, jak jejich studium postupuje, mohou získávat přístup do dalších modulů. Přístup k novým modulům uděluje administrátor e-learningového portálu.

Moduly obsahují sadu lekcí, které se vždy věnují nějakému konkrétnímu tématu. Existují například lekce věnující se tématům jako je *rodina*, *koníčky* nebo *jídlo*. Základem každé lekce v e-learningovém portálu je sada slovíček ve formě videí. Konkrétně se jedná o několikavteřinová videa, na kterých lektor ukazuje dané slovo ve znakovém jazyce.

Mimo jednotlivých slov lekce ještě často obsahují i videa s celými větami ve znakovém jazyce ukazující použití slovíček. Jedna lekce v e-learningu se tak

typicky skládá z pár desítek krátkých videí se slovíčky a větami. Video zobrazující celou větu bývá často doprovázeno kratším textovým popisem s jednotlivými znaky, které tvoří danou větu. Například věta *„Nepovedlo se mi upéct bábovku, vyhodím ji.“* se skládá ze znaků *„JÁ + BÁBOVKA + UPÉCT + NEPOVEDLO + VYHODÍM“*.

Lekce obsahují kromě videí ještě další materiály jako jsou obrázky k vytisknutí, které mohou sloužit jako podklad pro cvičení znakového jazyka. Jelikož se jedná o obrázky určené primárně k vytisknutí jako podklad pro cvičení znakového jazyka, jejich prezentování v mobilní aplikaci nemá příliš velký význam. Aplikace je navíc od počátku určena výhradně k přehrávání videí se slovíčky a větami.

Kromě výukových modulů obsahuje e-learning ve svých materiálech ještě takzvaný *Videoslovník*. Jde o seznam téměř tisíce videí s lektory, kteří ukazují jednotlivá slovíčka ve znakovém jazyce stejně jako ve studijních modulech. Videa jsou jako v běžném papírovém slovníku seřazena podle abecedy. Existuje zde i jednoduché vyhledávání. Množina videí se slovíčky v modulech a ve *Videoslovníku* není totožná, ale má velký průnik.

1.1.2 E-learning pro kurzy Tichého jazyka

Webový e-learning Tichého jazyka se nachází na adrese `http://www.e-tichyjazyk.cz` a je postaven na systému Moodle. Moodle je široce používaný výukový systém napsaný v jazyce PHP a používající relační SQL databázi, ve které jsou uloženy veškeré potřebné učební materiály, uživatelské účty a další data systému Moodle. Aplikace běží na běžném hostingu s PHP a databází MariaDB.

Výše popsaná struktura studijních materiálů je do značné míry ovlivněna právě strukturou dat v Moodle. Struktuře dat v databázi se konkrétněji věnuji v kapitole 1.4 Analýza dat. Ačkoliv je struktura a uživatelské rozhraní Moodle navrženo pro poměrně odlišný charakter materiálů, než je velká kolekce krátkých videí, webový portál je již zaběhnutý a svou funkci i přes své nedostatky zvládá plnit.

Všechna obsažená videa jsou uložena na YouTube účtu Tichého světa a jsou označena jako neveřejná, takže je bez znalosti odkazu nelze nalézt. Do prostředí Moodle jsou vkládána ve formě `iframe` HTML elementů prostřednictvím textového WYSIWYG[1] editoru, který je součástí portálu Moodle. Spolu s `iframe` elementem jsou součástí textového pole někdy i krátké popisky informující například o jednotlivých znacích ve větě. Tyto popisky jsou pro výuku důležité a je potřeba je v aplikaci zahrnout.

[1]editor zobrazuje zformátovaný text, který je vnitřně reprezentován pomocí HTML; z angl. *What You See Is What You Get*

1.1.3 Role mobilní aplikace pro Tichý jazyk

Na základě online schůzek se zástupci vedení Tichého světa by měla mobilní aplikace sloužit jako doplňková služba k webovému e-learningovému portálu. Primárním zdrojem materiálů by tedy i nadále měl být zaběhnutý webový portál, který je vhodné prohlížet na počítači. Webový e-learningový portál ctí responzivní webový design, a tak se v něm lze poměrně dobře pohybovat i na mobilním telefonu. Navzdory responzivitě se ale zejména videa zobrazují v mnohem větší šířce, než by bylo potřeba, a proto je použití webové verze e-learningu na mobilním telefonu poměrně nepříjemné. Účelem mobilní aplikace by tak mělo být poskytování snadného přístupu k videím se slovíčky a větami ve chvílích, kdy studenti nemají přístup k počítači.

Zde stojí za to zmínit, že existuje oficiální mobilní aplikace pro systém Moodle, na kterém je e-learningový portál postaven. Ta by měla být schopna uživatelům zobrazit materiály z webového e-learning portálu ve formě vhodné pro mobilní telefon. Aplikace ale zřejmě není příliš dobře udržovaná a v obchodech s aplikacemi si vysloužila nepříliš dobré hodnocení (4,2 hvězdičky na Apple AppStore[2], 3,5 hvězdičky na Google Play[3]).

Vývojem vlastní aplikace speciálně pro potřeby Tichého jazyka vzniká možnost vyvinout uživatelské prostředí na míru pro potřeby studentů znakového jazyka. Další výhodou je i možnost aplikaci prezentovat a publikovat čistě pod značkou Tichého jazyka. Z těchto důvodů je dle mého názoru vývoj vlastní aplikace oproti použití mobilní aplikace Moodle mnohem lepším řešením.

1.2 Prototyp aplikace od Alžbety Gogolákové

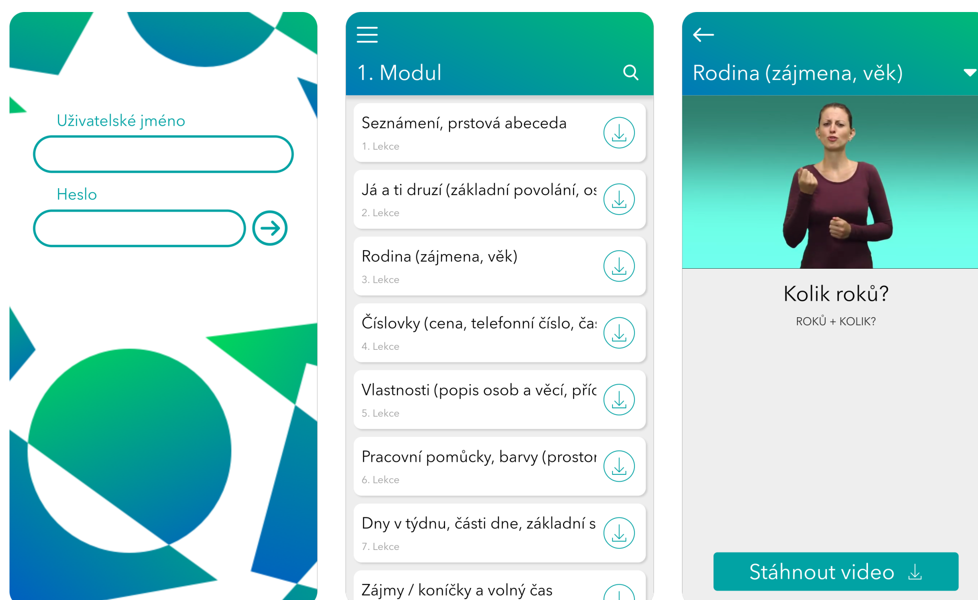
Aplikace, která vznikla v rámci praktické části bakalářské práce Alžbety Gogolákové, je ve stavu funkčního prototypu. Je napsána v jazyce Dart s použitím frameworku Flutter a je díky tomu připravena pro spuštění na operačních systémech Android i iOS.

Aplikace byla od začátku navrhována tak, aby poskytovala následující funkcionalitu, která plyne z funkčních požadavků v práci Alžbety Gogolákové: [1]

- přihlašování uživatelů,
- zobrazování výukových modulů a přehrávání videí se slovíčky a větami,
- fungování aplikace v omezeném režimu bez přístupu k internetu a možnost stažení videí do lokálního úložiště,
- vyhledávání videí pomocí zabudovaného vyhledávače,

[2] https://apps.apple.com/app/moodle/id633359593

[3] https://play.google.com/store/apps/details?id=com.moodle.moodlemobile



Obrázek 1.1: Uživatelské rozhraní prototypu mobilní aplikace pro Tichý jazyk

Na všechny zmíněné požadavky bylo uživatelské rozhraní prototypu připraveno.

Uživatelské prostředí

Prototyp aplikace se skládá z několika obrazovek, které uživatelům umožňují procházet hierarchickou strukturu učebních materiálů. První obrazovku, kterou uživatel po otevření aplikace uvidí, je jednoduchý přihlašovací formulář. Po vyplnění přihlašovacích údajů se prototyp aplikace pokusí nasimulovat přihlášení. V případě validních údajů přesměruje uživatele na obrazovku pro prohlížení výukových materiálů.

Struktura obrazovek po úspěšném přihlášení počítá stejně jako webový e-learning s výukovými moduly, které seskupují jednotlivé lekce do větších celků. Výběr modulu je možný z levého vysouvacího panelu (*navigation drawer*).

Po výběru modulu se uživatel přesouvá na obrazovku se seznamem lekcí vybraného modulu. U každé lekce je tlačítko ve formě ikonky pro možnost stažení dané lekce do offline paměti zařízení. Po rozkliknutí vybrané lekce se uživatel dostane na obrazovku s detailem lekce. Hlavním prvkem této obrazovky je přehrávač videa. Jelikož se v dané lekci vyskytuje více videí, uživatel se mezi nimi může pohybovat horizontálním posouváním do stran po obrazovce. U každého videa je opět tlačítko pro stažení daného videa do paměti telefonu.

V horním panelu doprovází uživatele napříč téměř celou aplikací ikonka lupy pro možnost vyhledávání ve videích. Vyhledávání je v aplikaci řešeno třemi způsoby. Prvním způsobem je vyhledávání pouze v rámci aktuálně zobrazeného modulu. Dále je možné vyhledávat pouze v rámci offline stažených videí. Posledním způsobem, kterým je možné vyhledávat ve videích, je vyhledávání napříč všemi moduly. Pro tento způsob existuje v levém vysouvacím panelu položka „Vyhledávání". O rozsahu vyhledávání je uživatel vždy informován krátkým textem pod vyhledávacím políčkem.

V levém vysouvacím panelu se kromě samotných modulů, položky „Vyhledávání" a položky „Odhlásit se" vyskytuje ještě položka „Stažená videa". Po jejím rozkliknutí se uživatel dostává na obrazovku se seznamem všech aktuálně stažených videí rozdělených podle jejich příslušných lekcí a modulů.

Prostředí pro běh prototypu

Z důvodu, že prototyp aplikace není napojen na žádné API postaveném na reálných datech Tichého jazyka, byl Alžbetou v rámci vývoje zřízen SSH server na školním serveru Fray, kde byla uložena testovací data. Prototyp aplikace tedy umí jednoduše komunikovat se vzdáleným SSH serverem, na kterém očekává data uložená v jednom souboru ve formátu JSON. Strukturu dat přikládám pro ukázku ve zdrojovém kódu 1.

Jedná se o data postavená na malém vzorku produkčních dat pro potřeby prototypu aplikace. V JSON datech jsou referencována videa, která se mají uživateli zobrazovat v rámci jednotlivých lekcí. Tato videa byla opět uložena a zpřístupněna aplikaci na Alžbetině SSH serveru. Všechny soubory, které jsou potřebné k naplnění aplikace daty pro účely testování, mi Alžbeta poskytla.

Testovací data obsahují data dvou uživatelů pro demonstraci rozdílných přístupových práv k modulům. Přihlašovací údaje jsou ověřovány proti uživatelským jménům a heslům uloženým prozatím přímo v prototypu aplikace. Ověření přihlašovacích údajů je tedy realizováno jen pro ukázku a není připraveno na produkční použití.

1.3 Potřebné úpravy pro dokončení aplikace

Komunikace mezi serverem a aplikací

Zásadním úkolem pro dokončení aplikace bude samozřejmě implementovat komunikaci mezi mobilní aplikací a serverem s výukovými materiály. Vznikne tedy nejspíše nějaká webová služba, která bude mobilní aplikaci poskytovat data s výukovými materiály i samotná videa.

V aktuální situaci, kdy jsou videa uložena na YouTube, není možné do mobilní aplikace implementovat funkcionalitu na jejich stažení pro offline režim. Proto bude nejspíš nutné videa přemístit na vlastní server, odkud bude možné jejich stažení. Pokud budou videa přesunuta na webový hosting, na kterém

```
{
  "modules": [
    {
      "id": 1,
      "lectures": [
        {
          "id": 1,
          "name": "Seznámení",
          "description": "Seznámení, prstová abeceda",
          "number": 1,
          "module_id": 1,
          "videos": [
            {
              "id": 1,
              "name": "Seznámit se",
              "ftp_location": "bachelor/videos/seznamit_se.mp4",
              "ytb_location": "https://youtu.be/GnFOvKu-wX4",
              "is_sign": 1,
              "lecture_id": 1
            },
            ...
          ]
        },
        ...
      ]
    },
    ...
  ]
}
```

Zdrojový kód 1: Ukázka testovacích dat pro prototyp mobilní aplikace

běží i samotný Moodle, bude nutné brát ohled na případné limity přenesených dat hostingu.

Další z úprav, kterou bude potřeba realizovat, je zajistit načítání pouze těch dat ze serveru, které aplikace v danou chvíli potřebuje, namísto jednorázového stažení veškerých dat. Pro účely prototypu s malým vzorkem dat nebyl tento jednoduchý přístup problémem. Avšak s produkčními daty, která obsahují stovky videí, by tento přístup vedl k zbytečné zátěži jak pro server, tak pro mobilní aplikaci. Ta by totiž musela přenášet zbytečně velký objem dat, který by následně musela zpracovat z formátu JSON, což může mít negativní vliv na výkon aplikace. [2]

Zásadním úkolem bude zároveň implementovat autentizaci a autorizaci

uživatelů na základě dat uložených ve webovém e-learningovém portálu. Existující databáze se tak stane zdrojem jednotné pravdy pro webový e-learning i mobilní aplikaci. Pro administrátory kurzů to navíc nebude znamenat žádnou změnu od současného způsobu správy studijních materiálů, vytváření uživatelů a úpravám jejich přístupu k učebním modulům.

Úpravy aplikace

Během jedné z videoschůzek s Tichým světem se objevily nové požadavky na funkcionalitu, kterou prototyp dosud neposkytoval. Prvním požadavkem bylo zakomponování *Videoslovníku* z e-learningu do mobilní aplikace.

S druhým nápadem přišel pan Leoš Mačák, ředitel Tichého světa. Navrhl přidat do aplikace sekci *Moje knihovna*, což by byla kolekce videí, kam by si studenti kurzů mohli sami ukládat vybraná videa k pozdějšímu zhlédnutí. Typickým případem užití by bylo, že uživatel si chce uložit slovíčka, která mu dělají problém a chce si je tedy později procvičit.

Společně jsme dospěli k závěru, že pro uživatelskou jednoduchost bude stačit jediná kolekce. Možnost vytvořit si více vlastních kolekcí by pravděpodobně akorát způsobila vyšší složitost celého procesu její tvorby. Také jsme se dohodli, že videa v *Mojí knihovně* budou seřazena stejně jako v rámci učebních modulů a lekcí.

1.4 Analýza dat

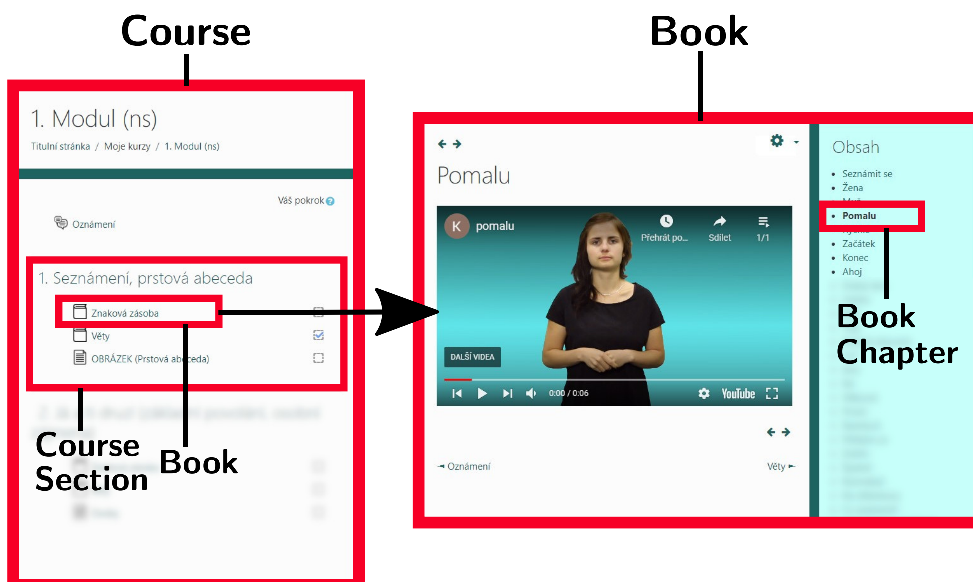
Na tomto místě je dobré si trochu přiblížit strukturu uložených dat v databázi systému Moodle za účelem seznámení se s často používanými názvy entit, se kterými Moodle pracuje, a vztahy mezi nimi.

Protože Moodle po své instalaci odstraňuje z databáze komentáře a relace pomocí cizích klíčů [3], nebylo vždy úplně jednoduché z databáze zjistit vzájemné vztahy mezi entitami. S tímto úkolem mi značně pomohl projekt Examulator.com[4], který se snaží o zmapování a dokumentaci schémata Moodle databáze.

1.4.1 Struktura studijních materiálů v systému Moodle

V začátcích mé činnosti na této diplomové práci mi byl zpřístupněn IT správcem Tichého světa SQL *dump* databáze. Aby nedošlo k uvolnění citlivých uživatelských dat, byla tato data anonymizována. Učební materiály ale zůstaly ve své původní formě. Databáze systému Moodle se skládá ze zhruba čtyř stovek tabulek. Relevantní data pro účely této práce jsou však uložena v jen malém zlomku všech tabulek. Ty nejdůležitější z nich popisuji v následujících odstavcích a pro snadnější představu je jejich použití ve webovém portálu znázorněno na obrázku 1.2.

[4]https://www.examulator.com/er/output/index.html



Obrázek 1.2: Struktura studijních materiálů Tichého jazyka v systému Moodle

Course

Základní entitou v systému Moodle je entita *Course* (tabulka `mdl_course`). Dle dat z databáze Tichého světa je tato entita využívána pro ukládání jednotlivých učebních modulů znakového jazyka. Pro přiblížení entity *Course* zde krátce uvedu, že například ČVUT ve své vlastní instanci Moodle má pod touto entitou uložené jednotlivé předměty, což se mnohem více blíží původně zamýšlenému použití této entity.

Course section

Entitou, která je na *Course* navázána vztahem 1:n, je *Course Section* (tabulka `mdl_course_sections`) a v kontextu Tichého jazyka představuje konkrétní lekci. *Course Section* v sobě nese hlavně informaci o svém názvu. Vzhledem k tomu, že tato entita ještě určuje to, co bude ve webovém rozhraní Moodle vykresleno na stránce detailu entity *Course*, *Course Section* si v sobě nese i informaci o tom, na kolikátém místě se má daná sekce zobrazit.

Vztah mezi lekcí a videem

Ve webovém rozhraní se každá lekce Tichého jazyka skládá z několika videí a případně PDF dokumentů s obrázky. Pro účely aplikace jsou důležitá hlavně videa, proto se ostatními typy obsahu nebudu příliš zabývat.

Entita *Course Section*, která představuje jednu lekci, se dále větví na entity typu *Book* (tabulka `mdl_book`). Ta tvůrcům obsahu umožňuje vytvořit kolekci stránek, na kterých může být prakticky libovolný obsah. Právě na těchto obsahových stránkách mají studenti k dispozici materiály ke studiu včetně videí.

Konkrétní jednu obsahovou stránku reprezentuje entita *Book Chapter* (tabulka `mdl_book_chapters`). V Tichém světě se tyto obsahové stránky typicky využívají k zobrazení videí se slovíčky nebo větami. Neplatí to ale vždy, na některých obsahových stránkách se vyskytují například obrázky či texty bez videa. Někdy je na stránce naopak videí několik. Důležitým sloupcem v tabulce `mdl_book_chapters` je sloupec `content`. Ten v sobě nese HTML dokument vytvořený v textovém WYSIWYG editoru Moodle, který je potom vykreslen jako tělo dané obsahové stránky. V případě videa z YouTube je zde k nalezení již dříve zmiňovaný HTML element `iframe` s URL odkazujícím na samotné video.

Podobnou funkci jako dvojice entit *Book* a *Book Chapter* zde hrají ještě entity *Glossary* a *Glossary Entry*. Ty se používají pro uložení *Videoslovníku*, respektive jednotlivých slovíček v něm obsažených.

Jelikož vztah mezi *Book* a *Course Section* není v ER diagramu v aplikaci Examulator.com zachycen, musel jsem tento vztah reverzně zjistit z kódu projektu Moodle. Odůvodnění absence tohoto vztahu v dokumentaci Examulator.com je takové, že jde o jakýsi „dynamický“ vztah, ve kterém mohou na jedné straně figurovat entity různých typů. Tyto entity Moodle souhrnně nazývá pojmem „aktivita“. Aktivitou může být například právě entita typu *Book*, *Glossary* a další.

Určitou formu vazební tabulky zde plní tabulka `mdl_course_modules`, která jednak obsahuje cizí klíč *Course Section* a jednak i cizí klíč vázané aktivity. A právě o konkrétním typu aktivity (konkrétní tabulce) rozhoduje třetí sloupec v tabulce `mdl_course_modules` s názvem `module`. Ten vede přes další vztah na tabulku `mdl_module`, což je ve skutečnosti číselník typů aktivit. Pro účely mobilní aplikace mě budou prakticky výhradně zajímat pouze zmíněné typy *Book* a *Glossary*, které se v e-learningu Tichého jazyka používají.

1.4.2 Uživatelé v databázi Moodle

Moodle své uživatelské účty ukládá do tabulky `mdl_user`. V ní jsou uloženy informace jako jméno a příjmení, přihlašovací jméno, email, zahashované heslo a příznaky týkající se stavu uživatelského účtu. Pro hashování hesel uživatelů používá Moodle od verze 2.5 funkci bcrypt. [4]

Bcrypt je poměrně široce používaná funkce pro hashování hesel. Je obsažená v řadě open-source projektů a frameworků a je postavená na šifře Blowfish. [5, 6] Na obranu proti útoku s použitím rainbow tables zajišťuje bcrypt automaticky pro každé heslo svou vlastní sůl, která je pak i součástí finální hashe.

Finální hash je ve formátu *modular crypt*, který stanovuje podobu hashe pro různé algoritmy. Začíná prefixem, který identifikuje použitý algoritmus. V případě funkce bcrypt bývají prefixem zpravidla hodnoty `$2a$` nebo `$2b$`. Za prefixem následuje parametr *work factor*, který určuje náročnost výpočtu. Dále je ve výsledné hashi uložena sůl a samotné zahashované heslo. [7] Z tohoto popisu vyplývá, že pro ověření správnosti uživatelského hesla stačí pouze tato samotná finální hash hodnota.

Pro autorizaci přístupu ke studijním materiálům jsou důležité zejména tabulky `mdl_enrol` a `mdl_user_enrolments`, které nesou informace o tom, do jakého kurzu v rámci struktury Moodle byl uživateli udělen přístup. Moodle rozlišuje přístup pouze na úrovni entit typu *Courses*. Jakmile pro daného uživatele existuje záznam o zápisu do kurzu, má přístup ke všem v něm obsaženým materiálům.

1.5 Funkční a nefunkční požadavky na webovou službu

Na základě předešlé analýzy a schůzek s představiteli Tichého světa jsem sestavil následující seznam funkčních a nefunkčních požadavků, které by měla webová služba poskytující data mobilní aplikaci splňovat.

Funkční požadavky

- FP1 – Získání seznamu modulů
- FP2 – Získání seznamu lekcí daného modulu
- FP3 – Získání seznamu videí dané lekce
- FP4 – Získání videí z *Videoslovníku*
- FP5 – Vyhledávání v rámci všech videí
- FP6 – Vyhledávání videa v rámci modulu
- FP7 – Autentizace uživatelů
- FP8 – Autorizace uživatelů
- FP9 – Přehrání a stažení videa ze serveru

Nefunkční požadavky

- NP1 – Postaveno na PHP a MariaDB
- NP2 – Dodržení datových limitů definovaných v FUP hostingu

- NP3 – Bezpečnost při přihlašování uživatelů
- NP4 – Monitoring chyb vzniklých za běhu

1.6 Analýza technologií

V následující podkapitole se věnuji analýze technologií, které by bylo možné použít zejména pro implementaci komunikace mezi serverem a mobilní aplikací.

1.6.1 Moodle Web API

Samotný Moodle poskytuje rozhraní přístupné přes protokol HTTP pro nejrůznější manipulaci s daty. Toto rozhraní je například využíváno oficiální mobilní aplikací Moodle. [8]

Zpřístupnění Moodle Web services API je možné z administrátorského nastavení ve webovém rozhraní. Zde je nutné nejdříve globálně zapnout funkcionalitu webových služeb, nastavit protokol (na výběr je SOAP, REST, XML-RPC a další) a přístup uživatelů k webovému API. Moodle totiž umožňuje specifikovat vybrané uživatelské účty, které budou mít k API přístup. Druhou možností je zpřístupnění webového API všem uživatelům. [8]

Ačkoliv Moodle ve své dokumentaci používá zkratku REST, tato konkrétní služba se od standardního pojetí REST odklání. Místo orientace na data, která je pro REST typická [9], pracuje Web services API s pojmem „funkce“ (*function*), která je po volání HTTP metody volána. Funkce v kontextu Web services API má nějaké vstupní parametry a vrací nějaká data. Každá funkce má unikátní název. Ten je v momentě volání předán v podobě query parametru `wsfunction`. Pokud daná funkce pracuje s nějakými parametry, ty jsou taktéž předány jako součást URL *query*. [10]

Moodle navíc používá pojem „služba“ (*service*) jako objekt, který v sobě sdružuje jednotlivé funkce. Když se v administraci systému Moodle provádí konfigurace webových služeb, je potřeba definovat, který uživatel má přístup k jaké službě, tedy k množině funkcí. Pro dodržení optimální bezpečnosti je samozřejmě vhodné uživatelům zpřístupňovat pouze takové funkce, které uživatel bude potřebovat. Mimo jiné z těchto důvodů si s sebou Moodle již nese zabudovanou webovou službu zvanou „Moodle web services“ sdružující funkce, kterých využívá například oficiální mobilní aplikace Moodle. [8]

Interakce s Moodle Web services API používá ověřovací token, který získá klient ze serveru výměnou za uživatelské jméno a heslo. [11] K ověření, zda má uživatel oprávnění danou funkci volat, je spolu s HTTP dotazem odeslán i přidělený token, který je odeslán jako URL *query* parametr `wstoken`. Ten lze získat na následující URL, kde musí klient správně vyplnit uživatelské jméno, heslo a tzv. *service short name*, což je krátký řetězec identifikující

požadovanou službu, se kterou chce klient posléze pracovat. Výsledná URL je tedy v následujícím tvaru:

```
https://mymoodle.com/login/token.php?username=USERNAME&
    password=PASSWORD&service=SERVICESHORTNAME
```

Dalším důležitým parametrem při volání webových služeb je parametr `moodlewsrestformat`, kterým je možné specifikovat, jaký formát dat si klient v HTTP odpovědi přeje. Na výběr je možné použít hodnoty `xml` nebo `json`.

Vzhledem k tomu, že Moodle ověřuje přihlášeného uživatele pomocí vygenerovaného tokenu, tento token si drží v databázi spolu s informací o jeho držiteli a datu expirace. Po expiraci tokenu je potřeba zažádat s přihlašovacími údaji o nový. Při každém volání funkce s ověřením pomocí tokenu tak musí Moodle interně pokaždé ověřit platnost tokenu s pomocí databáze.

V situacích, kdy klientovi nestačí předem vytvořené funkce pro jeho potřeby, umožňuje Moodle rozšiřitelnost pomocí pluginů. Plugin může implementovat svou vlastní funkci, která může být následně pomocí Moodle Web services API zpřístupněna. [10]

Pro případ, že bych se dále vydal cestou využít Moodle Web services API, jsem se na základě dokumentace webových funkcí[5] pokusil pokrýt všechny funkční požadavky dostupnými funkcemi z Web services API. Toto pokrytí dokumentuje tabulka 1.1.

[5] `https://docs.moodle.org/dev/Web_service_API_functions`

Funkční požadavek	Funkce pokrývající požadavek
FP1 – Získání seznamu modulů	• core_enrol_get_users_courses – získání seznamu kurzů, do kterých má student přístup • core_course_get_courses – detail kurzu
FP2 – Získání seznamu lekcí daného modulu	• core_course_get_contents
FP3 – Získání seznamu videí (kapitol) dané lekce	• mod_book_get_books_by_courses
FP4 – Získání videí z *Videoslovníku*	• mod_glossary_get_glossaries_by_courses
FP5 – Vyhledávání v rámci všech videí	• mod_data_search_entries – globální vyhledávání • mod_glossary_get_entries_by_search – vyhledávání ve *Videoslovníku*
FP6 – Vyhledávání videa v rámci modulu	• nelze
FP7 – Autentizace uživatelů	• pomocí tokenu
FP8 – Autorizace uživatelů	• core_enrol_get_users_courses – získání seznamu kurzů, do kterých má student přístup
FP9 – Přehrání a stažení videa ze serveru	• závisí na způsobu uložení videa, pokud by se jednalo o soubory spravované systémem Moodle, lze použít funkci core_files_get_files

Tabulka 1.1: Pokrytí funkčních požadavků dostupnými funkcemi z Moodle Web services API

Poskytované funkce z Moodle Web services API pokrývají téměř všechny požadavky, které jsem pro webovou službu poskytující data definoval. Komplikovanější situace je ale u vyhledávání, kde Moodle sice poskytuje funkci `mod_data_search_entries`, která ale nedokáže vyhledávat pouze v rámci jednoho modulu. Dokumentace této funkce je celkově hodně stručná a pro člověka bez větších zkušeností práce se systémem Moodle může působit problémy.

Tím se koneckonců vyznačuje celá dokumentace webového API systému Moodle. Každá z funkcí je v dokumentaci popsána často pouze jednou větou a mnohdy tak není jednoznačné, k čemu daná funkce slouží. Vývojář je tak často odkázán na pochopení dané funkce z kontextu jako jsou parametry,

které funkce přijímá, nebo z testovacího zavolání funkce a prozkoumání jejího konkrétního výstupu.

1.6.2 Symfony

Alternativní cestou, jak implementovat webovou službu poskytující data mobilní aplikaci spolu s funkcionalitou autentizace a autorizace uživatelů, je naimplementovat separátní serverovou aplikaci s vlastní logikou. Z povahy mobilní aplikace Tichého jazyka stačí data z databáze pouze číst, žádný z požadavků nevyžaduje zápis dat do Moodle databáze.

Z důvodu, že webový e-learning běží na hostingu s PHP a MariaDB databází, dává smysl implementovat webovou službu pomocí stejných technologií, aby nebylo nutné zřizovat další server. Frameworkem, který tyto požadavky splňuje a mám s ním navíc osobní zkušenost, je Symfony. Jedná se o framework postavený na architektuře MVC. Symfony je ve skutečnosti sada balíků a knihoven poskytující nejrůznější funkcionalitu. Stažení správných verzí balíků má na starosti správce závislostí Composer, který je v ekosystému jazyka PHP dobře známý.

Pro tvorbu webového API pomocí Symfony je základem naimplementovat `Controller` třídu, která plní funkci zpracování HTTP dotazů. Implementace konkrétní `Controller` třídy obsahuje sadu metod, kde každá z nich zpracovává HTTP dotaz s jinou URL. Konfigurace jedné takové metody je zpravidla tvořena pomocí anotace `@Route`.

V případě webové služby, která čte data z již existující databáze, je důležité správně naimplementovat práci s databází. O to se ve frameworku Symfony stará knihovna Doctrine. Doctrine umožňuje práci s databází jak na úrovni SQL, tak pomocí object relation mappingu (ORM). ORM je možné realizovat pomocí anotací v entitních třídách. Spolu s každou entitní třídou je úzce spojena `Repository` třída, která poskytuje vysokoúrovňové API pro práci s objekty dané entitní třídy. Umožňuje uživateli implementovat si vlastní funkce, které mohou za jednoduchým rozhraním vykonávat komplikovanější SQL dotazy.

Funkcionalitou, která je pro účely webové služby pro Tichý jazyk důležitá, je přihlašování uživatelů. Pro tyto účely obsahuje Symfony balíček `Security`, který do celého projektu integruje ověření identity uživatelů. Pomocí konfigurace tohoto balíčku lze aplikovat různá pravidla pro ověřování přístupu uživatelů k vybraným API endpointům.

1.6.3 Bezpečnost komunikace

Pro zabezpečenou komunikaci je nutné zajistit řadu vlastností, které přinesou dostatečnou míru jistoty, že nehrozí neoprávněný zásah do komunikace nebo její zneužití. Základem je ověření identit aktérů komunikace – tedy serveru a klienta. Kromě ověření identity je potřeba zajistit i důvěrnost komunikace pro utajení přihlašovacích údajů klienta a dat přenášených ze serveru. Je po-

třeba se bránit i proti útokům, které by mohly způsobit změnu odesílaných dat a dalším úkolem je tedy zajistit integritu dat. Jak těchto vlastností docílit, se věnuji v následujících odstavcích.

Důvěrnost a ověření identity

Přihlašovací údaje klienta a data ze serveru nelze přenášet jednoduše v otevřeném textu. Aby nemohla nějaká třetí strana číst obsah přenášených dat, musí být šifrována. Takovou ochranu poskytuje protokol TLS (dříve SSL), který se běžně používá v kombinaci s HTTP protokolem ve formě HTTPS. TLS zaručuje kromě šifrování dat ještě jejich integritu, kterou je možné ověřit pomocí MAC kódu (*message authentication code*), který obsahuje kontrolní součet celé zprávy. [12] Protokol TLS není v síťovém TCP/IP modelu obsažen a musí tak být implementován na úrovni poslední aplikační vrstvy. Samotné aplikace však jeho implementaci často nemusí příliš řešit, protože se spoléhají na jeho implementace v používaných knihovnách. [13]

Protokol HTTPS tedy poskytuje bezpečný kanál pro komunikaci mezi serverem a mobilní aplikací. Dalším krokem je autentizace obou zúčastněných stran. Ověření identity serveru je nutné, aby si klientská strana mohla být jista, že se opravdu jedná o server Tichého jazyka na dané doméně. Ověření serveru lze provést na základě kontroly validity certifikátu, kterým se server v rámci HTTPS komunikace prokáže.

Ověření serveru konkrétně v případě frameworku Flutter zajišťuje automaticky třída `HttpClient`. V případě, že identita serveru nebyla ověřena nebo došlo k nezabezpečenému použití protokolu HTTPS, dojde k vyhození výjimky `TlsException`, což je koneckonců správně dle toho, co říká standard „HTTP over TLS“:

> *„Automated clients MUST log the error to an appropriate audit log (if available) and SHOULD terminate the connection (with a bad certificate error).“* [14]

Na druhé straně server ověřuje identitu klienta, tedy uživatele aplikace. V tomto případě jej ověří na základě znalosti kombinace přihlašovacího jména a hesla. Protože existuje mnoho způsobů, jak autentizaci uživatelů provádět, je potřeba si před její implementací rozebrat dostupné možnosti. Věnuji se jim v následující podkapitole.

Autentizace uživatelů

Jedním ze základních úkolů integrace mobilní aplikace s e-learningovým portálem Tichého jazyka je otázka autentizace uživatelů – tedy prokázání, že uživatel je skutečně tím, za koho se vydává. Před tím, než vznikne návrh a implementace autentizačního mechanismu, je dobré se vůbec seznámit s běžnými útoky na uživatelská hesla.

Dle [15] to jsou:

1. odchycení z provozu na počítačové síti,
2. získání pomocí malware,
3. vylákání hesla sociálním inženýrstvím (např. phishingovým útokem),
4. uhádnutí,
5. „lámání“ - tedy opakované pokusy o uhádnutí hrubou silou.

Proti odchycení z provozu na počítačové síti poskytuje ochranu šifrovaný protokol HTTPS. Otázka obrany proti malware je komplikovaná, protože záleží, jakou aktivitu pro získání hesel malware vykonává. Základem by ale mělo být pokud možno vyvarování se uložení přihlašovacích údajů v zařízení.

Proti sociálnímu inženýrství a uhádnutí hesla se však dá technickými prostředky bránit jen omezeně. Tato rizika lze minimalizovat osvětou uživatelů a vynucováním používání silných hesel.

V případě Tichého jazyka jsou nová uživatelská hesla zadávána skrz rozhraní portálu Moodle. Poměrně nepříjemným překvapením bylo zjištění, že tato konkrétní instance Moodle nevynucuje žádná omezení na sílu hesel, a tak bylo možné si nastavit heslo pouze o jednom znaku. Ačkoliv vynucování síly hesel není v rozsahu zodpovědností mobilní aplikace či přidružené webové služby, informoval jsem o této skutečnosti administrátora IT Tichého světa, aby zvážil případná opatření.

Proti poslední možnosti – tedy opakovanému testování různých hesel – se lze bránit zamčením účtu na určitou dobu po několika neúspěšných pokusech, nebo umělým zpomalením vyhodnocení požadavků na ověření uživatele.

Autentizační schémata

Protokol HTTP sám o sobě nedefinuje, jakým způsobem autentizovat uživatele vůči serveru. Proto vznikla takzvaná autentizační schémata, která standardizují proces autentizace pro bezpečné použití v aplikacích. Autentizačních schémat existuje celá řada, já se zde věnuji pouze dvěma nejdůležitějším.

HTTP Basic Authentication Základním autentizačním schématem stavějícím na protokolu HTTP je schéma *HTTP Basic Authentication.* Stanovuje poměrně jednoduchý formát, v jakém jsou odeslány přihlašovací údaje na server. Formát popisuje následující předpis:

$$base64(\text{"uzivatel:heslo"})$$

kde $base64$ je algoritmus kódování binárních dat do ASCII textu (HTTP je textový protokol).

K výsledné hodnotě je připojen prefix „Basic “, aby bylo možné jednoznačně poznat, o jaké autentizační schéma jde. Celý tento výsledný řetězec je pak vložen do HTTP hlavičky *Authorization.* [16]

Jakmile server dostane takový HTTP požadavek, při jeho vyhodnocení musí dekódovat obsah hlavičky a zkontrolovat, zda jsou dané přihlašovací údaje správné.

Už název napovídá, že jde o velmi jednoduché autentizační schéma, které je díky tomu velmi snadné na implementaci. Nevýhodou však je, že klient si musí po celou dobu interakce se serverem pamatovat přihlašovací údaje a s každým HTTP dotazem je přenášet po síti. Ačkoliv implementace ve webových prohlížečích jsou schopny si přihlašovací údaje pamatovat, mobilní aplikace by toto musela řešit sama.

Bearer Authentication Autentizační schéma *Bearer Authentication* je definované v rámci standardu OAuth 2 [17], ale jeho základní princip je použitelný i odděleně. OAuth 2 standardizuje autorizaci aplikací pro přístup k uživatelským datům obsaženým v jiné aplikaci. Moodle poskytuje možnost fungovat jako klientské zařízení a pomocí OAuth 2 tak získat uživatelská data například od společností Facebook, Google a dalších. Použití systému Moodle v roli poskytovatele uživatelských dat ale není možné.

Schéma typu *Bearer Authentication* řeší nedostatky schématu *HTTP Basic Authentication* tím, že zavádí takzvaný *Bearer token.* Ten zde plní funkci „nositele tajemství“, kterým je v případě *HTTP Basic Authentication* řetězec obsahující kombinaci přihlašovacího jména a hesla. Díky použití tokenu tak odpadá nutnost odesílat uživatelské jméno a heslo s každým HTTP dotazem. Zároveň si klientská strana nemusí přihlašovací údaje ukládat, stačí jí mít uložený token.

Spolu se samotným tokenem bývají spojena nějaká data. Typicky je totiž potřeba znát minimálně vlastníka tokenu – uživatele, který se s daným tokenem snaží autentizovat. Kromě vlastníka je dobré omezit životnost tokenu stanovením data expirace, aby se snížila pravděpodobnost případného úniku validního tokenu do nepovolaných rukou. [17] V případě úniku je další výhodou, že útočník z tokenu nezíská kombinaci přihlašovacího jména a hesla, která se může u lehkomyslných uživatelů vyskytovat napříč různými službami. Obecně platí, že životnost tokenu by neměla být větší, než je nutné.

V kombinaci s tokeny s krátkou životností se používají takzvané *refresh* tokeny, které mají životnost naopak větší. S jejich pomocí je možné na speciálním API endpointu zažádat o vystavení nového tokenu s krátkou životností. Vzhledem k dlouhé životnosti *refresh* tokenu (například jeden měsíc) je dobré se soustředit na jeho ochranu před únikem. [18] Jeho případný únik je však stále o něco méně závažnějším incidentem než únik samotného hesla.

Tokeny lze obecně ještě dělit podle toho, kde jsou uložena data, se kterými se pojí. Neprůhledný (*opaque*) token je řetězec unikátní pro danou oblast po-

užití a bývá uložen na serveru spolu s ním spojenými informacemi. Samotný řetězec v sobě nenese žádné informace a slouží pouze jako jakýsi identifikátor k datům uloženým na serveru. [19]

Json Web Token

Kromě „neprůhledných“ tokenů existuje ještě druhý typ tokenů, které naopak všechny důležité informace nesou v sobě. Typickým představitelem takového tokenu je Json Web Token, zkráceně JWT. Skládá se ze tří částí – hlavičky, payloadu a podpisu. [20]

V hlavičce jsou obsaženy základní informace o tokenu ve formátu JSON jako je typ tokenu („JWT“) a použitý algoritmus pro vytvoření podpisu. Druhou částí JWT je payload, který je rovněž ve formátu JSON a obsahuje informace, které mají být v tokenu uchovány. Typicky je to opět uživatel, datum expirace nebo například uživatelské role. Aby zůstal token co nejkratší, v JSON struktuře se často využívá co nejkratších názvů atributů. Payload tokenu může vypadat například takto:

```
{
  "sub": "12345",       = subject (napr. uzivatelske id)
  "name": "John Doe",
  "exp": 1516239022     = expiration
}
```

Poslední částí Json Web Tokenu je podpis, který zaručuje, že token nebyl vystaven někým cizím a ani nebyl upraven. Podpis je vytvořen zřetězením hlavičky a payloadu, jejich zakódováním pomocí *base64* a aplikováním vybraného podpisového algoritmu (např. HMAC SHA256).

Json Web Token může a nemusí být dále zašifrován. [20] Záleží na tom, zda už například není komunikace šifrována sama o sobě.

Ze skutečnosti, že samotný token je nositelem informací a že server si nemusí udržovat stav o přihlášení uživatelů, plynou zajímavé důsledky. Server může využít informací obsažených v tokenu k tomu, aby vůbec nemusel číst data o uživateli z databáze. To může mít pozitivní vliv na výkon.

Uložení informací v tokenu se navíc velmi hodí pro prostředí, kde uživatel může být obsloužen více než jedním serverem. Kdyby se totiž jednalo o „neprůhledný“ token, musela by existovat sdílená databáze obsahující informace o přihlášeních, ke které by musely mít všechny serverové instance přístup. S použitím JWT tokenu je server schopen uživatele autentizovat bez přítomnosti databáze. Pro ověření údajů v tokenu stačí totiž jen ověřit podpis. Server tak nemusí udržovat stav uživatelského přihlášení, což je obzvlášť vítaná vlastnost v prostředí cloudu.

1.6.4 Automatizace procesů

Jedním z hlavních úkolů této práce je implementace automatického sestavení aplikace a jeho distribuce do obchodů s aplikacemi. Proto zde v následujících odstavcích shrnuji přístupy běžně používané v softwarových projektech pro dosažení značné míry automatizace.

Continuous Integration

Continuous Integration je postup při vývoji softwaru, který je aplikován po implementaci nové funkcionality před jejím zakomponováním do softwarového projektu jako celku. Proces integrace nových změn však nemusí být jednoduchý, a tak je v rámci moderních přístupů prosazováno Continuous Integration – tedy frekventované (kontinuální) integrování malých změn pro co nejrychlejší odhalení případných problémů a snížení rizika zanesení chyb do projektu. [21] Časté integrování nových změn nejen snižuje rizika zanesení chyb, ale i zjednodušuje proces mergování (*merge*) Git větví díky minimalizaci pravděpodobnosti *merge* konfliktů. [22]

Continuous Integration představuje skvělou příležitost pro automatizaci s využitím testů a automatického sestavení softwaru. Prerekvizitou Continuous Integration tak je kvalitní pokrytí kódu testy. Typicky je jejich spouštění navázáno na události v systému na správu verzí.

Pro efektivní práci vývojářů by mělo být neopomíjenou záležitostí optimalizace rychlosti běhu Continuous Integration skriptů, aby měl vývojář rychlou zpětnou vazbu o výsledku integrace jeho změn do celku. [21]

Continuous Delivery

Continuous Delivery je postaveno o úroveň výše nad Continuous Integration. Jeho cílem je udržovat software stále ve stavu připraveném pro nasazení do produkce a přinést tak možnost zvýšení frekvence vydání. [23] Závisí na zavedeném Continuous Integration, aby bylo možné zajistit dobrý stav celého softwarového projektu v daný moment. Continuous Delivery přináší softwarovým týmům možnost rychle reagovat na požadavky klienta na nasazení požadované verze softwaru.

Continuous Deployment

Continuous Delivery však nemusí nutně znamenat automaticky vysoce frekventované vydávání nových verzí. Taková praxe se skrývá až pod pojmem Continuous Deployment. Zde může být software nasazován na produkci prakticky při každém commitu do hlavní větve. [23]

Výhodami vyšší frekvence vydání je snížení rizika zanesení chyb na produkci díky menším balíkům změn. V případě, že k zanesení chyby přesto dojde, je její odhalení v kódu snadnější. Zároveň je Continuous Deployment lákavé

z byznysového hlediska, protože klient může častěji dostávat nové změny a nemusí tak dlouze čekat na velké balíky změn. Díky tomu jasně vidí postup v práci na softwarovém projektu. Continuous Deployment zároveň pomáhá získat rychleji zpětnou vazbu od koncových uživatelů, díky čemuž umožňuje rychleji detekovat problematické směřování produktu.

Kapitola 2

Návrh řešení

Na základě předešlé analýzy se v této kapitole snažím sestavit návrh řešení, jak konkrétně realizovat komunikaci mezi webovou službou a mobilní aplikací včetně autentizace a autorizace uživatelů. Návrh se nejdříve věnuje samotné webové službě a odůvodňuje důležitá rozhodnutí týkající se její implementace. V podkapitole, která se věnuje mobilní aplikaci, jsou navrženy změny, které je nutné provést v prototypu aplikace, aby získal veškerou funkcionalitu potřebnou pro nasazení do produkčního prostředí. V druhé polovině této kapitoly rozebírám možné přístupy pro správu Git repozitáře, který během implementace poslouží jako základ pro automatizaci sestavení a nasazení mobilní aplikace.

2.1 Webová služba

Pro poskytování dat mobilní aplikaci z webového e-learningového portálu postaveném na systému Moodle je potřeba zřídit nějaké webové API. Pro jeho tvorbu jsem se rozhodl jít cestou implementace separátní aplikace, kterou bude možné nasadit na webový hosting spolu s již existující instancí Moodle.

Cesta separátní aplikace přináší nutnost vlastní práce s daty v databázi systému Moodle, což je jistě riziková oblast, kterou bude třeba řešit se značnou opatrností. Dále bude nutné implementovat logiku pro autentizaci a autorizaci uživatelů. Ve výsledku bude tedy potřeba podrobně pracovat s dokumentací Moodle, aby práce s daty a s ověřením uživatelů fungovala stejně jako v samotném Moodle.

Vlastní implementace webového API místo použití existujícího Moodle Web services API ale přinese mnohem větší kontrolu nad tím, co se ve webové aplikaci děje. Díky tomu budu mít výkon webového API více pod kontrolou, protože tímto přístupem získám prostor dělat případné optimalizace.

Vlastní implementace navíc přináší mnohem větší flexibilitu ohledně transformací dat tak, aby je mobilní aplikace získala přesně v té formě, kterou

vyžaduje. Z analýzy vyplývá, že bude potřeba provádět transformace jako například extrakci URL adresy videa z HTML řetězce. Provádění transformací dat v mobilní aplikaci se chci pokud možno vyhnout, protože nějaké složitější nebo časté operace by mohly znamenat pokles výkonu aplikace a tím i snížení kvality uživatelské zkušenosti.

Vlastní implementace tedy za cenu vyšší pracnosti přináší možnost přizpůsobit si API na míru potřebám mobilní aplikace. To považuji za zásadní výhodu oproti Moodle Web services API, protože to zásadně zjednoduší kód mobilní aplikace, která by měla kvůli omezeným hardwarovým prostředkům provádět ideálně co nejméně výpočetní logiky.

2.1.1 Volba frameworku

Pro implementaci webové služby volím framework Symfony, který plní nefunkční požadavky specifikované v kapitole 1.5 plynoucí z používaných technologií webového hostingu Tichého jazyka.

Ačkoliv existují PHP frameworky, které jsou od základu mnohem minimalističtější co se týče závislostí jako například Slim framework[6], Symfony poskytuje pokročilé funkce, kvalitní dokumentaci a dobrou komunitní podporu. Příkladem je existence komunitního balíčku pro implementaci JWT autentizaci a *refresh* tokeny. Dalšími výhodami Symfony je úzká integrace s ORM knihovnou Doctrine, podpora *dependency injection* nebo široká možnost konfigurace zabezpečení.

Zdrojem dat pro webové API bude relační SQL databáze, ze které čerpá e-learningový portál postavený na Moodle. Vzhledem k omezenému rozsahu mobilní aplikace bude potřeba pracovat pouze s omezeným počtem tabulek v této databázi. Pro snadnější práci s daty v Symfony aplikaci budou tabulky namapovány do entitních tříd pomocí Doctrine ORM.

2.1.2 Autentizace a autorizace

Webová aplikace musí komunikovat pomocí protokolu HTTPS. Současný e-learningový portál Tichého jazyka však běží pouze na HTTP, a tak požádám IT administrátora Tichého světa o přechod na HTTPS. Tím dojde k získání šifrované komunikace a ověření identity serveru. Mobilní aplikace tak bude mít jistotu, že při přihlašování odesílá uživatelské přihlašovací údaje opravdu na server Tichého jazyka a ne kamkoliv jinam.

Autentizace uživatele bude probíhat ověřením kombinace přihlašovacího jména a hesla. Po jejich ověření dojde na straně serveru k vytvoření JWT tokenu ve spojení s autentizačním schématem *Bearer Authentication.* Token se bude používat pro ověření identity uživatele v následné komunikaci. Použitím *Bearer Authentication* schématu místo *HTTP Basic* se navíc vyhnu nutnosti

[6] `https://www.slimframework.com/`

ukládat přihlašovací údaje v mobilní aplikaci a jejich přenášení po síti při každém HTTP dotazu, což sníží riziko jejich vyzrazení.

Místo přihlašovacích údajů si bude mobilní aplikace držet přidělený JWT token. Nastavením krátké trvanlivosti tohoto tokenu budou sníženy teoretické následky v případě jeho úniku. Aby mohl být JWT token po jeho vypršení obnoven, webová služba bude mobilní aplikaci generovat ještě *refresh* token. Ten bude muset být také bezpečně uložen v zařízení. Pro vyšší bezpečnost bude *refresh* token navíc pouze jednorázový. To znamená, že po jeho použití server vygeneruje novou dvojici JWT tokenu a *refresh* tokenu a starý *refresh* token bude zneplatněn.

Na základě veřejně dostupných doporučení a s přihlédnutím k povaze kurzů Tichého jazyka navrhuji stanovit trvanlivost *refresh* tokenu na třicet dní. V kombinaci s jeho jednorázovou vlastností to bude v praxi znamenat, že pokud uživatel během třiceti dní nepoužije svůj *refresh* token (tedy že třicet dní prakticky nepoužije mobilní aplikaci Tichého jazyka), aplikace ztratí k e-learningu přístup a uživatel bude muset příště pro získání dvojice tokenů znovu zadávat své uživatelské jméno a heslo.

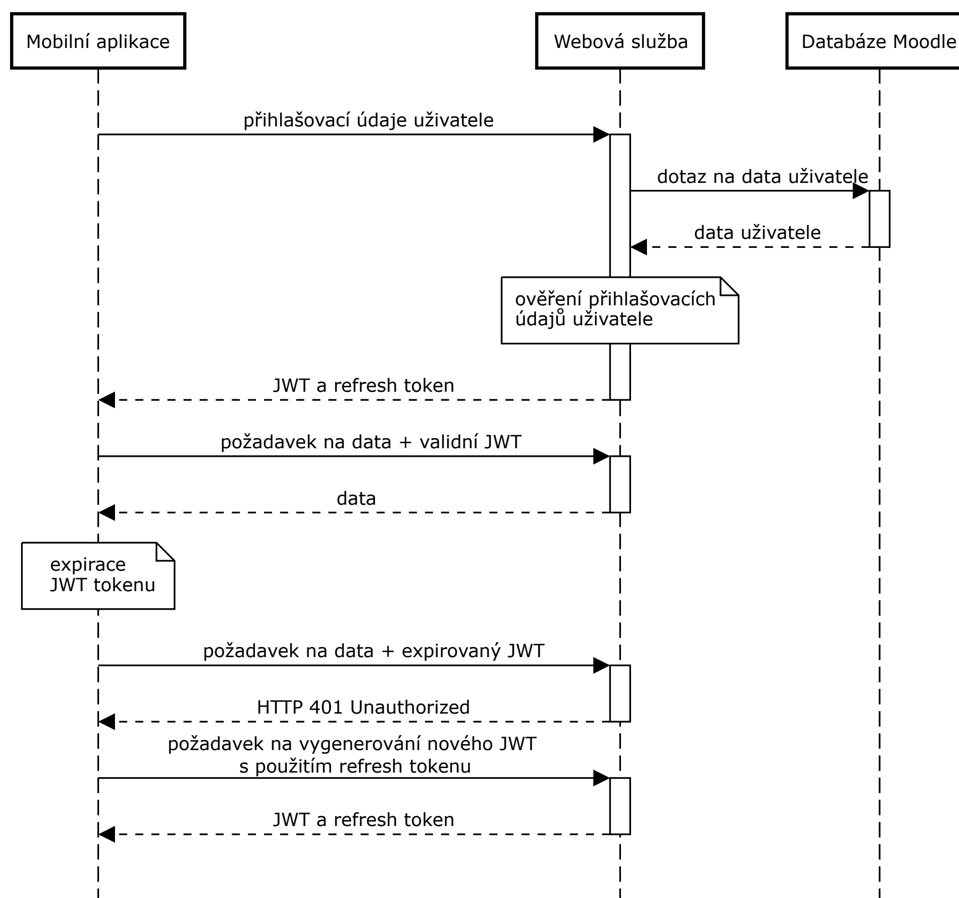
Webová aplikace tak bude pro účely autentizace implementovat dva API endpointy. První endpoint bude zprostředkovávat výměnu validních přihlašovacích údajů za dvojici JWT tokenu a *refresh* tokenu. V případě nesprávné kombinace přihlašovacích údajů vrátí standardní HTTP kód 401. Druhý endpoint bude sloužit k obnovení JWT tokenu za použití *refresh* tokenu. Ve výsledku tak bude opět zprostředkovávat výměnu – *refresh* token za novou dvojici tokenů typu JWT a *refresh*. Práce s JWT a refresh tokenem je nastíněna na sekvenčním diagramu 2.1

Autorizace uživatelů, tedy ověření přístupu uživatelů ke konkrétním výukovým materiálům musí probíhat na úrovni aplikační logiky. Webové API bude čerpat informace o tom, do kterých výukových modulů má uživatel přístup, z databáze systému Moodle. Využívat bude tabulku `mdl_user_enrolments`, která je vazební tabulkou mezi tabulkami s uživateli (`mdl_user`) a zápisy do kurzů (`mdl_enrol`).

2.1.3 REST API

Jak již bylo řečeno dříve, API by mělo být navrženo tak, aby mohla mobilní aplikace provádět co nejméně datových transformací. S touto myšlenkou bylo API navrženo. API je zcela jednoznačně orientováno na data, a půjde tedy o klasické RESTful API s daty ve formátu JSON, který zvládne aplikace napsaná v jazyce Dart snadno zpracovat.

Při výběru architektury REST jsem ještě uvažoval nad použitím dotazovacího jazyka GraphQL, který nabírá v posledních letech na popularitě. [24] GraphQL umožňuje klientským aplikacím, aby si samy definovaly strukturu dat, kterou chtějí ze serveru získat. Jazyk umožňuje definovat dotazy, které mohou po vyhodnocení obsahovat v jedné HTTP odpovědi data týkající se



Obrázek 2.1: Sekvenční diagram znázorňující interakci mobilní aplikace a webové služby během autentizace

více typů entit. Pro serverové aplikace existují knihovny, které dokážou jazyk GraphQL zpracovat a vyhodnotit s použitím zdroje dat, kterým by v tomto případě byla databáze.

Pro účely mobilní aplikace Tichého Jazyka ale použití GraphQL nemá příliš velké využití, protože všechny funkční požadavky webového API lze pohodlně splnit s použitím několika REST API endpointů. Jejich vytvoření je jednodušší na implementaci a dává programátorovi větší kontrolu nad prací s databází.

Na základě funkčních požadavků na webovou službu definovaných v kapitole 1.5 navrhuji v tabulce 2.1 implementaci API endpointů, které pokrývají jednotlivé požadavky.

Účel endpointu	Cesta v URL	Pokryté funkční požadavky
Seznam modulů	`/api/modules`	FP1
Seznamu lekcí daného modulu	`/api/modules/{module-id}/lectures`	FP2
Seznam videí dané lekce	`/api/lectures/{lecture-id}/chapters`	FP3
Seznam slovníků	`/api/glossaries`	FP4
Seznam videí ve slovníku	`/api/glossaries/{glossary-id}/entries`	FP4
Vyhledávání videí	`/api/chapters/search`	FP5, FP6
Ověření přihlašovacích údajů	`/api/login-check`	FP7

Tabulka 2.1: Návrh REST API endpointů

Kvůli probíhajícímu rozhodování Tichého světa o možnosti uložení videí na vlastním serveru místo na YouTube zatím v návrhu API nepočítám s endpointem pro stažení a přehrání videí (funkční požadavek FP8). Prozatím bude používáno přehrávání videí z YouTube a jakmile dojde k rozhodnutí o umístění videí, dojde k přidání požadovaného API endpointu.

U endpointu `/api/glossaries` stojí za povšimnutí, že může vracet více slovníků. V databázi Tichého jazyka se však vyskytuje pouze jeden *Videoslovník*. Tvůrcům materiálů ale nic nebrání ve vytvoření případného dalšího slovníku. Proto raději počítám s možností více slovníků, aby nenastala situace, že by se případný nově vytvořený slovník v aplikaci nezobrazoval.

V prototypu mobilní aplikace je vyhledávání řešeno načtením veškerých dat o videích do paměti a jejich řazením na základě míry shody s vyhledávacím dotazem. Takové řešení je samozřejmě výhodné ve chvílích, kdy je zařízení offline, protože nevyžaduje žádnou komunikaci se serverem. Načtení všech dat do paměti je však paměťově i výkonově náročné. Proto jsem navrhl vytvořit endpoint, jehož úkolem bude vyhledávání napříč všemi videi nebo videi pouze vybraného modulu. Vyhledávání ve stažených videích je samozřejmě možné realizovat s pomocí lokální databáze v mobilní aplikaci.

Úkolem takto navržených API endpointů bude často transformace dat pro potřeby mobilní aplikace. Na tyto transformace se lze dívat dvěma pohledy.

Prvním typem bude transformace samotných hodnot. Půjde zejména o extrakci dat z HTML řetězců, které tvoří hlavní obsah stránek s videi v portálu Moodle. Ačkoliv pro framework Flutter existují balíčky pro zobrazení HTML přímo v mobilní aplikaci, pro zachování přehledného uživatelského rozhraní je lepší zachovat jeho jednotný vizuální styl. Proto by mělo webové API při-

pravit data v takové podobě, která by byla připravena na zobrazení v pevně daném uživatelském prostředí aplikace.

Druhým typem bude transformace dat ze struktury, v jaké jsou uloženy v databázi systému Moodle, do struktury, kterou může mobilní aplikace snadno konzumovat. V rámci implementace tak bude potřeba provádět mapování struktury Moodle tabulek na strukturu výukových materiálů, kterou mobilní aplikace očekává.

Navigační struktura aplikace počítá se třemi typy entit – s moduly, lekcemi a videi. Databáze Moodle ale používá entity čtyři – *Course*, *Course Section*, *Book* a *Book Chapter*. Z povahy dat je jasné, že moduly musí být namapovány na entitu *Course* a videa budou představována entitami *Book Chapter*. Na základě používání webového e-learning portálu musí být lekce představována entitou *Course Section*. Z toho plyne, že v implementaci webové služby bude nutné sloučit videa z více entit typu *Book* do jedné lekce.

2.2 Mobilní aplikace

Aplikace byla psána během tvorby bakalářské práce Alžbety Gogolákové v první polovině roku 2019. Kvůli tomu, že ekosystém Flutteru je stále poměrně nový a ne příliš ustálený, kód aplikace závisí na řadě zastaralých balíčků. Některé z nich například nebyly v tehdejší době dostupné ve stabilní verzi, jiným naopak byla ukončena podporu a vývoj. Proto bude potřeba takové balíčky aktualizovat či nahradit tak, aby byla požadovaná funkcionalita zachována a aby aplikace využívala aktuální, udržovaný a stabilní kód.

Co se týče úprav na úrovni kódu mobilní aplikace, samozřejmostí bude korektní implementace autentizace s použitím JWT a *refresh* tokenů, jak bylo navrženo v předchozí kapitole 2.1.2 Autentizace a autorizace.

2.2.1 Offline mód

Důležitou vlastností aplikace má být schopnost fungovat i bez připojení k internetu. Vlastnost offline módu již do určité míry prototyp aplikace implementuje. Kvůli tomu, že prototyp ale počítá s načtením kompletních dat se studijními materiály při startu aplikace, bude potřeba provést u offline módu úpravy.

Aby byla aplikace schopna pracovat v offline módu i po změnách souvisejících s návrhem webového API, navrhuji, aby mobilní aplikace načtená data se strukturou materiálů ukládala do lokální databáze vždy, když dojde k jejich načtení ze serveru. Díky tomu bude uživatel moci prohlížet dříve navštívené výukové materiály i bez přístupu k internetu.

Automatické ukládání dat do lokální databáze by se však nemělo týkat samotných videí. Ta by měla být stažena až na základě uživatelova podnětu, protože jejich velikost není zanedbatelná. I přesto, že videa nebudou automaticky stahována, považuji toto řešení za lepší, než kdyby byly materiály

stahovány pouze na základě uživatelovy explicitní žádosti. Z mé osobní zkušenosti vždycky raději vítám, když aplikace poskytuje v offline módu alespoň nějaká data místo dat žádných.

Prototyp aplikace zjišťuje absenci připojení k internetu detekcí chyb při komunikaci se serverem. Reaguje na ně zobrazením jednoduché obrazovky informující o nepřítomnosti připojení k internetu s tlačítkem „Zkusit znovu“, které se na uživatelův příkaz pokusí znovu připojit k internetu.

Toto řešení bych chtěl změnit tak, aby hláška o absenci připojení nezabírala celou obrazovku telefonu a raději byla uživateli zobrazena v podobě malé lišty na spodním okraji displeje. Absenci tlačítka navrhuji vyřešit tak, že aplikace bude sama opětovné připojení k internetu kontrolovat.

Díky offline úložišti bude aplikace pracovat se dvěma zdroji dat (server a paměť telefonu). Rozhodnutí o tom, který ze zdrojů v dané chvíli použít, bude muset mobilní aplikace učinit právě na základě stavu připojení.

Žádoucím chováním zde je, aby aplikace používala vždy nejlepší dostupný zdroj dat. V praxi by to mělo znamenat, že pokud se aplikace nachází v offline módu (a zobrazuje tedy offline uložená data) a dojde k připojení k internetu, aplikace by měla automaticky tuto změnu rozpoznat a přepnout se na data dostupná ze serveru. Dva zdroje dat s sebou přinášejí i komplikaci, kterou je nutnost provádět konsolidace dat v případě jejich nesouladu mezi zdroji. Nesoulad může nastat typicky po úpravě dat na serveru.

2.2.2 Vyhledávání

Úpravami musí projít i vyhledávání. Vyhledávání čistě na straně mobilní aplikace, jak je připraveno v jejím prototypu, není optimálním řešením kvůli velkému množství videí. Proto bude implementováno na straně serveru.

Vzhledem k tomu, že aplikace má však umožňovat kvalitní použití i v offline módu, je vysoce očekávatelné počítat s případem užití, kdy uživatel bude chtít vyhledávat napříč offline uloženými daty. Navrhuji tedy řešení, aby se aplikace na základě stavu připojení vždy rozhodla, zda vyhledávání realizovat s použitím serveru, nebo zda k němu využít zabudovaný jednoduchý vyhledávač.

2.3 Strategie větví v nástroji Git

Aby bylo možné dobře splnit jeden z hlavních úkolů této práce – tedy automatizovat proces sestavení mobilní aplikace a její nasazení do obchodů s aplikacemi – je potřeba vhodně navrhnout práci s verzovacím nástrojem Git a jeho větvemi. Existuje několik často doporučovaných strategií.

2.3.1 GitFlow

GitFlow je strategie správy větví v nástroji Git, která vznikla v roce 2010 na základě příspěvku na blogu Vincenta Driessena. [25] GitFlow používá hlavní dvě větve – *master* a *develop*. Aktivní vývoj probíhá v *develop* větvi. Z ní mohou vycházet takzvané *feature* větve, které představují vývoj nějaké funkcionality. Po jejich dokončení jsou sloučeny (*merge*) zpátky do větve *develop*. Oproti tomu na větev *master* je nahlíženo jako na větev, kde každý commit představuje nové vydání (*release*). Pro mergování větví by měl být využíván přepínač `--no-fast-forward`, který vynutí pokaždé vytvoření *merge* commitu a graf commitů tak jasněji popíše vývoj nových funkcí. [25]

V momentě, kdy je v *develop* větvi stav kódu připraven na vydání, je vytvořena *release* větev. Ač tomu může její jméno napovídat, commity v této větvi přímo nepředstavují stav kódu, který je součástí konkrétního vydání. Místo toho *release* větev poskytuje prostor pro provedení posledních změn jako menších oprav chyb před finálním zamergováním do *master* větve. Finální stav vydání představuje až zmíněná *master* větev. GitFlow používá pro zaznamenání konkrétních verzí v *master* větvi Git tagy.

GitFlow myslí i na možnost vytváření hotfixů, tedy změn, které je potřeba provést okamžitě na produkční verzi. Pro takový účel zavádí *hotfix* větve, které vychází z *master* větve. Jakmile je hotfix hotov, příslušná *hotfix* větev musí být zamergována do obou větví *master* a *develop*.

Díky jasnému oddělení větví *master* a *develop* umožňuje GitFlow striktnější schvalování nových vydání, a proto se hodí pro použití ve větších organizacích nebo například v open-source prostředí, kde může přispívat změnami do kódu prakticky kdokoliv. [26]

2.3.2 Trunk-Based Development

Mírně odlišný pohled na strategii větvení Git repozitáře přináší přístup Trunk-Based Development. Ten, narozdíl od GitFlow, nepracuje s dvěma hlavními větvemi *master* a *develop*, ale používá jednu takzvanou *trunk* – kmenovou větev. Tou je standardně právě větev s názvem *master*. Trunk-Based Development říká, že celý vývoj by se měl točit právě okolo pouze této jedné *trunk* větve. Ta by měla obsahovat commity, které jsou funkční, procházejí testy, a jsou dostatečně aktuální. [27] Do jisté míry tak *trunk* větev plní funkci větve *develop* z GitFlow.

Trunk-Based Development také používá *feature*, *bugfix* a *hotfix* větve. Jejich funkce je velmi podobná jako v GitFlow. Odlišným stylem jsou ale řešena vydání. Pro ně se v rámci Trunk-Based Development používají *release* větve, které vycházejí z *master* větve a poskytují prostor pouze pro opravy chyb. Jejich úkolem je stabilizovat produkt do co nejlépe fungující podoby.

Pro použití v kombinaci s Continuous Delivery jsou ale *release* větve nepotřebné a pro frekventovaná vydání nových verzí je doporučováno použití

přímo *trunk* větve. [28] *Release* větve jsou vhodné pro projekty, u kterých vývojový tým z nějakého důvodu nedokáže udržet *master* větev v perfektně „zdravém“ stavu.

Maturity větev

Méně známým typem větví je takzvaná *maturity* větev, jejímž úkolem je dlouhodobě držet commity, které mají určitou *maturity* úroveň. *Maturity* úroveň udává určitou kvalitu kódu a s ní spojené prostředí, do kterého je možné daný commit nasadit.

Příkladem *maturity* větve může být větev s názvem *production*, která si drží přesný stav kódu v produkčním prostředí v daném časovém bodě. Díky tomu dokáže tato *maturity* větev skvěle mapovat postupný vývoj commitů, které se dostaly do daného prostředí (například *production*, *staging* apod.). *Maturity* větev na sebe navíc umožňuje snadno navázat automatizované procesy jako například nasazení produktu do příslušného prostředí.

Release-ready master větev

Tento přístup, který vyžaduje, aby veškeré commity v *master* větvi byly připraveny na vydání, je z hlediska strategie větví velmi jednoduchý. Nevyžaduje totiž použití *release* ani *maturity* větví, což samozřejmě zjednodušuje celý graf commitů. Není však snadné tento přístup adoptovat. Vyžaduje totiž opravdu kvalitní sadu testů, která je schopna zaručit kvalitu commitů v *master* větvi. Zároveň vyžaduje disciplinovaný tým, který si je vědom toho, že každý *merge* do *master* větve může být aplikován na produkční prostředí. [22] *Release-ready master* větev umožňuje snadné aplikování přístupu Continuous Deployment.

Trunk-Based Development klade důraz na jednoduchost, rychlost mergování *feature* větví do *master* větve a minimalizaci rizika velkých *merge* konfliktů. [22] Oproti GitFlow přináší jednodušší graf commitů a lepší připravenost na Continuous Integration a Delivery. [29] Tím pádem se hodí spíše pro menší projekty s důrazem na rychlý vývoj.

2.3.3 Volba strategie

Automatizace vývojového procesu je základním cílem této práce, proto navrhuji, aby bylo Continuous Integration základním kamenem celého procesu vývoje. Graf commitů v nástroji Git by měl být silně inspirován přístupem Trunk-Based Development organizovaného okolo hlavní *master* větve s použitím feature větví pro vývoj jednotlivých funkcí. Větší striktnost GitFlow se hodí spíše pro větší projekty s více vývojáři, a proto by zde použití této strategie působilo spíše komplikace a zpomalení práce s nástrojem Git.

V rámci Continuous Integration tedy navrhuji spouštět testy a sestavení aplikace pro každý commit. Pro zaručení kvality *master* větve je vhodné provést dodatečné konfigurace v systému pro správu Git repozitáře, aby bylo

možné do *master* větve mergovat pouze dobře otestovaný kód (např. označením větve jako *protected*). Jedním ze základních měřítek otestovanosti kódu by mělo být například pokrytí kódu, které by bylo dobré měřit.

Pro účely tohoto projektu však není dle mého názoru tolik vhodné použití *release-ready master* větve. Důvodem je, že mobilní aplikace bude minimálně zpočátku podstupovat větší množství změn, a tak není nutné se zpočátku soustředit na stav kódu připravený na nasazení. Takového stavu je možné dosáhnout později.

Dalším důvodem pro nepoužití *release-ready master* větve ve spojení s Continuous Deployment je fakt, že takový přístup by generoval velké množství vydání. Vysoká frekvence vydání je samozřejmě v mnoha případech žádaná věc, avšak v tomto případě každé vydání znamená nutnost aplikaci pokaždé nechat před publikováním schválit, což v případě Apple AppStore běžně trvá 1–2 dny. Proto budu mít za cíl držet se praktiky Continuous Delivery, nikoliv však Continuous Deployment, která zde nepřináší žádné benefity, spíše naopak.

Místo *release-ready master* větve navrhuji raději použít přístup *maturity* větví, který přinese lepší kontrolu nad tím, do jakého prostředí je aplikace v daný moment nasazena. To umožní i jasně definovat různé automatizované skripty pro nasazení aplikace do různých prostředí. Ačkoliv nebude *release-ready* stav v *master* větvi vyžadován, dává smysl se o něj i přesto snažit, protože kvalitní kód v *master* větvi může znamenat snadnější integraci nových funkcí.

2.4 Continuous Integration a Delivery

Na výše definované strategii větví v Gitu bude možné stavět skripty pro Continuous Integration a Delivery. Zásadním nástrojem, který mi v tom pomůže, bude nástroj Fastlane. Fastlane je nástroj pro automatizaci úkonů se zaměřením na mobilních aplikace pro iOS a Android. Umožňuje sestavení aplikace, její podepsání a dokáže i komunikovat s API obchodů s aplikacemi Google Play a AppStore a nahrát tak na ně sestavenou aplikaci včetně potřebných metadat pro její publikování. Pro svůj běh vyžaduje běhové prostředí pro jazyk Ruby, ve kterém je nástroj napsán.

Hlavním konfiguračním souborem je `Fastfile`, který obsahuje sadu procedur, které Fastlane nazývá termínem *lanes*. Každá *lane* je posloupností příkazů, které Fastlane v daném pořadí vykoná. Tvůrce dané *lane* má na výběr z bohaté nabídky příkazů, které jsou součástí Fastlane. Jejich kompletní seznam je dostupný v dokumentaci Fastlane[7]. Je ale možné využít i příkazů třetích stran, které jsou dostupné ve formě pluginů.

Díky široké paletě příkazů dokáže Fastlane automatizovat celý proces sestavení aplikace a její nahrání do obchodu s aplikacemi. Když je celý tento

[7] `https://docs.fastlane.tools/actions/`

proces vykonáván manuálně, dokáže být kvůli nezanedbatelnému počtu kroků poměrně časově náročný. Fastlane je proto nástrojem, který má potenciál vývojářským týmům ušetřit nemalé množství času a peněz. Jeho použití v CI/CD procesech se přímo nabízí, a tak jej v nich použiji i v rámci této práce.

Obchody s aplikacemi nabízejí několik módů vydání aplikace jako například uzavřené alfa testování, otevřené beta testování a podobně. Z tohoto důvodu, že aplikace může být sestavena a distribuována v řadě různých prostředí, vznikne několik Fastlane procedur pro každé z těchto prostředí. U každé platformy vznikne minimálně jedno prostředí pro uzavřené testování a dále samozřejmě produkční prostředí. Každé takové prostředí dostane svou vlastní *maturity* větev, která bude sledovat jejich vývoj. Názvy *maturity* větví by měly být jasně odlišitelné od ostatních větví, a proto pro ně budu používat názvy ve tvaru „`{platforma}-{prostredi}`“.

Kapitola 3

Implementace webové služby

3.1 Zprovoznění vývojového prostředí

Před začátkem implementace webové služby postavené na frameworku Symfony bylo potřeba zprovoznit vývojové prostředí, ve kterém aplikace poběží, a poskytnout jí všechny závislosti, které vyžaduje. Prostředí pro běh webového API se skládá z následujících komponent:

- databáze MariaDB,
- webový server s PHP,
- systém Moodle.

Vývojové prostředí a stejně tak i samotné webové API postavené na Symfony jsem začal vyvíjet v rámci repozitáře `tichy-jazyk-api` na fakultní instanci GitLab[8].

3.1.1 Docker

Aby bylo možné rozběhnout identické vývojové prostředí i na jiném počítači, rozhodl jsem se použít nástroj Docker. Docker je kontejnerizační engine, který umožňuje spouštět procesy v předem vytvořeném a odděleném prostředí. Takové prostředí označuje pojmem „kontejner“. Díky předem vytvořenému prostředí zaručuje, že kontejner spuštěný na dvou různých počítačích poběží prakticky stejně. [30] Oddělenost kontejnerů přináší řadu bezpečnostních benefitů, vývojářům umožňuje mimo jiné například snadno spouštět různé verze dané aplikace. [30] Docker je primárně ovládán pomocí příkazové řádky. Spuštění aplikací, pro které již existuje vytvořený kontejner, se tak může zjednodušit na spuštění jediného příkazu pomocí Docker CLI.

[8] `https://gitlab.fit.cvut.cz/zakjindr/tichy-jazyk-api`

Předlohou pro běžící kontejnery jsou takzvané obrazy (*images*). Obraz se skládá ze souborového systému, ve kterém jsou spustitelné soubory a knihovny, které kontejner vyžaduje k běhu. [30] Tvorba obrazů se běžně realizuje pomocí souboru `Dockerfile`, který obsahuje posloupnost instrukcí, které se pro sestavení obrazu mají vykonat. V souboru `Dockerfile` lze použít sadu instrukcí jako například instrukci `RUN` pro spouštění příkazů, `COPY` pro vložení souborů do souborového systému obrazu, `ENV` pro nastavení proměnných prostředí a další.

Souborové systémy Docker obrazů se skládají z vrstev. Jedna vrstva obsahuje množinu souborů dle toho, jak byl obraz postupně tvořen. Ve výsledném kontejneru jsou vrstvy souborovým systémem „sloučeny" a tváří se pro proces v kontejneru jako jednotný souborový systém. [31] Vrstvy, ze kterých se výsledný souborový systém skládá, jsou určeny pouze pro čtení. Díky tomu dokáže Docker šetřit místo na disku, protože neměnnost vrstev přináší možnost jejich sdílení napříč kontejnery. Aby se do souborového systému kontejneru dalo zapisovat, Docker pro něj vytváří novou vrstvu, která je zapisovatelná.

Problém ale nastává, když dojde k smazání kontejneru a spolu s ním i k smazání zapisovatelné vrstvy souborového systému – a tedy i smazání uložených dat. Toto není problém pro kontejnery, které představují nějakou aplikaci, která nepotřebuje ukládat perzistentně svůj stav. Například pro databázi už ale takové chování není žádoucí.

Řešením jsou takzvané *Docker volumes*, které umožňují dovnitř do kontejneru namapovat nějaký adresář z hostitelského systému. Právě do tohoto adresáře lze v kontejneru ukládat data, která je potřeba uchovat napříč jednotlivými kontejnery. Když pak dojde ke smazání kontejneru, data uložená v tomto *Docker volume* zůstanou uložena na hostitelském systému a mohou být později připojena k novému kontejneru.

3.1.2 Docker Compose

Pro ještě snadnější rozběhnutí celého prostředí včetně webové služby jsem zvolil použití nástroje Docker Compose. Jde o nástroj, který je součástí Dockeru, a jeho účelem je zjednodušené spouštění více kontejnerů, které jsou na sobě nějakým způsobem závislé.

Úkony, které Docker Compose vykonává nad rámec běžného `Dockerfile`, je spouštění více kontejnerů najednou, či správa *Docker volumes* nebo Docker sítě propojující kontejnery. Docker Compose používá konfigurační soubor `docker-compose.yml`, který do značné míry nahrazuje výše zmíněný `Dockerfile`. V `docker-compose.yml` lze definovat seznam služeb – tedy kontejnerů, které mají být nástrojem spouštěny. Každá služba musí mít svůj unikátní název a obraz, na kterém má být postavena. Běžně je ale pro jejich konfiguraci použita řada dalších parametrů. Konkrétní použité parametry popisuji v následujících částech této podkapitoly.

Obsluha Docker Compose je jednoduchá. Pro spuštění všech kontejnerů definovaných v `docker-compose.yml` stačí použít příkaz `docker-compose up`.

V praxi tak stačí při zakládání vývojového prostředí na jiném počítači použít pouze jeden příkaz a Docker se o vše postará.

3.1.3 Databáze

Při tvorbě vývojového prostředí jsem se nejdříve zaměřil na samotnou databázi. Dle informací od Tichého světa by měla na hostingu Tichého Jazyka běžet databáze MariaDB 10.3. V repozitáři Docker obrazů Docker Hub jsem si tedy našel korespondující obraz pro tuto verzi.

Aby zůstala databáze perzistentně uložena i po případném smazání kontejneru, v konfiguračním souboru `docker-compose.yml` definuji pro databázovou službu *Docker volume*, který je uvnitř kontejneru namapován na adresář `/var/lib/mysql`, což je lokace, kam MariaDB ukládá své datové soubory.

Nejedná se však o jediný *Docker volume*, který tato služba používá. V dokumentaci MariaDB obrazu na portálu Docker Hub se lze dozvědět, že obraz MariaDB umožňuje inicializaci databáze pomocí SQL souboru při prvním spuštění kontejneru. Toto je přesně funkcionalita, která se pro mé účely hodí, protože mi umožňuje automaticky vytvořit instanci databáze, která bude rovnou obsahovat data z SQL *dumpu* produkční databáze Tichého jazyka.

Dump databáze je kontejneru předán právě pomocí *Docker volume*, a to na umístění `/docker-entrypoint-initdb.d` uvnitř kontejneru. Kontejner dokonce umožňuje použití SQL souboru komprimovaného pomocí gzip. To mi velice jednoduše umožnilo zmenšit velikost souboru z původních 180 MB na necelých 13 MB, což je velikost, která může být bez větších starostí uložena do Git repozitáře. Repozitář si tak s sebou nese všechno, co potřebuje ke spuštění databáze naplněné daty.

Další konfigurace služby MariaDB se již týká hlavně přihlašovacích údajů k databázi, názvu databáze a vystavení portu 3306 ven z Docker sítě do hostitelského OS. To sice není pro běh kontejneru a jeho spolupráci s ostatními službami nutné, ale mě jakožto vývojáři to umožňuje jednodušeji nahlížet do databáze pomocí programů s grafickým rozhraním pro práci s SQL databází.

3.1.4 Prostředí pro běh PHP

Webový hosting běží na PHP verze 7.2, a proto jsem na repozitáři Docker Hub našel odpovídající verzi obrazu. PHP ale vždy nabízí vybranou verzi v několika variantách. Pro mé účely jsem zvolil variantu `php-7.2-cli`[9], která není určena pro produkční použití a hodí se naopak spíše pro vývoj.

Vlastní PHP obraz

V obrazu `php-7.2-cli` mi však nějaké nástroje chyběly. Rozhodl jsem se proto vytvořit svůj vlastní soubor `Dockerfile`, který rozšíří funkce obrazu `php-7.2-`

[9] `https://hub.docker.com/_/php`

cli o mé specifické požadavky.

Docker Hub obsahuje kromě oficiálních obrazů i mnoho obrazů, které sestavila komunita. Velmi pravděpodobně by tedy šlo nalézt hotový obraz, který poskytuje všechny mnou požadované nástroje. Nechtěl jsem se ale od oficiálního obrazu odchýlit použitím neoficiálního obrazu, který může mít horší uživatelskou podporu. Další věcí je, že jsem si chtěl udržet kontrolu nad dodatečně přidanými nástroji, abych udržel velikost obrazu na rozumné hodnotě.

Prvním z nástrojů, který mi v obrazu chyběl, je příkaz symfony, který ulehčuje práci při vývoji Symfony aplikací. Samozřejmostí pro vývoj PHP aplikací je dále správce závislostí Composer, který v obrazu také chyběl. Posledním nástrojem, který mi v obrazu php-7.2-cli chyběl, byl nástroj XDebug. Ten nabízí integraci s různými vývojovými nástroji IDE, do kterých přináší možnost debuggingu kódu.

PHP v základu poskytuje omezenou funkcionalitu, a tak je často potřeba ji rozšířit pomocí PHP rozšíření (*extensions*). Mně konkrétně v obraze chybělo rozšíření zip (vyžadováno nástrojem Composer) a pdo_mysql pro komunikaci s databází. Pro snazší instalaci rozšíření naštěstí oficiální PHP obraz obsahuje příkaz docker-php-ext-install, takže se nejednalo o nic složitého.

Do Dockerfile jsem přidal příkazy pro instalaci všech zmíněných nástrojů a rozšíření. Dockerfile jsem uložil na fakultní GitLab ve formě nového Git repozitáře s názvem php-symfony-docker[10].

Docker Compose dokáže v konfiguračním souboru docker-compose.yml referencovat Docker obraz pro danou službu dvěma způsoby. Prvním je odkaz na konkrétní Dockerfile soubor. V takovém případě Docker Compose zajistí sestavení obrazu. [32] Druhým způsobem je odkazování na již sestavený obraz z nějakého repozitáře Docker obrazů. Výchozím repozitářem je Docker Hub, je ale možné odkazovat i na jakýkoliv jiný repozitář.

Jednou z funkcí GitLabu je právě možnost vytvořit si vlastní privátní repozitář Docker obrazů (*Docker Image Registry*). Do tohoto repozitáře lze nahrávat své obrazy přímo z lokálního úložiště ve svém počítači pomocí příkazu docker push. Tuto metodu jsem zpočátku použil. Použití sestaveného obrazu v docker-compose.yml je podobné jako kdyby se jednalo o obraz z repozitáře Docker Hub. Jediným rozdílem je nutnost uvést adresu repozitáře na doméně gitlab.fit.cvut.cz. V souboru docker-compose.yml pak tedy stačilo referencovat Docker obraz pomocí následující URL:

```
gitlab.fit.cvut.cz:5000/zakjindr/php-symfony-docker:latest
```

Automatické sestavení Docker obrazu

Později jsem se ale rozhodl sestavování tohoto obrazu a jeho uložení do repozitáře obrazů automatizovat pomocí CI/CD skriptu. Vytvořil jsem tedy v re-

[10] https://gitlab.fit.cvut.cz/zakjindr/php-symfony-docker

pozitáři soubor `.gitlab-ci.yml`, který slouží jako konfigurace CI/CD skriptů pro systém GitLab.

Jeho běh se skládá z jedné fáze. Pro automatizaci procesu totiž stačí zavolat příkazy pro přihlášení se k repozitáři obrazů, sestavení obrazu a jeho nahrání do repozitáře. Přihlášení k repozitáři obrazů probíhá skrz můj účet na gitlab.fit.cvut.cz. Abych nemusel ukládat mé přihlašovací údaje do souboru (a riskovat tak jejich vyzrazení), používám v souboru `.gitlab-ci.yml` několik proměnných prostředí, které GitLab běhovému prostředí automaticky poskytuje.

Aby vůbec bylo možné v rámci skriptu sestavit Docker obraz, bylo potřeba nejdříve zpřístupnit v rámci CI/CD skriptů příkaz `docker`. GitLab ve své dokumentaci nabízí několik způsobů, jak toho docílit. [33]

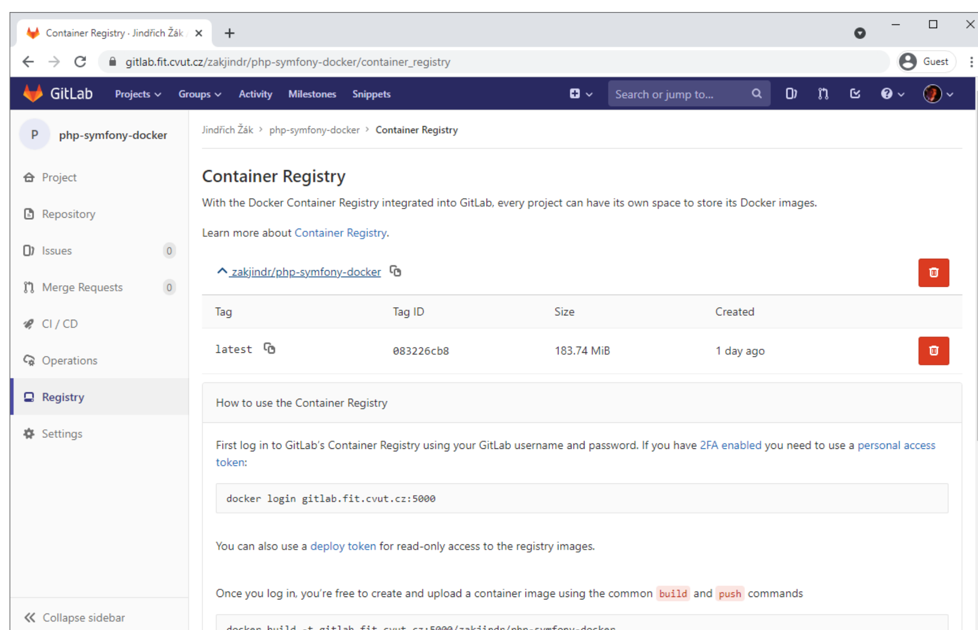
Prvním způsobem je používat pro spouštění CI/CD skriptů tzv. *Shell executor*. To znamená, že skripty jsou spouštěny přímo na zařízení, na kterém je nainstalován daemon `gitlab-runner` spouštějící CI/CD skripty. Aby měl uživatel, prostřednictvím kterého jsou skripty vykonávány, přístup k příkazu `docker`, musí být členem skupiny `docker`, což mu ve výsledku uděluje práva `root`. [33] Ačkoliv je tento přístup nejjednodušší, není z hlediska bezpečnosti doporučován. Na fakultní instanci GitLab se proto nepoužívá. Místo *Shell executoru* je použit *Docker executor*, takže skripty jsou vždy vykonávány uvnitř kontejneru, který má omezená práva.

Způsob, který pro zpřístupnění příkazu `docker` nevyžaduje použití *Shell executoru*, je použití *Docker executoru* a v něm spuštění dalšího kontejneru s Dockerem (*Docker-in-Docker*). [33] Existuje totiž oficiální Docker obraz nástroje Docker, a tak stačilo upravit `gitlab-ci.yml` skript tak, aby používal právě tento kontejner pro získání přístupu k příkazu `docker`. Následně už stačilo pouze provést přihlášení k danému repozitáři Docker obrazů, sestavit Docker obraz a pomocí `docker push` jej do repozitáře obrazů nahrát. Nahraný obraz je vidět v rozhraní GitLabu v sekci „Registry“.

Konfigurace v Docker Compose

PHP služba v rámci Docker Compose vystavuje port 8000 pro webový server a 9003 pro spojení s nástrojem XDebug. Zdrojový kód Symfony aplikace je do kontejneru namapován pomocí *Docker volumes*. Jakmile tedy dojde ke změně kódu v hostitelském operačním systému, změna je okamžitě dostupná i uvnitř kontejneru. Po spuštění kontejneru se spustí příkaz `composer install`, který nainstaluje všechny PHP závislosti a vývojář se tak o jejich instalaci nemusí starat.

Služba navíc definuje několik proměnných prostředí. Jde zejména o konfiguraci spojení s databází. Vyplněné hodnoty korespondují s hodnotami konfigurace MariaDB databáze. Díky nástroji Docker Compose, který nastaví každému kontejneru jeho *hostname* na základě jeho názvu, jsem nemusel za-



Obrázek 3.1: Repozitář Docker obrazů v systému GitLab

dávat pro referencování kontejneru s databází jeho IP adresu, která navíc není fixní. Stačilo na tento kontejner odkazovat pomocí jeho názvu – tedy `mariadb`.

3.1.5 Moodle

Kontejnery s databází a PHP by pro vývoj webového API bohatě stačily. Napadlo mě ale vytvořit kompletní prostředí Tichého jazyka ještě přidáním kontejneru s aplikací Moodle. Oficiální Docker obraz pro Moodle pod názvem `moodlehq/moodle-php-apache` sice existuje, ale dokumentace jeho konfigurace není příliš bohatá.

Proto jsem raději sáhl po obrazu `bitnami/moodle`[11], který má dokumentaci mnohem lepší, a proto pro mne bylo mnohem snadnější jej v rámci `docker-compose.yml` správně odkázat na kontejner s databází. Obraz `bitnami/moodle` má navíc dle Docker Hub zhruba pětkrát vyšší počet stažení, a i jeho repozitář na GitHubu je jednoznačně populárnější. To slibuje lepší podporu od komunity.

Aby bylo možné si Moodle otevřít ve webovém prohlížeči na hostitelském stroji, vystavil jsem jeho server na portu 8080. Díky zahrnutí kontejneru se systémem Moodle do vývojového prostředí jsem získal možnost zobrazovat si stejná data jak v mobilní aplikaci, tak v instanci Moodle, což umožnilo jednoduše porovnávat kvalitu zobrazení dat v těchto dvou aplikacích.

[11] `https://hub.docker.com/r/bitnami/moodle`

Pro zmíněné tři služby PHP, MariaDB a Moodle jsem sestavil konfiguraci v `docker-compose.yml`. Jelikož služby `php` a `moodle` závisejí na službě s databází `mariadb`, definoval jsem v tomto konfiguračním souboru ještě jejich vzájemné závislosti pomocí atributu `depends_on`. Ten zajistí postupné spouštění kontejnerů na základě jejich vzájemných závislostí.

Zapnutí a vypnutí celého prostředí pro běh aplikace tak je otázkou spuštění jednoho příkazu, a to i v případě že se jedná o počítač, na kterém toto prostředí nikdy nebylo spuštěno. Jediným předpokladem pro spuštění prostředí se tak stává přítomnost nástroje Docker (spolu s Docker Compose), který se o zřízení vývojového prostředí kompletně postará.

3.2 Vývoj Symfony aplikace

Po vytvoření prostředí pro vývoj jsem se mohl pustit to vývoje samotného webového API postaveném na frameworku Symfony. Pomocí příkazu `symfony`, který jsem přidal do Docker obrazu s PHP, jsem založil projekt na Symfony ve verzi 5.2.

Nejdříve bylo potřeba provést konfiguraci připojení k databázi, která se standardně provádí prostřednictvím proměnné prostředí `DATABASE_URL`. V ní Symfony očekává URL obsahující všechny potřebné parametry pro připojení do databáze. Aby nebylo nutné nastavovat proměnné prostředí ručně, Symfony pracuje se souborem `.env`, který na jednom místě drží všechny hodnoty proměnných prostředí.

Parametry potřebné pro připojení k databázi jsem však již jako proměnné prostředí definoval v souboru `docker-compose.yml`. Aby nedocházelo k jejich duplicitní definici znovu v souboru `.env`, pracuji zde s proměnnými pocházejícími právě z `docker-compose.yml`. Výsledná URL databáze tak vypadá takto:

```
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${
    DB_PORT}/${DB_DATABASE}
```

3.2.1 Modelové třídy

Abych měl jasně a pevně definovanou strukturu dat v REST API, připravil jsem si na úrovni webové aplikace sadu modelových tříd. Tyto třídy přesně představují entity, které vyžaduje mobilní aplikace Tichý jazyk. Jedná se konkrétně o třídy `Module`, `Lecture`, `LectureChapter`, `Glossary`, `GlossaryEntry` a `SearchResult`.

Třídy `Module` a `Lecture` představují moduly a lekce, ze kterých se skládají studijní materiály. Názvem `LectureChapter` jsem pojmenoval třídu obsahující videa obsažená v lekcích. Jelikož tato entita obsahuje kromě videí ještě například název a popis a má vztah ke konkrétní lekci, usoudil jsem, že název „Video“ by nepopisoval daný objekt úplně správně. Proto jsem radši zvolil název `LectureChapter`, ze kterého je ihned patrná vazba na objekt typu `Lecture`.

Třídy `Glossary` a `GlossaryEntry` představují slovník, respektive jednotlivá slovíčka v něm obsažená. Třída `SearchResult` v sobě drží informace potřebné k vykreslení výsledků vyhledávání.

Modelové třídy jsem implementoval jako neměnné (*immutable*), což je běžná praxe funkcionálního programování. Neměnné datové struktury programátorovi zaručí, že v nich vždy nalezne stále stejná data, což umožňuje snadněji přemýšlet nad chováním programů a může to tak pomoci k vyvarování se chyb v kódu. [34] Výhodou neměnných datových struktur je dále to, že umožňují v kódu daleko snadnější nalezení místa původu případné chybné hodnoty.

V mobilní aplikaci existují případy (například obrazovka se seznamem videí z vybrané lekce), kde potřebuji mít o daném objektu informace z různých API endpointů. Abych se vyvaroval nadbytečným HTTP dotazům, upravil jsem strukturu dat HTTP odpovědí tak, aby obsahovala vše, co mobilní aplikace potřebuje. Na endpointu se seznamem videí dané lekce tak například každý objekt typu `LectureChapter` obsahuje navíc ještě název související lekce. Ten je sice možné získat z endpointu se seznamem lekcí, ale to vyžaduje nadbytečný HTTP dotaz, kterému se tímto způsobem vyhýbám.

3.2.2 Controller

Pro vytvoření API endpointů jsem vytvořil třídu `DefaultController`. Ve frameworku Symfony je `Controller` třída, která obsahuje sadu metod představujících API endpointy, pro které se v kontextu Symfony používá termín „route“. V rozsáhlejších Symfony aplikacích dává smysl rozdělit vzájemně související endpointy do více `Controller` tříd, v případě tohoto projektu jsem použil pouze jednu `Controller` třídu, protože endpointů není potřeba příliš mnoho.

Konfigurace API endpointu se v Symfony standardně tvoří pomocí PHP anotace `@Route`. Konkrétní endpoint může vypadat například takto:

```
/**
 * @Route(path="/modules/{id}/lectures", methods={"GET"})
 */
public function getModuleLectures(string $id) : Response
{
  // ...
}
```

Tento příklad dokáže obsloužit HTTP dotaz metody GET mířící na URL s cestou `/modules/{id}/lectures`. Identifikátor modulu `id` je tzv. *placeholder* a jeho konkrétní hodnotu získá metoda ve formě stejnojmenného argumentu. Úkolem metody je na základě HTTP dotazu sestavit HTTP odpověď. Tu metoda vrací ve formě objektu třídy `Response`.

Aby měla `Controller` třída ve svých metodách přístup k různým službám Symfony frameworku jako je například komunikace s databází, objekt

pro ukládání záznamů do logu a další, Symfony umožňuje jejich *dependency injection*. Ve výsledku tak stačí pouze pro danou metodu definovat potřebné služby ve formě argumentů s jejich typy a ve chvíli, kdy je kód daného endpointu spuštěn, se Symfony pokusí naplnit tyto argumenty konkrétními objekty.

3.2.3 Datová vrstva

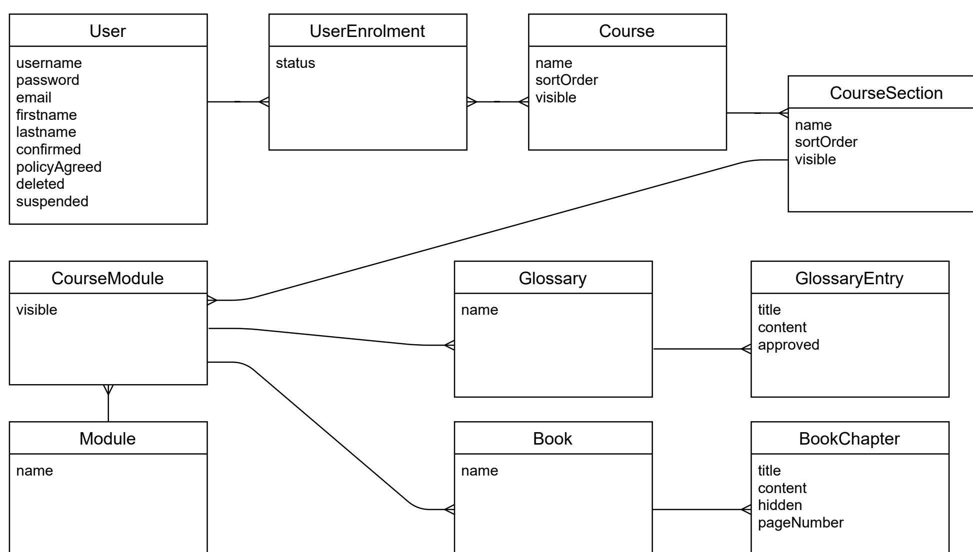
Abych mohl v začátcích brzo zprovoznit komunikaci mezi mobilní aplikací a webovým API, HTTP odpovědi jsem v `DefaultController` třídě zpočátku sestavoval na základě ručně vytvořeného vzorku dat v kódu. To mi umožnilo brzy otestovat funkčnost komunikace mezi Flutter aplikací a webovým API a dostat tak mobilní aplikaci do bodu, kdy místo samých obrazovek s načítacími animacemi ukazuje nějaká data.

Postupně jsem však samozřejmě musel postoupit k implementaci čtení dat z databáze Moodle. Symfony používá knihovnu Doctrine, s pomocí které jsem vytvořil potřebné entitní třídy. Mapování entitních tříd probíhá opět s pomocí PHP anotací. Strukturu dat v Moodle samozřejmě nelze měnit, a tak probíhalo ORM mapování směrem z databáze do entitních tříd. Základní kostry entitních tříd byly vytvořeny s pomocí symfony příkazu `make:entity`, který se formou interaktivního průvodce zeptá programátora na všechny atributy a jejich datové typy a na základě těchto údajů vygeneruje v PHP entitní třídu s potřebnými anotacemi, atributy, gettery a settery.

Databázové schéma Moodle obsahuje řadu nekonzistencí týkajících se jmenné konvence. Studium struktury dat ztěžuje například nesystematické pojmenování sloupců s cizími klíči (`course`, `courseid`, `book_id`) nebo sloupců specifikujících pořadí záznamů (`sortorder`, `section`). Evidentně neustálené používání jednotných a množných čísel v názvech tabulek (`mdl_course`, `mdl_modules`) je už pouhý detail. Tyto nesourodosti řeším na úrovni ORM, kde pro větší přehlednost následné práce s daty používám takové názvosloví, které je napříč ostatními entitními třídami jednotné.

Entitním třídám z Moodle databáze jsem pro jasné odlišení (zejména od modelových tříd) do jejich názvu přidal prefix `Moodle`. S pomocí projektu Examulator.com, který slouží jako jakási neoficiální dokumentace struktury databáze Moodle, jsem sestavil ORM mapování atributů a vztahů mezi entitami. Vznikly tak třídy jako `MoodleCourse`, `MoodleBook`, `MoodleUser` a další. Namapované entitní třídy neobsahují úplně všechny vztahy ale pouze ty, které se pro logiku webové služby hodí. Mapování všech vztahů by jednak bylo zbytečné a jednak by teoreticky mohlo mít negativní dopad na výkon ORM.

Webová služba potřebuje k získání dat pro jednotlivé API endpointy vykonávat sadu dotazů s řadou `JOIN` a `WHERE` klauzulí. Implementaci databázových dotazů jsem oddělil do `Repository` tříd, což je standardní způsob Doctrine, jak vlastní složitější SQL dotazy implementovat. Dotazy není nutné psát v čistém SQL, Doctrine umožňuje jejich tvorbu pomocí objektu `QueryBuilder`, který



Obrázek 3.2: ER diagram znázorňující entity v systému Moodle potřebné pro čtení dat studijních materiálů

používá návrhový vzor *builder*. Každá `Repository` třída poskytuje zbytku aplikace vysokoúrovňové API, což programátorovi umožňuje psát ve zbytku aplikace kratší a přehlednější kód.

Pro získání studijních materiálů Tichého jazyka je potřeba číst data z několika tabulek. Potřebné tabulky a vztahy mezi nimi jsou znázorněny na ER diagramu 3.2.

Aby API poskytovalo správná data, pro tabulky představující studijní modul nebo lekci je potřeba zjistit, zda obsahuje alespoň jednu aktivní entitu `MoodleBookChapter`. Nedávalo by totiž smysl nabízet uživateli v seznamu modulů modul, v jehož lekcích se nevyskytují žádná videa.

Zároveň je potřeba řešit autorizaci přístupu daného uživatele k studijním materiálům a to tak, že je nutné se skrz vazby dostat od tabulky `mdl_course` přes tabulky s informacemi o zápisech do kurzů až k samotnému uživateli.

Transformace z instancí entitních tříd (tedy z konkrétních záznamů v databázi) na instance modelových tříd, které je schopna mobilní aplikace konzumovat, jsem umístil do třídy `MoodleMapper`. Tato třída obsahuje sadu statických funkcí, jejichž jména začínají prefixem `map`, a mají tedy za úkol provést nějaké mapování z entitní třídy na třídu modelovou. Jejich logika většinou neobsahuje nic složitého, většinou jde prakticky jen o volání konstruktoru modelové třídy v nějakém `for` cyklu.

Kromě transformací z entitních tříd na modelové obsahuje `MoodleMapper` ještě několik statických funkcí, které provádějí transformaci konkrétních hodnot. Mimo jiné je zde implementována funkce, která extrahuje YouTube URL

z řetězce obsahujícího HTML kód, což je způsob uložení videí v databázi Moodle.

Dále se zde vyskytuje funkce pro extrakci popisku videa z HTML kódu. Během analýzy dat jsem přišel na to, že textový WYSIWYG editor v systému Moodle neprodukuje vždy validní HTML. Proto se zde plně nemohu spoléhat na PHP funkci `strip_tags`, která odstraňuje HTML značky, ale některé úpravy zde provádím manuálně. Na základě pozorování různých dat jsem tak implementoval funkci `extractDescriptionFromWysiwygHtml` a její správnou funkčnost jsem podložil testy.

3.2.4 Autentizace uživatelů

V kapitole 1.6.3 Autentizace uživatelů jsem definoval způsob, jakým realizovat autentizaci uživatelů s použitím JWT tokenu. Pro jeho implementaci jsem nejdříve do projektu nainstaloval balíček `LexikJWTAuthenticationBundle` , který se o vytváření JWT tokenů stará.

Konfigurace zabezpečení a JWT tokenů

Centrálním místem, kde se dá konfigurovat zabezpečení aplikace, je soubor `config/packages/security.yaml`. Pro zřízení požadované funkcionality je nejdříve potřeba definovat tzv. *User Provider*. *User Provider* je objekt implementující rozhraní `UserProviderInterface`, prostřednictvím kterého Symfony aplikace získává data o uživatelských účtech. Typicky je použit tzv. *Entity Provider*, který poskytuje data o účtech na základě nějaké konkrétní entitní třídy, ale lze použít i *User Provider* čerpající data o uživatelích například z konfiguračního souboru. Případně lze použít kompletně vlastní implementaci.

Jelikož se data o uživatelích e-learningu vyskytují v databázi systému Moodle, využil jsem zde právě `Entity Provider` a nasměroval jej na již dříve vytvořenou entitní třídu reprezentující uživatelské účty `MoodleUser`. Aby *User Provider* fungoval správně, bylo potřeba, aby třída `MoodleUser` implementovala rozhraní `UserInterface` definované v knihovně `Symfony\Component\Security`.

Dalším krokem v konfiguraci zabezpečení bylo nakonfigurování parametru `encoder`, který určuje, jakým algoritmem jsou zahashována hesla. Z analýzy vím, že Moodle používá bcrypt.

Poslední důležitou věcí, kterou je potřeba v souboru `security.yaml` nakonfigurovat, jsou takzvané *firewally*. Symfony chápe *firewall* jako nějakou množinu URL cest, na kterou jsou aplikována stejná bezpečnostní pravidla. [35] Množina cest je zde popsána regulárním výrazem.

Nejdůležitější *firewall* v této webové aplikaci se týká přímo samotného REST API. Množinu jeho URL cest lze jednoduše popsat regulárním výrazem `^/api`. Tento *firewall* čerpá uživatele z výše zmíněného objektu *Entity Provider*. Protože JWT tokeny obsahují všechna potřebná data o svém vlastníkovi uvnitř sebe, tento *firewall* jsem označil příznakem `stateless`, což znamená,

že Symfony nebude ukládat žádná data o autentizovaném uživateli na straně serveru. Kromě *firewallu* pro REST API se v `security.yaml` vyskytují ještě další *firewally*. Jiným pravidlům totiž podléhá například URL pro přihlášení nebo pro obnovení JWT tokenu.

Způsob, jakým do projektu integrovat `LexikJWTAuthenticationBundle`, je pomocí parametru `guards.authenticators` v souboru `security.yaml`. Zde referencuji třídu `JWTTokenAuthenticator` pocházející právě z tohoto balíčku. Třída `JWTTokenAuthenticator` dědí od abstraktní třídy `AbstractGuardAuthenticator`, a jejím úkolem je právě implementovat mechanismus pro autentizaci uživatelů.

Za povšimnutí zde stojí i parametr `user_checker`, který v tomto případě odkazuje na mnou vytvořenou třídu `UserChecker`. Ta mi umožňuje provádět dodatečné kontroly uživatelů nad rámec třídy `JWTTokenAuthenticator`. V praxi ji používám pro kontrolu, zda například uživatelský účet není označen jako smazaný, deaktivovaný nebo zda má potvrzenou emailovou adresu.

`LexikJWTAuthenticationBundle` ještě vyžaduje pro své fungování v souboru `config/packages/lexik_jwt_authentication.yml` konfiguraci dvojice veřejného a soukromého klíče pro podepisování JWT tokenů. V tomto souboru se však pouze odkazuji na proměnné prostředí, abych měl možnost tyto hodnoty nastavit pro různá prostředí zvlášť. Konkrétní hodnoty se tedy vyskytují až v souboru `.env`.

Refresh token

Funkcionalitu pro práci s *refresh* tokeny `LexikJWTAuthenticationBundle` nepřináší, je potřeba nainstalovat balíček `GesdinetJWTRefreshTokenBundle`, který s výše zmíněným balíčkem spolupracuje. V konfiguračním souboru `config/packages/gesdinet_jwt_refresh_token.yaml` vyžaduje konfiguraci *User Provider* a *User Checker* tříd, které jsou pro JWT autentizaci používány.

Jelikož se v případě *refresh* tokenu nejedná o JWT token, který si informace nese uvnitř sebe, je potřeba aktivní *refresh* tokeny ukládat na straně serveru. Pravděpodobně by nebyl problém vytvořit si v Moodle databázi novou tabulku pro tyto účely, ale raději jsem se rozhodl využít jiné úložiště, abych se pro jistotu zcela vyhnul modifikaci Moodle databáze.

Knihovna Doctrine umožňuje v rámci projektu používat více databází. Této příležitosti jsem využil pro nakonfigurování jednoduché SQLite databáze, která bude sloužit právě jako úložiště *refresh* tokenů. Díky tomu se Symfony aplikace kompletně vyhne modifikacím Moodle databáze, a tak zůstane úložiště těchto dvou aplikací odděleno.

3.2.5 Vývojový HTTP klient a monitoring chyb

Po implementaci autentizace uživatelů se však objevila menší komplikace pro vývoj. Před implementací autentizace totiž byly všechny API endpointy veřejně dostupné, a tak se dala data z API prohlížet i přes webový prohlížeč

otevřením dané URL. S autentizací však již toto přestalo být ve webovém prohlížeči jednoduše možné.

Musel jsem si tedy zvolit nějakého HTTP klienta, který by mi umožnil pracovat s HTTP hlavičkami pro vkládání JWT tokenů a dovolil by mi tak i nadále manuálně testovat vyvíjené API. Pro tyto účely existují široce používané nástroje jako Postman, pro Symfony existuje i balíček `NelmioApiDocBundle`, který přináší webové rozhraní Swagger pro interakci s API. Samozřejmě by nebylo dobré zapomenout ani na příkaz `curl`.

Nakonec jsem se ale rozhodl použít využít HTTP klienta, který je dostupný ve vývojářských prostředích firmy JetBrains. Existuje i velmi podobná alternativa ve formě doplňku pro Visual Studio Code. Pro použití klienta stačí vytvořit jednoduchý textový soubor `requests.http`, do kterého lze přímo psát konkrétní HTTP dotazy. Ve stejném souboru lze navíc i zpracovávat HTTP odpovědi pomocí jednoduchých skriptů a uložit si tak například některá data do proměnných pro použití v dalších HTTP dotazech. Tímto způsobem si v souboru `requests.http` ukládám dvojici tokenů získaných ze serveru a následně je v dalších HTTP dotazech používám pro autentizaci.

```
#http/requests.http

### Login with credentials
POST http://localhost:8000/api/login_check
Content-Type: application/json

{
  "username": "uzivatel",
  "password": "heslo"
}

> {%
//script that saves tokens to variables
client.global.set("jwt", response.body.token);
client.global.set("refresh", response.body.refresh_token);
%}


### Get modules with received token
GET http://localhost:8000/api/modules
Authorization: Bearer {{jwt}}
```

Zdrojový kód 2: Ukázka práce s tokeny v souboru `requests.http`

Ve výsledku tedy stačí pouze postupně volat HTTP dotazy definované v tomto souboru bez nutnosti je pokaždé upravovat a nastavovat tokeny ručně.

Příjemným bonusem je, že soubor `requests.http` lze zahrnout do repozitáře v Gitu a kdykoli později se k němu vrátit. Soubor `requests.http` se nachází v projektu s webovou službou v adresáři `http`.

3.2.6 Continuous Integration

Pro zajištění kvality kódu jsem se rozhodl i pro projekt webové služby aplikovat principy Continuous Integration. Kvalitu kódu při integraci nových funkcí kontroluji jednotkovými a funkcionálními testy. Pro jejich spouštění používám standardní testovací framework PHPUnit, který úzce spolupracuje s frameworkem Symfony. Veškeré testy jsou k nalezení v adresáři `tests`.

Testy

Jednou z klíčových oblastí, kterou bylo potřeba otestovat, byla funkční autentizace. Proto jsem vytvořil testovou třídu `AuthenticationTest`, která obsahuje funkcionální testy, které netestují pouze nějakou omezenou část kódu, ale interagují s projektem podobným způsobem, jako s ním bude interagovat mobilní aplikace v reálném nasazení.

Symfony ve své dokumentaci obsahuje kvalitní článek[12], jak pracovat s testovacím HTTP klientem, prostřednictvím kterého lze testovat poskytované API. V testech tak konkrétně testuji přístup k jedné z URL z REST API, ke které je potřeba vlastnit validní JWT token.

Testy jsou spouštěny v separátním prostředí, pro které mají definované konkrétní proměnné v souboru `.env.test`. V něm jsem nastavil, aby Symfony aplikace pracovala s dočasnou SQLite databází místo s MariaDB kontejnerem s kompletními daty Moodlu. Před každým testem si ve funkci `setUp` získávám referenci na databázi. Vzápětí vytvořím v SQLite databázi databázové schéma na základě namapovaných entitních tříd a vložím do ní testovací data.

Následuje samotné provedení testů, které se zde skládá hlavně z testování očekávaných HTTP odpovědí v různých situacích. Simuluji zde pokusy přihlásit se prostřednictvím smazaného, deaktivovaného, neověřeného a aktivního účtu. U jakkoli neaktivních účtů kontroluji, zda jim byl odepřen přístup k získání tokenů. U aktivního účtu dále testuji vytvoření a funkčnost *refresh* tokenu a přístup k API endpointům.

Statická analýza kódu

Kromě testů používám během Continuous Integration ještě nástroj pro statickou analýzu kódu. Ačkoliv je PHP dynamický, slabě typovaný jazyk, s novějšími verzemi se do jazyka dostávají volitelné prvky, které pomáhají programátorovi specifikovat typy výrazů, [36] takže vývojové prostředí pak dokáže

[12] `https://symfony.com/doc/current/testing.html#write-your-first-application-test`

provádět statické typové kontroly, což je velmi užitečné při vývoji. Tyto prvky jazyka například dovolují specifikovat typ návratové hodnoty funkce, typy jejích argumentů a podobně.

Definice typů ovšem nestačí pokrýt naprosto vše potřebné, a proto existuje ještě způsob dokumentace PHPDoc, pomocí kterého lze specifikovat další informace pomocí speciálních komentářů. [37] PHP bez použití PHPDoc například dokáže specifikovat typ dynamického pole pouze klíčovým slovem `array` (platí pro verzi 7.2). To ale programátorovi ani statickému analyzátoru nedává žádnou informaci o obsahu daného dynamického pole. Takovou informaci dokáže specifikovat až PHPDoc.

Pro sledování kvality kódu jsem v rámci CI skriptu přidal statický analyzátor PHPStan. Kromě standardních PHP specifikací typů analyzuje právě i dokumentaci ve formátu PHPDoc. PHPStan je distribuován ve formě Composer balíčku. Po jeho instalaci přináší do projektu spustitelný soubor `phpstan`, který statickou analýzu provádí. Nástroj je možné spouštět s naprosto minimální konfigurací, pro pokročilejší použití ale nabízí možnost konfigurace pomocí souboru `phpstan.neon`, ve kterém je například možné definovat adresáře vyloučené ze statické analýzy.

PHPStan umožňuje upravit míru striktnosti statické analýzy. Míru striktnosti je možné u příkazu `phpstan` upravit přepínačem `--level`, který přijímá hodnoty od 0 do 8, kde platí že čím vyšší hodnota, tím striktnější jsou kontroly. Doporučovaný postup je začít s analýzou na hodnotě 0, nalezené chyby opravit a postup opakovat s vyšší striktností analýzy. [38] V mém případě jsem se dostal na maximální úroveň 8, která kontroluje i takzvané *nullable* datové typy. Tento typ kontrol dokáže odhalit potenciální místa vzniku chyb při dereferencování `null` ukazatele, a je tedy velmi žádoucí jej dosáhnout.

Kapitola **4**

Implementace aplikace

Pravděpodobně největší část času na praktické části této práce zabraly úpravy kódu mobilní aplikace. Prototyp od Alžbety Gogolákové posloužil jako skvělý základ, ale vzhledem k mírně odlišným potřebám Tichého světa, které se od tehdejší doby změnily, a novým přístupům při tvorbě Flutter aplikací došlo v implementaci k řadě změn. Co se konkrétně v implementaci změnilo, popisuji v této kapitole.

Projekt od Alžbety Gogolákové se vyskytuje na fakultním GitLabu. Po domluvě s Alžbetou jsem si repozitář zkopíroval do svého jmenného prostoru (*fork*) na GitLabu.

V rámci hledání způsobů, jak provádět automatizované sestavení iOS verze aplikace v CI/CD prostředí, jsem narazil na překážku, kterou je nemožnost spouštět na fakultní instanci GitLab skripty na zařízení s macOS. Tento operační systém je totiž podmínkou pro spuštění nástroje Xcode, který provádí sestavení mobilních aplikací pro iOS. Provedl jsem tedy průzkum veřejně dostupných služeb, které umožňují spouštění CI/CD skriptů na macOS zařízeních.

Vzhledem k nepříliš široké nabídce takové služby jsem nakonec našel jen několik málo plnohodnotných kandidátů. Jeden z prvotních způsobů řešení, které mě napadly, bylo zřízení virtuálního privátního zařízení s macOS a jeho připojení k fakultní instanci GitLab v roli *runner*. Ukázalo se však, že zřízení takového zařízení je poměrně nákladná záležitost[13], a tak jsem se raději poohlédl po sdílených službách.

Jako nejzajímavější jsem nakonec vyhodnotil služby TravisCI a GitHub Actions. Obě služby standardně nabízejí pro běh skriptů na macOS zařízeních měsíční limit 200 minut pro privátní repozitáře. GitHub ale nabízí program pro studenty, který nabízí limit 300 minut za měsíc. Proto jsem se rozhodl projekt přesunout do privátního repozitáře na GitHub. Ačkoliv se touto změnou odkláním od zadání práce, která výslovně zmiňuje GitLab, použití služby

[13]`https://www.macstadium.com/pricing`

GitHub Actions neznamená výrazné změny oproti službě GitLab CI. V obou případech se jedná o principielně velmi podobné služby. Celá tato změna byla navíc konzultována a schválena mým vedoucím práce.

4.1 Aktualizace Flutter závislostí

Funkční prototyp aplikace pro podporu výuky znakového jazyka v Tichém světě byl vytvořen na jaře 2019. Od té doby došlo k řadě aktualizací používaných balíčků i samotného frameworku Flutter. Tento prototyp zároveň používal balíčky, které jsem vyhodnotil jako nevhodné a bylo potřeba je nahradit.

Jazyk Dart, ve kterém se Flutter aplikace píší, používá pro správu balíčků nástroj Pub. Jeho hlavní repozitář balíčků se jmenuje Pub.dev. Ovládání nástroje Pub je postaveno na podobných principech jako správci balíčků u ostatních jazyků (například Composer nebo NPM). Poskytuje konzolový příkaz `pub`, pomocí kterého mohou být balíčky do projektu instalovány, odstraňovány a aktualizovány. Seznam přímých závislostí včetně dalších metadat projektu je uložen v souboru `pubspec.yaml`. Pro vedení záznamů o konkrétních nainstalovaných verzích balíčků používá soubor `pubspec.lock`.

Můj první krok pro zprovoznění čerstvě naklonovaného projektu bylo spuštění příkazu `pub get`, jehož úkolem je stažení potřebných balíčků a knihoven. Po jeho spuštění se mi okamžitě zobrazilo upozornění, že aplikace využívá starou verzi způsobu integrace do Android aplikací (*Android Embedding*), která je označena jako *deprecated*, a je doporučeno používat novější verzi *Android Embedding v2*.

Android Embedding se stará o obalení běhového prostředí Flutter pro možnost jeho běhu ve formě aplikace na OS Android. Zajišťuje spuštění aplikace, a poskytuje běhovému prostředí Flutteru takzvanou *aktivitu*, která v kontextu vývoje aplikací pro Android představuje obrazovku, která obsahuje nějaké uživatelské prostředí. Běžné nativní Android aplikace se skládají z několika *aktivit*, kde každá z nich poskytuje uživateli nějakou jinou funkcionalitu. Flutter runtime vyžaduje jedinou speciální *aktivitu*, ve které se následně sám ujme vykreslování grafického rozhraní. A právě zde došlo ke změnám, kdy v předchozí verzi *Android Embedding* musela tato *aktivita* dědit od třídy `io.flutter.app.FlutterActivity`, která byla označena jako *deprecated* ve prospěch novější `io.flutter.embedding.android.FlutterActivity`.

Před začátkem programování jsem se nejdříve věnoval konfiguraci vývojového prostředí. Dart SDK s sebou přináší příkaz `dart analyze`, který provádí statickou analýzu kódu. Existuje řada pravidel, která definují, co bude během analýzy kontrolováno. Aby vývojáři nemuseli sestavovat soubor používaných pravidel sami, existuje balíček `pedantic`, který obsahuje sadu pravidel pro statickou analýzu vytvořenou a používanou přímo ve firmě Google. Již z ná-

zvu vyplývá, že `pedantic` provádí poměrně přísnou kontrolu. Jeho zahrnutí do projektu je řešeno v souboru `analysis_options.yaml`.

Equatable

Dále jsem provedl aktualizaci balíčku `equatable`. Ten slouží k usnadnění porovnávání objektů, které sice nejsou jednou stejnou instancí, ale jejich obsah lze považovat za shodný. Děděním třídy `Equatable` objekt získá přetížené implementace operátoru `==` a `hashCode`. Programátorovi pak stačí v dané třídě pouze implementovat atribut `props`, který určuje, na základě kterých vnitřních proměnných má být prováděno porovnání instancí. Equatable pak zajistí, že dvě různé instance se shodným obsahem se budou rovnat. Po aktualizaci jeho verze jsem prošel všechny třídy, které dědí od třídy `Equatable` a zkontroloval jsem, zda jejich implementace koresponduje s aktuální verzí.

Nejčastěji bylo nutné upravit definici atributu `props`, který se ve staré verzi definují pomocí volání rodičovského konstruktoru. V nové verzi se definují pomocí přetížení getteru. Zároveň jsem upravil konstruktory těchto tříd tak, aby používaly modifikátor `const`. Ten značí, že instance, která s pomocí daného konstruktoru vznikne, není modifikovatelná. Dart se z nich pak může pokusit vytvořit takzvané *konstanty vzniklé při kompilaci* (*compile-time constants*), což může pomoci optimalizovat výkon. [39]

Jazyk Dart umožňuje předávat funkcím parametry běžným pozičními parametry, u kterých záleží na pořadí. Druhou možností je použít pojmenované parametry, které jsou funkcím předávány stylem *klíč – hodnota*. Pojmenované parametry jsou ve výchozím stavu vždy volitelné. Jejich použití lze vynutit anotací `@required`.

Úprava, která sice nebyla nezbytná, ale dle mého názoru zjednoduší kód, je nahrazení pojmenovaných parametrů konstruktoru s anotací `@required` pozičními parametry. Vzhledem k tomu, že poziční parametry jsou vyžadovány ve výchozím stavu, použití anotace zde již není nutné. Dle mého názoru se použití pojmenovaných parametrů hodí spíše na místech, kde existuje větší počet parametrů a některé z nich jsou navíc volitelné. Na místech, kde jsem provedl tyto úpravy, se ale jednalo vždy jen o konstruktory s maximálně dvěma parametry.

BLoC

Aktualizací prošly dále balíčky `bloc` a `flutter_bloc` pro správu stavu aplikace implementující návrhový vzor BLoC. Oba balíčky prošly od doby tvorby prototypu aplikace zásadním vývojem, a tak se konkrétně `flutter_bloc` dostal z původní verze `0.8.0` na aktuální `7.0.0`. Během tohoto období získaly balíčky `bloc` a `flutter_bloc` značnou popularitu a staly se jednou z hlavních doporučovaných přístupů pro správu stavu aplikace. [40] To napovídá tomu, že tehdejší (a dle mého názoru i mírně odvážná) volba použít právě návrhový

vzor BLoC se prokázala jako správná. U takové aktualizace o několik hlavních verzí dopředu bylo třeba vypořádat se se změnami, které balíčky v nových verzích získaly. Dle migračního návodu[14] jsem postupoval postupně od nižších verzí k vyšším.

Kvůli zastavení vývoje jsem odstranil balíček `flushbar`, který umožňoval zobrazovat na obrazovce krátké textové zprávy ve formě malé lišty u spodního okraje displeje. Dobrou náhradu ale začal mezitím poskytovat samotný Flutter v podobě widgetu `SnackBar`, kterým jsem `flushbar` nahradil. K odstranění došlo ještě u balíčku `ssh`, který byl v rámci prototypu použit pro stahování dat o studijních modulech.

4.2 Tvorba uživatelského rozhraní

Flutter je framework, který pro konstrukci uživatelského rozhraní staví na několika základních myšlenkách. První z nich je myšlenka, že „všechno je widget". Widget je objekt představující nějaký element z uživatelského rozhraní. Widgety ve Flutteru často obsahují další potomky, které pak můžou ve výsledku tvořit rozsáhlý strom widgetů. Koneckonců i samotná aplikace je widget, který je na začátku běhu Flutteru předán voláním funkce `runApp`.

Strom widgetů ovšem neslouží jako přímý podklad pro vykreslení uživatelského rozhraní. Flutter totiž uvnitř pracuje ještě se stromem objektů třídy *RenderObject*, které již přímo představují vykreslené elementy na obrazovce. Strom widgetů tak ve Flutteru funguje jako jakýsi předobraz toho, jak má rozhraní vypadat. [41] O sestavení *RenderObject* stromu se ale již stará samotný Flutter.

Vzhledem k deklarativní tvorbě rozhraní se může strom widgetů na základě různých událostí často konstruovat znovu a znovu, byť jen s malými změnami v uživatelském rozhraní. Flutter proto sleduje změny ve stromu widgetů a konkrétní *RenderObjecty* upravuje pouze v případě, kdy je to nutné, protože překreslování elementů na obrazovce je výkonnostně nezanedbatelná operace.

Programátor musí počítat s tím, že daný widget může být v různých chvílích instanciován znovu. Z tohoto důvodu existují ve Flutteru dva základní typy widgetů – `StatelessWidget` a `StatefulWidget`.

`StatelessWidget` je jednodušší z této dvojice. Všechny parametry potřebné k jeho vykreslení musejí být známy v momentě vytvoření jeho instance. Po opětovném vytvoření jeho instance by došlo ke ztrátě dříve uložených dat, a tak s ním musí být zacházeno tak, jako by jeho vnitřní proměnné byly neměnné. Použít `StatelessWidget` se tak hodí na prvky uživatelského rozhraní, které jsou neměnné a uživatel s nimi neinteraguje, jako například statické texty nebo ikony.

[14] `https://bloclibrary.dev/#/migration`

Aby bylo možné docílit dlouhodobějšího udržení vnitřních dat widgetu, Flutter přináší `StatefulWidget`. Ten narozdíl od `StatelessWidgetu` umožňuje vytvoření přidruženého objektu, který představuje nějaký jeho stav. Po opětovném vytvoření instance třídy `StatefulWidget` je sice vytvořen nový widget, ale pro jeho vykreslení je znovupoužit objekt zapouzdřující stav z předchozí instance. [42] `StatefulWidget` si tak zachovává stav napříč opětovným vykreslením, a proto je vhodný pro použití u widgetů, které se mění v závislosti na uživatelské interakci s nimi.

4.3 State management

V aplikacích s uživatelským rozhraním je běžné pracovat s nějakým stavem aplikace. Jak uživatel s rozhraním pracuje, očekává, že se bude měnit v reakci na ním vykonané operace. Flutter, jakožto deklarativně orientovaný framework se snaží i na stav aplikace nahlížet deklarativním pohledem. Konkrétně říká, že aktuální uživatelské rozhraní v daný moment je funkcí stavu aplikace. [43]

$$\text{UI} = f(state)$$

To znamená, že deklarativní styl konstrukce uživatelského rozhraní ve Flutteru by měl být dodržen i při konstrukci uživatelského rozhraní na základě stavu aplikace. Tedy že by se v kódu aplikace neměly příliš často objevovat imperativní konstrukce které představují nějakou přímou změnu uživatelského rozhraní (pro představu například `widget.setText()`). Místo toho programátor jasně definuje, jak se má na základě stavu sestavit strom widgetů. To, jak se mají změny uživatelského rozhraní realizovat, programátor už nechá na samotném Flutteru, který má své mechanismy, jak se se změnou v uživatelském rozhraní vypořádat.

4.3.1 Pomíjivý a sdílený stav

Flutter rozlišuje mezi dvěma typy stavu. [44] Prvním z nich je takzvaný *pomíjivý* (*ephemeral*) stav. Ten se vyznačuje tím, že je možné jej uchovat v jediném widgetu a nezávisejí na něm ostatní widgety. Příkladem *pomíjivého* stavu může být například text v textovém poli nebo informace, zda je tlačítko momentálně stisknuté. Na uchování takového stavu se hodí třída `StatefulWidget`.

4.3.2 Přístupy pro správu stavu

`StatefulWidget` ovšem není vhodné řešení pro uložení druhého typu stavu, kterým je takzvaný *aplikační* (popřípadě *sdílený*) stav. Takový stav se netýká pouze jednoho widgetu, ale může na něm záviset vykreslování uživatelského rozhraní v různých částech aplikace. V naivním řešení může mít programátor

nutkání jej ukládat do globálních proměnných. Typické příklady *aplikačního* stavu mohou být:

- přítomnost připojení k internetu,
- stav přihlášení uživatele,
- uživatelské nastavení aplikace.

Pro uložení sdíleného stavu je vhodné využít některého z pokročilejších *state-management* přístupů, než je samotný `StatefulWidget`. Takových přístupů existuje celá řada, a proto se v následujících odstavcích chci věnovat pouze dvěma nejdůležitějším.

Provider

Populární balíček s názvem `provider` poskytuje poměrně jednoduchý způsob, jak s aplikačním stavem pracovat. `Provider` respektuje strom widgetů a říká, že pokud je nějaký stav sdílený více widgety v rozdílných částech stromu widgetů, tak daný stav by měl být uložen v nejbližším společném předkovi těchto widgetů. Pokud bychom tedy například měli aplikaci s dvěma obrazovkami, které by obě potřebovaly pracovat se stejným sdíleným stavem, tento stav by měl být uložen ve widgetu, který má tyto dvě obrazovky jako své potomky ve stromu widgetů.

Takový přístup totiž umožňuje procházet strom widgetů od widgetu, který na daném stavu závisí až ke kořenu stromu widgetů, díky čemuž je možné poměrně rychle nalézt hledaný stav. V rámci tohoto společného rodiče může být vytvořen objekt, který bude držet požadovaný stav aplikace. Takový objekt by měl dědit od třídy `ChangeNotifier`, která umožňuje ostatním objektům poslouchat změny, které se ve stavu stanou.

Takový objekt zapouzdřující stav je do stromu widgetů zakomponován pomocí widgetu `ChangeNotifierProvider`. Ten si drží referenci na příslušný `ChangeNotifier` a zároveň umožnuje definovat podstrom widgetů, které jsou jeho potomkem.

Tento podstrom má pak snadný přístup k objektu zapouzdřující aplikační stav pomocí dalšího widgetu s názvem `Consumer`. `Consumer` při hledání požadovaného stavového objektu jednoduše prochází strom widgetů směrem ke kořenu až do momentu, než nalezne hledaný `ChangeNotifierProvider`. Jakmile jej nalezne, může na základě aktuálního stavu spustit překreslení jeho vlastního podstromu widgetů.

4.3.3 Business Logic Component

Prototyp od Alžbety Gogolákové již z části staví na návrhovém vzoru *Business Logic Component* (zkráceně BLoC). Ten umožňuje oddělení business logiky a správy stavů aplikace od prezentační vrstvy.

Základem činnosti BLoC komponent je příjem událostí z uživatelského rozhraní aplikace (např. „uživatel kliknul na tlačítko"), nebo událostí týkajících se například IO operací (např. „data z API byla stažena"). Příslušná BLoC komponenta má pak možnost na danou událost zareagovat například ve formě interakce s databází nebo webovými službami.

Na základě výsledku těchto interakcí se BLoC komponenta může rozhodnout, že změní svůj veřejně dostupný stav. Jádrem každé BLoC komponenty je metoda `mapEventToState`, jejímž úkolem, jak již název napovídá, je reagovat na přicházející události a s pomocí business logiky je mapovat na výsledné stavy. Právě na základě těchto stavů pak dochází v uživatelském rozhraní ke změnám. BLoC komponenta navenek poskytuje stavy ve formě *streamu*, který lze v prezentační vrstvě aplikace poslouchat a reagovat na jeho změny.

Ve výsledku má tedy prezentační vrstva pouze dvě odpovědnosti:

- poskytovat BLoC komponentě události informující o změnách v uživatelském rozhraní,
- reagovat na změny stavu BLoC komponenty.

Doporučovaný způsob, jak zpřístupnit BLoC komponentu ve stromové struktuře uživatelského rozhraní je pomocí třídy `BlocProvider`. Tato třída je zároveň widget, který je možné zakomponovat do stromu widgetů. Jejím úkolem je držet si referenci na instanci požadované BLoC komponenty. Dále pak obsahuje atribut `child`, do které může programátor přiřadit potomka v rámci stromové struktury widgetů. Díky třídě `BlocProvider` má pak celý tento podstrom přístup k požadované BLoC komponentě. Princip přístupu k BLoC komponentě je tedy vlastně velmi podobný přístupu ke stavovému objektu v případě balíčku `provider`.

Widget, který dokáže kdekoliv v tomto podstromu zpřístupnit danou BLoC komponentu je `BlocBuilder`. `BlocBuilder` při své konstrukci vyžaduje funkci `builder`, která je volána při každé změně stavu a na základě něj umožňuje sestavit nový strom widgetů.

4.3.4 Implementace s použitím BLoC patternu

Vzhledem k tomu, že BLoC pattern umožňuje implementovat i poměrně komplikovanější business logiku a aplikace již v omezené míře BLoC pattern používá, rozhodl jsem se pokračovat v jeho používání i nadále. Na správu *pomíjivého* (*ephemeral*) stavu jsem se rozhodl použít standardní nástroj Flutteru – `StatefulWidget`.

ConnectivityBloc

Základní BLoC komponentu, kterou aplikace využívá, je `ConnectivityBloc`, která na základě dostupných prostředků sleduje dostupnost připojení k internetu. Taková funkcionalita je potřebná k rozhodování o tom, zda má aplikace

zobrazovat data stažená ze serveru, nebo zda se musí pokusit zobrazit pouze lokálně uložená data. Tato BLoC komponenta je dále užitečná pro zobrazování informačních zpráv uživateli o stavu připojení k internetu.

Pro `ConnectivityBloc` jsem navrhl tři následující stavy:

- `ConnectedOnline` – zařízení je připojeno k síti a komunikace se serverem je v pořádku,
- `ConnectedOffline` – zařízení je připojeno k síti, ale komunikaci se serverem se nedaří realizovat (vyčerpaný datový limit FUP, Wi-Fi síť nemá přístup k internetu, ...),
- `Disconnected` – zařízení není připojeno k síti.

Oddělení dvou stavů, které signalizují nefunkční komunikaci se serverem (`ConnectedOffline` a `Disconnected`), je potřeba, aby se mohla `ConnectedBloc` komponenta pokoušet znovu navazovat spojení v případě `ConnectedOffline` stavu. Pro takové účely jsem dodatečně implementoval ve webové službě API endpoint `/ping`, který neposkytuje aplikaci žádná data, ale pouze aplikaci odpovídá HTTP kódem 200 v případě fungující komunikace mezi mobilní aplikací a webovou službou.

Pro jasnou detekci aktuálního stavu připojení `ConnectivityBloc` přijímá následující události:

- `Connected` – zařízení bylo připojeno k síti; optimisticky předpokládám, že jde o síť s přístupem k internetu,
- `Disconnected` – zařízení bylo odpojeno od sítě,
- `RequestFailed` – nastala chyba v rámci nějakého HTTP dotazu mezi aplikací a webovou službou,
- `RequestSuccessful` – HTTP dotaz mezi mobilní aplikací a webovou službou skončil úspěchem.

Pro zjišťování stavu připojení využívá komponenta `ConnectivityBloc` knihovnu `connectivity`, která umožňuje zjistit, zda je zařízení připojeno k mobilní síti, k Wi-Fi nebo připojeno není. Tato knihovna umožňuje kromě přímého dotazu na stav připojení k síti i možnost poslouchat tyto změny stavu. To umožní komponentě `ConnectivityBloc` okamžitě reagovat na změny v připojení.

NavigationBloc

Další BLoC komponentu představuje `NavigationBloc`. Jedná se o poměrně jednoduchý objekt, jehož zodpovědností je držet informaci o aktuální obrazovce v aplikaci. Rozlišuje mezi těmito obrazovkami:

- detail modulu se seznamem jeho lekcí,
- detail slovníku,
- seznam stažených videí,
- obrazovka s vyhledáváním,
- obrazovka se sekcí *Moje knihovna.*

Z těchto požadavků vyplývá nutnost vytvořit potřebné stavové objekty `ModulePage`, `GlossaryPage`, `DownloadsPage`, `SearchPage` a `MyLibraryPage` a k nim i příslušné objekty reprezentující události přesunu na danou obrazovku.

Jedinou netriviální funkcionalitou, kterou tato BLoC komponenta vykonává, je uložení si naposledy prohlíženého modulu. Uvažuji totiž, že v momentě, kdy by se uživatel nacházel na obrazovce s vyhledáváním nebo staženými videi a následně by stiskl navigační tlačítko zpět, intuitivně bude očekávat návrat na obrazovku s detailem modulu, ve které se aplikace nachází na začátku po zapnutí.

DataManagementBloc

Důležitou součástí aplikace je spolehlivá práce s daty v závislosti na dostupnosti připojení k internetu. Vzhledem k tomu, že na různých místech v aplikaci se musí s daty pracovat velmi podobně, jsem se rozhodl implementovat BLoC komponentu, která má na starosti získání dat, která mají být zobrazena uživateli.

V aplikaci totiž existují tři místa, na kterých je potřeba řešit stejný problém – načíst seznam objektů stejného typu buď z webové služby, nebo z lokálního úložiště, pokud je webová služba nedostupná. Tento úkol je potřeba řešit na následujících třech místech:

- levý vysouvací panel se seznamem modulů,
- obrazovka vybraného modulu se seznamem jeho lekcí,
- obrazovka vybrané lekce se seznamem jejích videí.

Rozhodnutí, z jakého zdroje mají být data načtena, musí `DataManagementBloc` učinit na základě aktuálního stavu komponenty `ConnectivityBloc`. Aby bylo možné použít `DataManagementBloc` na více místech pro různé modelové třídy a vyvarovat se tak duplikace kódu, jedná se o generickou třídu s parametrem specifikujícím třídu modelových objektů. Spolu s tímto parametrem vyžaduje `DataManagementBloc` ještě parametr pro specifikaci třídy, prostřednictvím které budou data získávána. Ta musí implementovat metody `getOnline` a `getOffline` z abstraktní třídy `DataManagementRepository`. `DataManagementBloc` tyto metody využívá právě k získání dat z online či offline zdroje.

Logika `DataManagementBloc` komponenty musí umět dále správně reagovat na změny v připojení. Například v momentě, kdy byla již načtena offline data z lokálního úložiště (pravděpodobně z důvodu absence připojení k internetu), a následně dojde k připojení k internetu, měla by se komponenta `DataManagementBloc` pokusit stáhnout aktuálnější data ze serveru. V opačném případě, kdy má již `DataManagementBloc` naopak načtená online data, a dojde k odpojení od internetu, by ale nemělo dojít k přepnutí zobrazení na data z offline úložiště.

Při zobrazování offline dat bez připojení k internetu je pro uživatele užitečné znát, které z objektů v seznamu je možné rozkliknout pro zobrazení detailu. Nedávalo by například smysl umožnit uživateli kliknout na modul, pro který neexistuje v lokálním úložišti uložena žádná lekce. Proto kromě získání samotných dat má `DataManagementBloc` ještě i zodpovědnost získat informace o objektech, které nemají v lokálním úložišti uložené žádné související objekty.

Pro použití ve stromu widgetů jsem definoval následující stavy, které pokrývají všechny potřebné situace:

- `Initial` – žádná data zatím nebyla načtena, uživatelské rozhraní by mělo zobrazovat načítací animaci,
- `LoadOnlineInitial` – došlo k rozhodnutí vyžádat si data ze serveru a momentálně probíhá jejich stažení,
- `LoadOfflineInitial` – došlo k rozhodnutí vyžádat si data lokálně a momentálně probíhá jejich získání,
- `LoadOnlineSuccess` – byla získána data z webové služby,
- `LoadOfflineSuccess` – byla získána data z lokální databáze.

`DataManagementBloc` přijímá následující typy vnějších událostí:

- `DataRequestedOnline` – data byla vyžádána ze serveru,
- `DataRequestedOffline` – data byla vyžádána z lokální databáze.

LoginBloc

Už v prototypu figurovala `LoginBloc` komponenta, která má za úkol reagovat na událost, když uživatel zadá na úvodní obrazovce své uživatelské údaje a klikne na tlačítko pro přihlášení. Úkolem `LoginBloc` komponenty je pak provést přihlášení a v rámci tohoto procesu měnit stav přihlašovací obrazovky, aby uživatele informovala o probíhajícím přihlašování a o jeho výsledku. Její stavy a události jsou triviální, proto se jim zde blíže nevěnuji.

AuthenticationBloc

Komponenta `AuthenticationBloc` poskytuje celé aplikaci stav o přihlášení uživatele. Stanovuje jej na základě přítomnosti JWT tokenu v bezpečném perzistentním úložišti v zařízení. Pro obsluhu tohoto úložiště používá třídu `UserRepository`, která byla pro tyto účely přítomna už v prototypu aplikace. Kromě správy stavu přihlášení má `AuthenticationBloc` na starosti ještě vymazání uživatelských dat při odhlášení uživatele. Z toho důvodu závisí kromě `UserRepository` i na dalších třídách zprostředkovávajících komunikaci s lokální databází. Komponenta `AuthenticationBloc` vystavuje následující stavy:

- `AuthenticationInitial` – počáteční neznámý stav,
- `AuthenticationInProgress` – signalizuje probíhající přihlášení nebo odhlášení,
- `AuthenticationNotAuthenticated` – stav informující o absenci JWT tokenu,
- `AuthenticationAuthenticated` – stav informující o přítomnosti JWT tokenu,
- `AuthenticationFailure` – stav informující o automatickém odhlášení z důvodu získání chybové odpovědi ze serveru (například při použití expirovaného *refresh* tokenu).

4.4 Videoslovník

Jednou z úprav v aplikaci, která v prototypu aplikace chyběla, bylo přidání *Videoslovníku*. Ten představuje velkou kolekci videí se slovíčky a slovními spojení ve znakovém jazyce a funguje jako doplňkový studijní materiál vedle standardních výukových modulů. *Videoslovník* není členěn na jednotlivé lekce, tvoří jej pouze seznam videí seřazený podle abecedy.

Z pohledu uživatelského rozhraní bylo potřeba vymyslet, jak strukturu rozdílnou od modulů v aplikaci prezentovat. Na jedné z online schůzek s Tichým světem jsme se poměrně rychle shodli na tom, že uživatel by se měl do *Videoslovníku* dostat po kliknutí na položku v levém vysouvacím panelu, stejně jako tomu je v případě modulů.

Dále však bylo nutné vyřešit, jak uživatelům umožnit snadný pohyb v rámci slovníku. Mobilní podobou *Videoslovníku* jsem se chtěl pokud možno přiblížit fyzickému slovníku, kde si člověk nejdříve nalistuje stránku s požadovaným počátečním písmenem a následně abecedním hledáním zpřesňuje svůj výběr.

Napadlo mě tedy použít podobný přístup, který se používá v mobilních aplikacích s telefonními kontakty, kde bývá na pravé straně displeje zobrazen seznam písmen abecedy, po němž uživatel posouvá prstem. Tímto způsobem

by se uživatel mohl ve slovníku dostat na slovíčka začínající na vybrané písmeno.

Vzhledem k obrovskému množství videí bylo potřeba myslet na to, aby zvolené řešenínebylo výkonově a datově náročné. Provedl jsem tedy testovací dotaz na kompletní seznam slovíček z *Videoslovníku* a zjistil jsem, že celá HTTP odpověď se všemi slovíčky má velikost 205 kB, což je podle mého názoru přijatelná velikost na jeden HTTP dotaz.

Pro vykreslení seznamu slovíček s bočním seznamem písmen abecedy jsem využil existující balíček `azlistview`, který přesně tuto funkcionalitu přináší. Widget `AzListView` ve svém konstruktoru přijímá pole s daty, na základě kterých bude seznam vykreslen, a také funkci `itemBuilder`, jejímž úkolem je přijmout prvek z pole jako vstup a na základě něj sestavit widget, který má být v seznamu vykreslen. `AzListView` se může tímto přístupem omezit na vykreslování pouze těch prvků, které jsou aktuálně viditelné, což optimalizuje jeho výkon.

4.5 Komunikace s API

Pro komunikaci pomocí HTTP obsahuje Flutter balíček `http`, který poskytuje programátorovi vysokoúrovňové API pro vytváření a odesílání HTTP dotazů. Pro základní použití poslouží naprosto v pořádku. V případě potřeby pokročilé funkcionality jako například možnosti cachingu HTTP odpovědí, manuální manipulace s HTTP hlavičkami a podobně už však není dostačující. Proto jsem v průběhu vývoje začal používat knihovnu z balíčku `dio`, která pokročilou funkcionalitu poskytuje. Tato nihovna pro práci s protokolem HTTP poskytuje stejnojmenou klientskou třídu `Dio`.

4.5.1 Klientské třídy

Pro komunikaci s webovou službou jsem v kódu vytvořil třídu `ApiClient`. Ta obsahuje jednotlivé metody pro každý endpoint webové služby. Pro snadnější ošetření problémů v komunikaci s webovou službou jsou chyby v komunikaci ošetřeny na úrovni samotné třídy `ApiClient`. V případě chyby týkající se autentizace (například expirace *refresh* tokenu) vyhodí třída výjimku typu `ApiRequestAuthenticationFailedException`. V případě chyby v komunikaci (například vypršení časového limitu HTTP dotazu) je vyhozena výjimka typu `ApiRequestCommunicationFailedException`. Ta má význam například pro komponentu `ConnectivityBloc`, která dokáže na základě této výjimky detekovat stav `ConnectedOffline`.

Knihovna `dio` poskytuje použití takzvaných *interceptor* funkcí. *Interceptor* je funkce, která je vykonána v návaznosti na různé události během komunikace, typicky před odesláním HTTP dotazu nebo ihned po přijetí odpovědi. Použití této funkce umožňuje manipulovat s HTTP komunikací nad rámec toho, jak

s HTTP klientem pracuje samotná aplikace. *Interceptor* funkce se registrují pro každého klienta třídy `Dio` zvlášť.

V mém případě používám *interceptor* funkci pro realizaci autentizace. Před tím, než je HTTP dotaz odeslán na server, prochází dotaz *interceptor* funkcí, ve které do dotazu připojuji HTTP hlavičku `Authorization` s uloženým JWT tokenem. Aplikace se tak nemusí o sestavování korektního dotazu starat, vše je řešeno na jednom místě.

Protože jsem tyto *interceptor* funkce potřeboval aplikovat pouze na HTTP dotazy, které získávají pomocí metody GET data se studijními materiály ze serveru, vytvořil jsem ještě druhou třídu `TokenClient` využívající interně jinou instanci třídy `Dio`, která nepoužívá dříve popsané *interceptor* třídy. Třída `TokenClient` je určena pro interakci s autentizačními endpointy webové služby. Pro ně totiž neplatí, že dotazy na ně mířící musejí obsahovat `Authorization` hlavičku, jak je tomu v případě fungování třídy `ApiClient`.

Další *interceptor* funkce zasahující do komunikace v třídě `ApiClient` je volána v momentě, kdy dojde k nějaké chybě v komunikaci. V kontextu knihovny `dio` může být chybou vypršení časového limitu pro dotaz nebo i navrácení chybového HTTP kódu. V této *interceptor* funkci mě konkrétně zajímá, zda nedošlo k vrácení HTTP kódu 401, a tedy odepření přístupu z důvodu chyby v autentizaci. Pokud takovou situaci detekuji, okamžitě pomocí instance třídy `TokenClient` vysílám na server dotaz s *refresh* tokenem pro vygenerování nového JWT tokenu.

V takovém případě je žádoucí, aby během obnovování JWT tokenu nedocházelo k dalším dotazům na serverové API. Proto je volání funkce pro obnovení JWT tokenu obaleno voláním funkcí `_dio.lock()` a `_dio.unlock()`, které se postaraji o to, aby byly všechny ostatní HTTP dotazy, které v této chvíli vzniknou, zařazeny do interní fronty HTTP klienta. Odeslání těchto dotazů na server bude vykonáno až po odemčení klienta.

4.5.2 HTTP cache

Vzdělávací materiály nejsou vzhledem k své povaze měněny příliš často. Proto se nabízelo využít HTTP cache pro umožnění cachování dat na straně klienta. Knihovna `dio`, kterou mobilní aplikace používá, sama od sebe ukládat HTTP odpovědi do cache neumí. Existuje však několik balíčků, které HTTP cache implementují pomocí `dio` *interceptor* funkcí.

Zvolil jsem konkrétně balíček `dio_cache_interceptor`[15], který jako jeden ze způsobů uložení odpovědí ze serveru nabízí i uložení dat v paměti. Existují možnosti použít i dlouhodobé perzistentní úložiště jako například SQLite, ale minimálně pro zrychlení reakcí aplikace v rámci jednoho uživatelského sezení je uložení v paměti dle mého názoru dobrý začátek. Velikost paměťové cache je výchozí konfigurací omezena na 7 MB, což považuji za přijatelné.

[15] `https://pub.dev/packages/dio_cache_interceptor`

```
// ošetření HTTP kódu 401 při získávání dat z API

try {
  // zamčení klienta pro získávání dat z API
  apiClient.lock();
  // obnovení JWT tokenu
  await tokenClient.refreshToken();
} on DioError catch (e) {
  // ošetření případných chyb při obnovení tokenu
  return handler.reject(e);
} finally {
  // odemčení klienta pro získávání dat z API
  dio.unlock();
}

// opětovné odeslání původního HTTP dotazu
// nyní s aktuálním JWT tokenem
final res = await _dio.fetch(e.response!.requestOptions)
handler.resolve(res);
```

Zdrojový kód 3: Automatické obnovení JWT tokenu při obdržení HTTP kódu 401

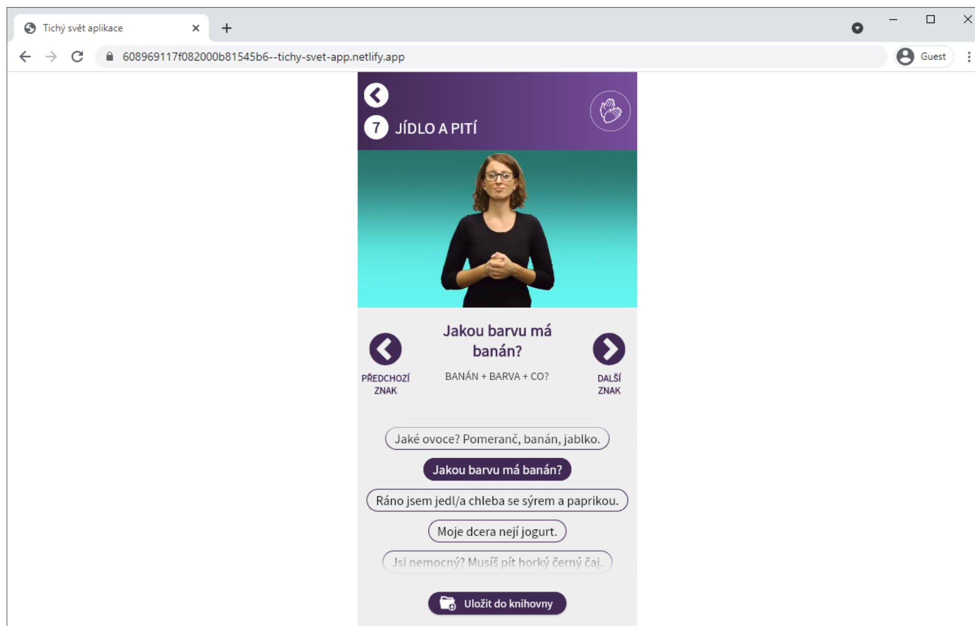
Pro signalizaci, že je na daný zdroj z REST API možné aplikovat cache, jsem využil HTTP hlavičku `Cache-control`. Parametr `max-age` jsem nastavil na hodnotu 1800 (tedy 30 minut), což je doba, která by měla v pořádku pokrýt trvání jednoho sezení v rámci mobilní aplikace. Použití HTTP cache jsem programově i manuálně otestoval a při opakovaných dotazech na stejná data aplikace získala okamžité reakce.

4.6 Nasazení aplikace na web

Během vývoje aplikace došlo k vydání frameworku Flutter ve verzi 2. Kromě řady dalších změn přinesla tato nová verze stabilní podporu pro možnost sestavit aplikaci pro spuštění ve webovém prohlížeči, která byla dosud ve fázi beta. Nově je tak díky této funkci možné sestavovat aplikaci pro opravdu široké spektrum zařízení na základě jednotného zdrojového kódu.

Sestavení pro web využívá možnosti kompilace Dart kódu do JavaScriptu. Výsledné sestavení aplikace pro web se tak skládá pouze ze statických souborů, které jsou webovým prohlížečem staženy a spuštěny.

Aplikace, která je předmětem této práce, je samozřejmě určena výhradně pro Android a iOS. Možnost spouštět aplikaci na webu mi ale vnuknulo myšlenku využít této příležitosti pro rychlejší distribuci testovacích verzí lidem



Obrázek 4.1: Nasazení mobilní aplikace na web

z vedení Tichého světa. Schvalovací procesy a instalace testovacích verzí skrz oficiální obchody s aplikacemi trvají vždy nějaký čas, a tak může být velmi výhodné využít právě cestu nasazování aplikace na web pro rychlé testování nějaké nově implementované funkce.

Rozšířit aplikaci původně vytvořenou pro běh na operačních systémech iOS a Android nebylo příliš složité. Dokumentace Flutteru obsahuje jasný návod na přidání podpory pro web. Tento proces přidá do struktury projektu složku `web`, která slouží jako základ webové aplikace. Obsahuje krátký soubor `index.html`, který se odkazuje na soubor `main.dart.js`, který vznikne sestavením Flutter projektu.

Aby se ale webová aplikace na webu zobrazovala co nejpodobněji jako na mobilním telefonu, zabalil jsem celou aplikaci ještě do elementu `iframe`, jemuž jsem nastavil rozměry tak, aby odpovídaly poměru stran displeje mobilního telefonu v orientaci na výšku.

4.6.1 Nekompatibilní závislosti

Při přidávání podpory pro web však bylo potřeba myslet na kompatibilitu balíčků s prostředím webu. Vzhledem k tomu, že spouštět aplikace na webu bylo experimentální vlastností Flutteru již od září 2019, [45] mnoho balíčků již bylo za tu dobu připraveno pro použití na webu. Zjistit, zda daný balíček web podporuje, je možné snadno z repozitáře balíčků Pub.dev. I přesto, že aplikace

používala poměrně populární a dobře udržované balíčky, nastala situace, že zkrátka některé balíčky web nepodporovaly.

Sqflite

Takovým případem byl například oblíbený balíček `sqflite`, který umožňuje práci s SQLite databází, která je přítomná na mobilních zařízeních a aplikace ji doposud využívala. Ve webovém prohlížeči ale k žádné formě relační databáze nemáme standardně přístup, ačkoliv určité pokusy pracovat s SQL v prostředí webu existovaly a v některých prohlížečích byly dokonce i implementovány.

Asi nejzásadnějším pokusem přinést SQL databázi do prostředí webu byl standard WebSQL. [46] Ačkoliv byl implementován v řadě prohlížečů včetně například Google Chrome, v roce 2010 byl označen jako zastaralý (*deprecated*). Hlavním kritikem této technologie byla organizace Mozilla, podle které není jazyk SQL vhodný pro klientské webové aplikace a která namísto Web SQL úspěšně prosadila úložiště IndexedDB postavené na perzistentních B-tree strukturách. [47, 48]

Kromě neúspěšně standardizované Web SQL existují dále ještě implementace jako například knihovna SQL.js, která umožňuje spustit SQLite na webu pomocí WebAssembly. WebAssembly je běhové prostředí široce rozšířené ve webových prohlížečích, které umožňuje běh kompilovaných programů s téměř nativním výkonem. [49] Vyznačuje se tak mnohem větším výkonem než JavaScript. SQL.js ale pracuje se souborem v paměti, a tudíž není perzistentní. Z těchto důvodů bylo rozhodnuto, že knihovna `sqflite` prozatím nebude mít podporu pro web. [50]

Moor

Proto jsem se pokusil ohlédnout po jiných knihovnách zprostředkovávajících perzistentní vrstvu Flutter aplikacím. V repozitáři Pub.dev lze najít balíček `moor`, který poskytuje API pro práci s SQLite databází a na mobilních zařízeních používá vnitřně právě knihovnu `sqflite`. Rozdíl oproti `sqflite` je v tom, že `moor` poskytuje mnohem vysokoúrovňovější API pro práci s daty. Dalším rozdílem je, že `moor` podporuje prostředí webu. Tam totiž používá právě výše zmíněnou knihovnu SQL.js. Problém s perzistencí SQL.js, která ve výchozím stavu používá pouze soubor v paměti, řeší `moor` tak, že tento soubor ukládá perzistentně pomocí LocalStorage, což je standardní prostředek pro uložení dat v prostředí webového prohlížeče.

Hive

Alternativou k relačním SQL databázím v kontextu jazyka Dart je databáze Hive. Jedná se o jednoduchou *in-memory key-value* databázi, která je implementována čistě v jazyce Dart. Díky tomu může Hive automaticky běžet

kdekoliv, kde jazyk Dart dokáže běžet. Hive má v ekosystému Dartu a Flutteru znatelnou popularitu.[16]

Na mobilních operačních systémech jsou data podobně jako u SQLite perzistentně uložena do souboru. V případě běhu ve webovém prohlížeči používá Hive databázi IndexedDB.

Na základě výše zmíněných vlastností má databáze Hive své výhody a nevýhody, které ji nedělají vhodnou pro některá použití. Kvůli tomu, že data ze souboru jsou po inicializaci načtena do paměti, zde automaticky mohou vznikat vyšší paměťové nároky u větších objemů dat. Díky uložení přímo v paměti má však Hive mnohem vyšší výkon při čtení než SQLite.

Vzhledem k jednoduchosti *key-value* struktury ale nemusí být databáze Hive vhodná pro případy, kdy je nutné provádět nějaké složitější dotazy, spojení a podobně. Na základě těchto vlastností se Hive hodí spíše pro menší objemy dat se strukturou, která je vhodná pro uložení dat pomocí *key-value* databáze – například seznamy dat s unikátními identifikátory.

Isar

Určitým kompromisem mezi klasickou relační SQLite a jednoduchou *key-value* databází může být nově vznikající databáze Isar[17], která podporuje mobilní zařízení i web. Na rozdíl od Hive nabízí i snadnější tvorbu vazeb mezi entitami, možnost tvorby komplexnějších dotazů a další. Tato databáze je ale zatím v alfa fázi vývoje, proto jsem ji nezvažoval pro použití v aplikaci Tichého jazyka. Pouze ji zde zmiňuji jako možnou alternativu pro možné použití v budoucnosti.

Volba úložiště

Nakonec jsem se rozhodl pro použití databáze Hive, jejíž *key-value* model vyhovuje požadavkům na uložení dat v mobilní aplikaci Tichý jazyk a její vysoká nezávislost na okolním prostředí dovoluje její použití na webu. Databáze Hive navíc umožňuje uložená data šifrovat, což je vlastnost, která dovoluje do databáze uložit i JWT a *refresh* tokeny.

4.6.2 Definice odlišných prostředí s .env

Podobně jako webové API postavené na Symfony i mobilní aplikace potřebuje být sestavována ve verzích s rozdílnou konfigurací.

Z důvodu, že se dlouhou dobu nedařilo od Tichého světa získat přístup k serveru pro nasazení webové služby, jsem potřeboval vyřešit jak sestavit aplikaci, která na API nebude závislá, abych mohl Tichému světu prezentovat průběžné výsledky stavu uživatelského rozhraní alespoň s testovacími daty.

[16] `https://pub.dev/packages/hive`
[17] https://github.com/isar/isar

Abych to mohl realizovat, potřeboval jsem vytvořit globálně dostupný parametr obsahující informaci o aktuálním prostředí. Dalším parametrem, který bylo potřeba konfigurovat zvlášť pro různá prostředí, byla adresa webového API.

Pro konfiguraci těchto parametrů jsem použil balíček `dotenv`, který stejně jako v případě Symfony aplikace načítá parametry ve formě *klíč – hodnota* ze souboru `.env`. Soubory pro všechna prostředí se v projektu nacházejí ve složce `dotenv`.

V této fázi vývoje však ještě nebyla nasazena Symfony aplikace s REST API, a tak bylo potřeba zabudovat testovací data do aplikace. Proto v případě, že proměnná prostředí `APP_ENV` obsahuje hodnotu „demo“ provádím v třídách `ApiClient` a `TokenClient` registraci dalších *interceptor* funkcí, které na základě požadované URL vracejí „falešná“ (*mock*) data ve stejném formátu jako reálné REST API. Pro přihlášení přijímá aplikace jakékoliv přihlašovací údaje jako validní.

Aplikace tak sice reálně nekomunikuje s webovou službou, ale všechno probíhá prakticky stejně, jako kdyby se službou komunikovala. Jediným rozdílem je, že zdrojem dat je místo webového API zmíněná *interceptor* funkce. To samozřejmě znemožňuje například funkcionalitu vyhledávání, pro zobrazení uživatelského rozhraní to ale není překážkou.

4.6.3 Nasazení testovacích verzí

Aby bylo možné sestavené webové verze distribuovat lidem z Tichého světa, bylo potřeba vyřešit nasazení těchto sestavení. Využil jsem službu Netlify, která poskytuje prostor pro hosting statických webových stránek, což je přesně tento případ.

Aby mělo každé nasazení svou vlastní URL, využil jsem funkci Deploy Previews, která před nasazením na produkční doménu provede nasazení na unikátně vygenerovanou doménu. Díky tomu mohlo v rámci vývoje vzniknout množství nasazení, která jsou dostupná současně, což přináší možnost rychle se vracet k starším verzím.[18]

Mírnou komplikací při nasazení na web bylo, že služba Netlify automaticky zakazuje poskytovat soubory začínající tečkou, což je problém pro soubor s konfigurací proměnných prostředí `.env`. Musel jsem toto omezení obejít tím, že v prostředí webu má tento soubor název `dotenv`.

Same Origin Policy je bezpečnostní mechanismus, který zjednodušeně řečeno nepovoluje ve webovém prohlížeči získávat data z jiného než aktuálního *originu*. Pod pojmem *origin* se rozumí trojice hodnot protokol, doména a port. [51] V prostředí mobilní aplikace sice mechanismus Same Origin Policy není aplikován, ale pro webovou verzi Flutter aplikace Tichého jazyka bylo potřeba vyřešit, jak budou data po nasazení webové služby získávána.

[18] Jedna z testovacích verzí je dostupná na adrese:
`https://608969117f082000b81545b6--tichy-svet-app.netlify.app/`

V rámci webové služby jsem tedy nainstaloval balíček `NelmioCorsBundle`, který přináší mechanismus Cross-Origin Resource Sharing, který povolí určitým *originům* přístup k datům webové služby. Konfigurace tohoto balíčku se provádí v souboru `config/packages/nelmio_cors.yaml`, kde jsem nastavil, že k datům mohou mít přístup webové aplikace s *originem* odpovídajícím regulárnímu výrazu

```
^https://[0-9a-z]+--tichy-svet-app\.netlify\.app$
```

tedy testovací aplikace nasazené na hostingu statických stránek Netlify, kam pomocí CI/CD nasazuji nová sestavení. Jak výsledné nasazení aplikace na web vypadalo, zobrazuje obrázek 4.1

4.7 Přechod na null safety

Jednou ze zásadních novinek nové verze Flutter 2.0 je příchod tzv. *null safety*. Jde o vlastnost jazyka, která dovoluje programátorovi specifikovat u proměnných, zda mohou nabývat hodnoty `null`. Tato vlastnost je mezi běžnými programovacími jazyky rozšířená zatím jen částečně, a tak se s ní jazyk Dart řadí po bok jazyků jako je Kotlin, TypeScript nebo `C#`. [52]

4.7.1 Výhody a změny v jazyce

Null safety zásadně zvyšuje sílu typové kontroly kompilátoru, a výrazně tím omezuje pravděpodobnost výskytu výjimek vznikajících při dereferencování `null` pointeru. [53] Tony Hoare, tvůrce jazyka ALGOL W, ve kterém se poprvé objevila možnost přiřadit do proměnné `null` ukazatel, tuto vlastnost jazyka zpětně označil za „billion dollar mistake". [54] Silnější typová kontrola programátora nutí zamýšlet se nad každou proměnnou, zda pro ni dává smysl, aby v rámci běhu programu nabývala `null` hodnoty, a nutí jej případné `null` hodnoty kontrolovat. Z chyb, které by se jindy projevily až za běhu programu, se stávají chyby, které se projeví už při kompilaci.

Zavedení této vlastnosti se v kódu Dart konkrétně projevuje mírnými změnami ve specifikaci typů. Všechny datové typy jsou od zavedení změn takzvaně *non-nullable*, tedy že do nich nelze přiřadit `null` hodnotu a jejich konkrétní hodnota musí být známa ihned při založení proměnné. Pro signalizaci, že daná proměnná může nabývat i `null` hodnot se používá otazník za názvem typu. Příklad *nullable* a *non-nullable* proměnných znázorňují následující řádky kódu:

```
String nonNullableString = "Hello world!";
nonNullableString = null;                   // error

String? nullableString = null;              // ok
```

Tato typová kontrola zohledňující null hodnoty se netýká jen proměnných, ale je aplikována i na argumenty funkcí a na jejich návratové hodnoty.

Speciálním klíčovým slovem, které se s příchodem *null safety* dostalo do jazyka, je modifikátor `late`. Ten umožňuje vytvořit proměnnou, která nemůže nabývat `null` hodnot, ale její inicializace neproběhne hned. Typickým příkladem použití jsou například proměnné v objektech představujících stav třídy `StatelessWidget`. Vnitřní proměnné těchto objektů totiž bývají často inicializovány až uvnitř funkce `initState`, která je volána po vytvoření stavového objektu.

```
String nonNullableString;                      // error

late String lateNonNullableString;             // ok
lateNonNullableString = "Hello world!";        // ok
lateNonNullableString = null;                  // error
```

4.7.2 Migrace projektu

Provedení migrace projektu na bezpečnou typovou kontrolu zohledňující `null` hodnoty záviselo především na připravenosti používaných balíčků. Zda je balíček na *null safety* připraven, lze snadno zjistit v repozitáři Pub.dev. Díky tomu, že zavedení této vlastnosti bylo avizováno již zhruba rok dopředu, drtivá většina balíčků, které projekt mobilní aplikace používal, již *null safety* podporovalo. Výjimkou byl balíček obsahující přehřávač YouTube videí `youtube_player_iframe`, ale nedlouho poté získal podporu *null safety* i on.

Po aktualizaci balíčků na verze podporující *null safety* již bylo možné provést samotnou migraci projektu. Pro co nejsnadnější migraci vznikl nástroj `dart migrate`, který se na základě kódu snaží odhadnout u všech typů, zda mohou `null` hodnot nabývat nebo ne. Přestože jeho výsledek nebyl stoprocentně správný a vzniklé změny bylo potřeba ručně zkontrolovat a místy upravit, migraci výrazně urychlil.

Kompilátor tedy v současném stavu projektu provádí kontrolu *nullable* typů. Mohu potvrdit, že jako programátor tuto vlastnost jazyka velmi vítám a jazyk Dart se díky ní pro mě stal velmi atraktivním jazykem, protože během vývoje umožňuje více se soustředit na samotný algoritmus místo na kontrolu `null` hodnot.

4.8 Změny v průběhu vývoje

Během vývoje probíhala průběžně komunikace s lidmi z vedení Tichého světa a konzultace provedených změn v aplikaci s použitím webových nasazení popsaných v kapitole 4.6 Nasazení aplikace na web. V rámci emailové komunikace a online schůzek vzniklo několik změnových požadavků týkajících se mobilní aplikace.

4.8.1 Odstranění podpory offline módu

Změnou, která zásadně zasáhla do směřování aplikace bylo rozhodnutí o odstranění offline módu z aplikace. Přestože byla tato funkcionalita od počátku Tichým světem schválena a došlo i na její implementaci, v pozdější fázi vývoje došlo k překvapivému rozhodnutí tuto funkcionalitu v aplikaci nakonec nezahrnout.

Jedním z hlavních argumentů pro jeho odstranění byly finanční důvody plynoucí z nutnosti zřídit dedikované úložiště s videi. Na webovém hostingu s e-learningovým portálem totiž nebylo možné videa uložit kvůli omezením na množství přenesených dat. Dalším argumentem Tichého světa pro odstranění offline módu byla i postupně klesající poptávka po offline funkcionalitě v mobilních aplikacích způsobená stále více rozšířeným používáním mobilních datových tarifů.

Z aplikace tak byla odstraněna velká část implementované funkcionality související s offline módem. Součástí aplikace jsem však ponechal komponentu zjišťující stav připojení k internetu `ConnectivityBloc` a s ní související prvky uživatelského rozhraní informující o stavu připojení.

4.8.2 Grafické úpravy

Od doby práce na prototypu mobilní aplikace před dvěma lety prošla grafická identita Tichého světa a Tichého jazyka určitou proměnou. Primární barvou kurzů Tichý jazyk se stala fialová barva, změny se dočkal i font a ikony.

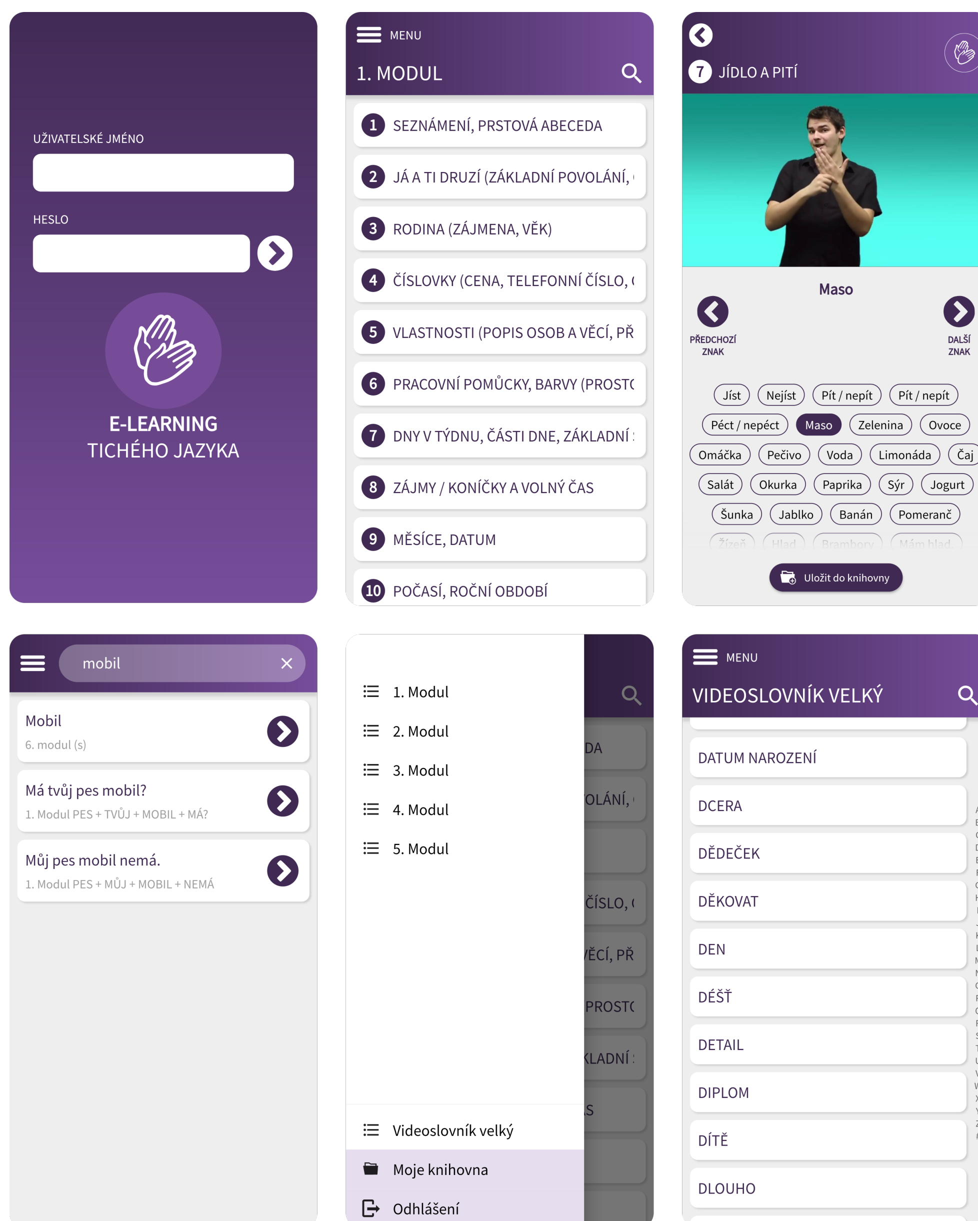
Proto jsem se spojil s grafikem Tichého světa, aby mi připravil grafický návrh pro úpravu designu aplikace, aby korespondovala s ostatními grafickými materiály Tichého jazyka.

Velkých změn se dočkala přihlašovací obrazovka, kde se místo barevného pozadí objevilo logo Tichého jazyka. Zásadní změnou prošla i obrazovka s detailem lekce, na které se seznam s názvy videí přesunul z vysouvací nabídky z horní lišty přímo do spodní poloviny obrazovky. Ostatní úpravy se týkaly hlavně typografie v seznamech lekcí, výměny ikonek a zaoblení rohů různých prvků uživatelského rozhraní.

Díky flexibilitě frameworku Flutter byla grafická úprava velmi dobře realizovatelná, takže nebylo nutné se od grafického návrhu jakkoliv odklánět. Výsledné screenshoty z aplikace jsou vyobrazeny na obrázku 4.2.

4.9 Testy

Flutter podporuje tři typy testů. Kromě klasických jednotkových testů umožňuje provádět i takzvané *widget testy*. Jejich cílem je otestovat izolovaně nějaký widget – tedy vytvořit jeho instanci, programově simulovat nějakou uživatelskou interakci a ověřit jeho očekávané chování.



Obrázek 4.2: Snímky obrazovek aplikace v novém grafickém designu

Třetím typem testů jsou integrační testy, pro které Flutter používá nástroj `flutter driver`. Tento typ testů spouští reálnou aplikaci buď na fyzickém zařízení nebo na emulátoru. V průběhu testů aplikace s nástrojem `flutter driver` komunikuje, což umožňuje nástroji provádět konkrétní scénáře a testovat celkové chování aplikace.

4.9.1 Jednotkové testy

Nejjednodušší formou testů, které ověřují kvalitu kódu aplikace jsou jednotkové testy. V projektu se nacházejí ve složce `test` a jsou rozděleny do podobné adresářové struktury jako samotný kód aplikace. Například v adresáři `test/blocs` je tak testována primárně funkcionalita tříd z `lib/blocs`.

V jednotkových testech využívám mockingu pro odstínění testovaného objektu od vnějších vlivů. Ačkoliv jsem původně používal pro vytváření *mock* objektů balíček `mockito`, s přechodem kódu na *null safety* jsem začal používat balíček `mocktail`, který se snaží poskytovat podobné API jako `mockito`, ale narozdíl od něj dokáže mnohem lépe vytvářet *mock* objekty v prostředí se zapnutými *null safety* kontrolami.

Kromě balíčku `mocktail` používám také balíček `http_mock_adapter`, prostřednictvím kterého vytvářím *mock* HTTP odpovědi, které HTTP klient z knihovny `dio` vrátí mnou vytvořeným třídám `ApiClient` a `TokenClient`. Použití balíčku `http_mock_adapter` lze vidět hlavně v testové třídě `test/http/clients_test.dart`.

Speciálním jednotkovým testům podléhají i třídy implementující BLoC pattern. K jejich testování existuje od tvůrce balíčků `bloc` a `flutter_bloc` ještě testovací balíček `bloc_test`. Ten přináší funkci `blocTest`, ve které lze pomocí několika parametrů definovat:

- funkci pro vytvoření BLoC objektu (parametr `build`),
- funkci, která s BLoC objektem může interagovat (parametr `act`)
- a seznam očekávaných stavových objektů, do kterých by měl BLoC objekt v průběhu přejít (parametr `expect`).

Kromě testování business logiky zapouzdřené do BLoC tříd pomocí jednotkových testů testuji ještě například správnost transformací JSON objektů z REST API do modelových tříd.

4.9.2 Widget testy

Speciálním typem testů, které Flutter umožňuje tvořit, jsou takzvané *widget testy*. Jejich cílem je izolovaně otestovat chování nějakého konkrétního widgetu, ze kterého je v aplikaci tvořeno uživatelské rozhraní. Tyto testy se spouštějí kompletně v prostředí jazyka Dart, není potřeba zřizovat žádné reálné zařízení nebo jeho emulátor. Testy tohoto typu se nacházejí v adresáři

test/widget. Věnuji se zde všem widgetům, které v aplikaci nabízejí nějakou formu interaktivity.

Tvorba *widget testů* je poměrně intuitivní. Veškerá interakce s testovanými widgety probíhá s použitím třídy WidgetTester. Nejdříve je potřeba v testovacím prostředí sestavit testovaný widget pomocí volání metody pumpWidget. Jakmile je instance widgetu vytvořena, programátor s ním může interagovat simulováním uživatelských akcí, například kliknutí, scrollování apod.

K ověření chování daného widgetu se používá funkce expect, která jako první argument přijímá instanci třídy Finder, druhým argumentem je predikát, který musí Finder objekt splňovat. Finder objekt slouží k vyhledání widgetů ve virtuální aplikaci, která byla voláním metody pumpWidget vytvořena. Finder může být použit například k vyhledání widgetu na základě textu, který obsahuje. Jednoduchý příklad *widget testu* přikládám v ukázce zdrojového kódu 4.

Jednotkové i widget testy jsou spouštěny v rámci Continuous Integration na každý commit, který se dostane do GitHub repozitáře. Během testů je měřena metrika *code coverage* a pro snadnější vizualizaci je odesílána do služby CodeCov[19]. V době odevzdání této práce se hodnota *code coverage* vyšplhala na 80 procent.

```
// sestav aplikaci s levým vysouvacím panelem AppDrawer
await tester.pumpWidget(MaterialApp(
  home: Scaffold(
    drawer: AppDrawer(),
    body: TitleAppBar("Hello world!"),
  ),
));

// ověř, ze text "Hello world!" je zobrazen
expect(find.text("Hello world!"), findsOneWidget);

// klikni na "hamburger" tlačítko pro vysunutí panelu
await tester.tap(find.byIcon(TichyJazykIcons.hamburger));

// překresli změnu v aplikaci
await tester.pump();

// ověř, ze levý vysouvací panel byl zobrazen
expect(find.byType(AppDrawer), findsOneWidget);
```

Zdrojový kód 4: Ukázka *widget testu*

[19] https://codecov.io/

4.9.3 Uživatelské testy

Pro testování práce uživatele s mobilní aplikací jsem sestavil sadu testovacích scénářů. Tyto scénáře bohužel nepokrývají veškerou funkcionalitu aplikace, protože v době realizace testů bohužel ještě Tichý svět neposkytl přístup k serveru pro nasazení webového REST API. Proto byly tyto testy realizovány v demonstračním prostředí s testovacími daty uloženými přímo v mobilní aplikaci. Testovací scénáře se týkaly práce s *Videoslovníkem*, procházení studijních materiálů a jejich uložení do sekce *Moje knihovna* a další práce s ní. Konkrétní znění testovacích scénářů je přiloženo ve formě přílohy B Testovací scénáře.

4.9.4 Flutter driver

Jedním z poměrně zajímavých nástrojů, kterým se Flutter pyšní, je nástroj `flutter driver`. Podle dokumentace se jedná o nástroj, který umožňuje na reálné aplikaci běžící v zařízení nebo v emulátoru spouštět integrační testy, které interagují s uživatelským rozhraním a testují chování aplikace.

Poměrně nepříjemným překvapením ale hned zpočátku bylo, že tato funkcionalita není připravena na vlastnost *null safety*, a balíčky `integration_test` a `flutter_driver` tak mají závislosti na starých verzích ostatních balíčků. Naštěstí bylo možné problém vyřešit přepnutím distribučního kanálu Flutter SDK ze *stable* na *dev*. V získané novější verzi Flutter SDK závisejí dva dříve zmíněné balíčky na aktuálních balíčcích, takže jejich instalace do projektu již nebyla problém. Flutter SDK z *dev* kanálu jsem použil pouze pro spouštění integračních testů s pomocí nástroje `flutter driver`. Pro vše ostatní jsem standardně využil *stable* kanál.

V rámci integračních testů poháněných nástrojem `flutter driver` jsem implementoval výše zmíněné testovací scénáře. Jejich implementace vyžaduje prakticky stejné znalosti jako pro tvorbu *widget testů*, jelikož `flutter driver` dokáže pracovat se stejným API, které poskytuje třída pro interakci s widgety `WidgetTester`.

Testy používající nástroj `flutter driver` bohužel nebyly implementovány v rámci testovacích skriptů v rámci Continuous Integration. Důvodem byla nutnost spouštět tyto testy na reálném zařízení nebo emulátoru. Problémem konkrétně bylo, že emulátor OS Android vyžaduje hardwarovou akceleraci, která je momentálně v rámci GitHub Actions dostupná pouze pro zařízení s macOS. [55] Emulátor iOS zařízení lze také spouštět pouze pod macOS.

Jedna minuta běhu CI/CD skriptu na macOS zařízení je ale v rámci GitHub Actions započítána jako deset minut běhu na OS Linux. Kvůli časové náročnosti běhu integračních testů jsem proto zůstal pouze u běhu těchto testů na lokálním zařízení. Na speciální vyhrazené infrastruktuře, která by nepodléhala limitům služby GitHub Actions a poskytovala by emulátoru hardwarovou

akceleraci, by bylo spouštění integračních testů s nástrojem `flutter driver` jistě realizovatelné.

Kapitola 5

Sestavení aplikace a nasazení

Poslední úkolem této práce bylo sestrojit CI/CD procesy, které automatizují sestavení a nasazení aplikace. Manuální sestavení a nahrání aplikace do obchodů s aplikacemi Google Play a AppStore může pokaždé zabrat nižší desítky minut. Navíc jde o proces, který je s vydáním každé nové verze nutné vykonat znovu. Proto může automatizace tohoto úkolu zásadním způsobem snížit odpracovaný čas a tím i náklady.

5.1 GitHub Actions

Jak již bylo řečeno v kapitole 4 Implementace aplikace, ke spouštění CI/CD skriptů používám službu GitHub Actions místo fakultní instance GitLab z důvodu nemožnosti spouštět skripty na macOS zařízení.

Základem GitHub Actions je takzvaný *job*. *Job* se skládá ze sekvence kroků, které jsou na konkrétním stroji následně vykonány. Na úrovni *jobu* je nutné specifikovat, zda poběží na stroji s OS Linux, Windows nebo macOS.

Každému kroku lze nastavit jeho název, takže při zobrazení záznamu běhu nějakého *jobu* lze velmi přehledně vidět, jaké kroky byly vykonány. Jeden krok může představovat obyčejné spuštění příkazu v příkazové řádce, nebo se může jednat o takzvanou *akci*, které jsou specifikem GitHub Actions.

Akce většinou představuje nějaký komplexní krok. Tvůrce skriptu může ve své sekvenci kroků používat veřejně dostupné *akce*, které za něj mohou vyřešit například stažení obsahu Git repozitáře, uložení výsledku skriptu ve formě artefaktu nebo třeba přípravu vývojového prostředí. V mých *jobech* používám například *akci* `subosito/flutter-action`, která je široce používána pro instalaci Flutter SDK na strojích ve službě GitHub Actions.

Jednotlivé *joby* jsou seskupovány do skupin, které se označují termínem *workflow*. Jeden *workflow* představuje právě jeden konfigurační soubor ve formátu YAML v adresáři `.github/workflows`. *Joby* v rámci stejného *workflow* na sobě mohou záviset a předávat si mezivýsledky. Závislost *jobů* je možné de-

finovat pomocí parametru `needs`. Jeho použitím tak vzniká v rámci *workflow* orientovaný graf na sebe závisejících *jobů*. Běhové prostředí GitHub Actions pak jednotlivé *joby* vykonává v pořadí jejich topologického uspořádání v rámci grafu.

Konfigurace jednoho *workflow* může například specifikovat, při jakých událostech má být spouštěn a kterých větví se má týkat. Pro každé prostředí, do kterého bude aplikace nasazena, jsem tedy vytvořil vlastní *workflow*, který vykonává požadované úkony pro dané prostředí. Každá taková větev má pro sebe vytvořen vlastní *workflow*. V adresáři `.github/workflows` je tedy možné nalézt několik souborů, které obsahují skripty pro dané prostředí.

5.1.1 Optimalizace doby běhu

GitHub Actions pro studenty nabízejí limit 3000 minut běhu skriptů pro jejich projekty měsíčně. Je však nutno dodat, že tento limit je takto štědrý pouze pro stroje s OS Linux. Za každou minutu běhu na OS Windows jsou totiž z limitu odečteny minuty dvě, u zařízení s macOS jde dokonce o deset minut. Proto je při spouštění CI/CD skriptů dobré myslet na optimalizaci a pokusit se dobu běhu snížit.

Pro snížení doby běhů využívám *akci* `actions/cache`, která dokáže uložit obsah nějakého adresáře pro použití v příštích bězích. Obsah cache je pak přístupný různým strojům v rámci stejného *workflow*. Při optimalizaci doby běhu jsem se zaměřil na kroky, které zabírají nejvíce času. Šlo zejména o kroky, které připravují prostředí pro sestavení, instalují závislosti projektů a provádějí samotné sestavení aplikace. Bohužel, z povahy věci není možné sestavení aplikace příliš urychlit. Proto jsem se soustředil na optimalizaci přípravy prostředí a instalaci závislostí.

S pomocí *akce* `actions/cache` jsem začal do cache ukládat celý adresář s Flutter SDK. Jeho standardní příprava trvala původně zhruba minutu, s použitím cache se celkový čas (včetně uložení a obnovy cache) snížil na zhruba patnáct sekund. Do tohoto adresáře si Flutter zároveň ukládá i balíčky z repozitáře Pub.dev. Použitím cache tak došlo i ke zkrácení doby běhu příkazu `pub get` o pár desítek sekund.

Druhou optimalizací, kterou jsem provedl, bylo uložení do cache balíčků Ruby, které jsou na daný stroj instalovány pro získání nástroje Fastlane. Instalaci Ruby je možné v GitHub Actions provést pomocí *akce* `ruby/setup-ruby`. Tato *akce* zároveň dokáže automaticky nainstalovat potřebné balíčky pomocí nástroje Bundler a uložit je pro další běhy skriptu do cache. Díky použití cache opět došlo ke zkrácení doby běhu této *akce* z necelé minuty na několik málo vteřin, což je konkrétně v případě macOS zařízení značná úspora.

5.2 Automatizace sestavení a nasazení

Aplikaci je možné standardně distribuovat do obchodů s aplikacemi Google Play a AppStore pomocí jejich standardních webových rozhraní. Google Play používá webovou aplikaci Google Play Console, AppStore používá AppStore Connect. Úkolem této práce je však prozkoumat možnosti automatizace nasazení a propojení se systémem pro správu verzí.

5.2.1 Fastlane

Pro tyto účely existuje open source nástroj Fastlane, který se právě o automatizaci procesu sestavení a nasazení mobilních aplikací snaží. Nástroj Fastlane dokáže komunikovat se službami AppStore Connect a Google Play Console pomocí jejich API. [56] Díky tomu dokáže automatizovat i úkony potřebné pro nasazení aplikace do obchodů s aplikacemi. Nástroj je distribuován ve formě aplikace s rozhraním v příkazové řádce a pro svůj běh vyžaduje Ruby ve verzi 2.4 – 2.7.

Základem pro použití Fastlane je soubor `Fastfile`. Ten v sobě s použitím syntaxe jazyka Ruby definuje jednotlivé procedury (Fastlane používá pojem „lanes"), které mají být provedeny. Každá procedura se skládá ze sekvence úkonů, které mohou vykonávat například vytvoření screenshotů aplikace, její sestavení nebo nahrání do AppStore Connect nebo Google Play Console.

Tyto úkony mají podobu příkazů s možností specifikování dodatečných parametrů. Skrze příkazy Fastlane interně volá externí nástroje jako například Gradle nebo Xcode pro sestavení aplikací. Příkazy standardně pocházejí ze samotného Fastlane, který jich nabízí opravdu široké množství.

Ukázková procedura skládající se ze čtyř příkazů, která sestaví aplikaci pro iOS, zajistí její podepsání a výsledek nahraje do služby AppStore TestFlight, může vypadat například takto:

```
lane :beta do
  increment_build_number(
    build_number: app_store_build_number(live: false) + 1,
  )
  sync_code_signing(type: "appstore", readonly: is_ci)
  build_app()
  upload_to_testflight()
end
```

Zdrojový kód 5: Ukázka procedury pro sestavení iOS aplikace a jejího odeslání do AppStore TestFlight v souboru `Fastfile`

Jelikož Flutter z jednoho kódu v jazyce Dart generuje dva separátní projekty představující standardní Android a iOS aplikace, je potřeba pro každý

z těchto dvou projektů vytvořit vlastní `Fastfile` soubor. Výsledná konfigurace nástroje Fastlane se nachází v adresářích `android/fastlane` a `ios/fastlane`.

Pro možnost uzamčení konkrétní verze je doporučeno instalovat nástroj Fastlane pomocí nástroje Bundler, což je správce balíčků jazyka Ruby. Adresáře s Android a iOS projekty tak dále obsahují i soubory `Gemfile` a `Gemfile.lock` se seznamem Ruby závislostí a jejich konkrétních verzí. Pro instalaci nástroje Fastlane je tak potřeba pouze v daném adresáři spustit příkaz `bundle install`.

5.2.2 iOS

AppStore Connect

Společnost Apple nabízí pro správu publikování mobilních aplikací v AppStore službu AppStore Connect. Prostředí dále nabízí správu propagačních materiálů a metadat týkajících se aplikace, možnost podání žádosti o její schválení a také statistiky týkající se počtů stažení, když už je aplikace nasazena v produkci. AppStore Connect dále umožňuje správu uživatelů, kteří mají přístup k vývojářskému účtu. Tímto způsobem mi byl udělen přístup a práva pro editaci aplikace Tichý jazyk.

Kromě veřejného publikování aplikace nabízí AppStore Connect ještě službu TestFlight, která je určena k zpřístupnění testovací verze aplikace vybraným uživatelům. Právě tuto službu jsem se rozhodl využít pro dodávání neprodukčních verzí aplikace lidem z vedení Tichého světa.

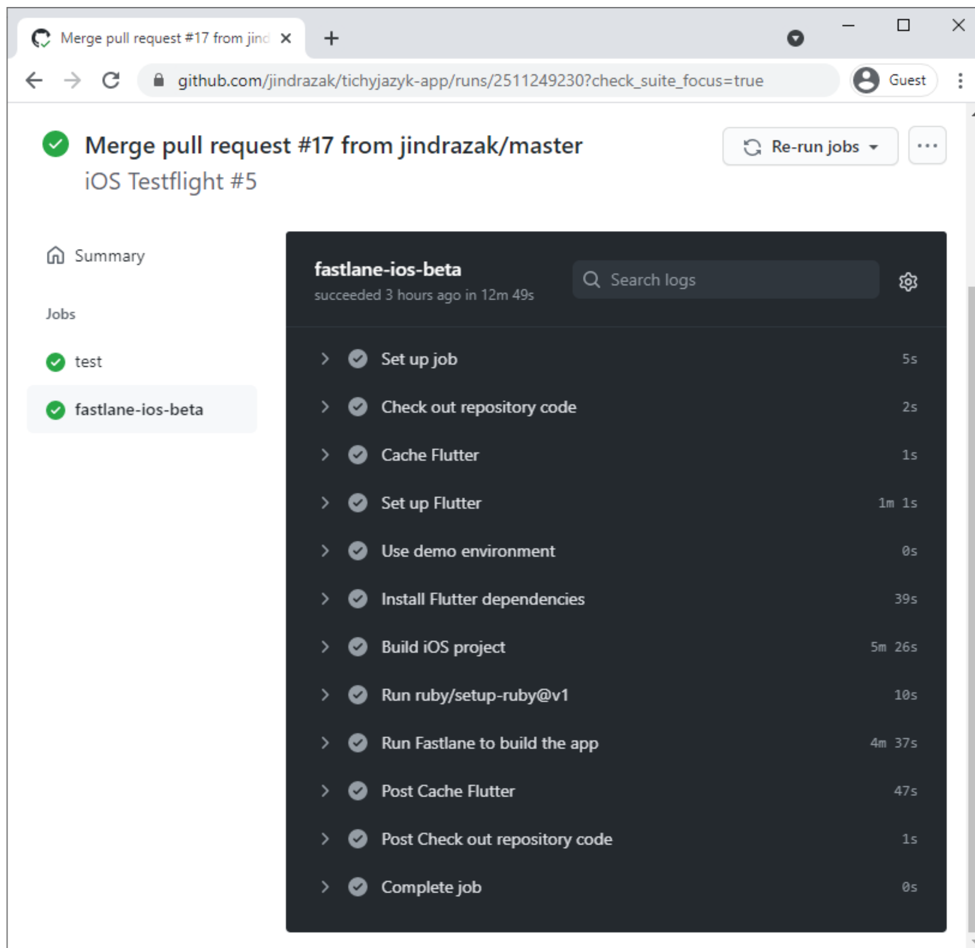
Věc, kterou se AppStore odlišuje od ostatních obchodů s aplikacemi, je celkový důraz na kontrolu aplikací a jejich bezpečnost pro uživatele. Každá verze aplikace musí být před jejím publikováním ze strany Apple důkladně kontrolována, zda splňuje podmínky publikování. Tyto kontroly jsou nutné i v případě služby TestFlight.

Tvorba fastlane procedury

V adresáři `ios`, který slouží jako kostra iOS projektu jsem vytvořil soubor `Fastfile`, který obsahuje procedury pro sestavení iOS aplikace. Spolu s ním zde nástroj Fastlane vytvořil ještě další soubory s metadaty týkajícími se aplikace.

Prvním krokem při sestavení Flutter aplikace pro systém iOS je spuštění příkazu `flutter build`, který z Flutter projektu sestaví standardní Apple iOS projekt, který musí být do finální aplikace sestaven nástrojem Xcode.

Kvůli tomu, že iOS aplikace lze sestavit pouze na počítači s operačním systémem macOS, který nevlastním, zřídil jsem si na lokálním počítači macOS ve virtuálním stroji, abych mohl skripty pro sestavení a publikování aplikace vyvíjet. Z důvodu nižšího limitu pro běh CI/CD skriptů na macOS zařízeních ve službě GitHub Actions jsem se snažil minimalizovat množství běhů Github Actions na stroji s macOS na co nejnižší číslo, aby nedošlo k vyčerpání limitu.



Obrázek 5.1: Průběh GitHub Actions *jobu* sestavujícího iOS aplikaci s nahráním do AppStore Connect

Podepisování aplikací

Pro samotné sestavení a distribuci iOS aplikace je potřeba zařídit podepsání aplikace. K tomu je potřeba vytvořit certifikát a takzvaný *provisioning profil*. Existuje několik typů *provisioning profilu*. Ten může být buď typu *development*, který slouží k instalaci aplikace na testovací zařízení, nebo typu *distribution*, který umožňuje navíc aplikaci veřejně publikovat. *Provisioning profil* pro distribuci musí být navíc vázán na jednoznačné App ID. App ID je unikátní řetězec identifikující aplikaci. V tomto případě jsem použil řetězec `cz.cvut.fit.tichyjazyk`. Druhou věcí, která je k podepsání aplikace potřeba, je certifikát pro ověření totožnosti, kdo aplikaci podepsal. [57]

Manuální řešení podepisování aplikací vyžaduje vykonání řady kroků při

vytváření certifikátu a *provisioning profilu*. Při práci v týmu to často může znamenat vytvoření velkého množství *profilů*, což může vést na jejich složitou správu ve webovém rozhraní AppStore Connect.

Fastlane proto přichází s řešením v podobě nástroje Match, který si bere zodpovědnost práce s certifikáty a *provisioning profily* na sebe. Jejich vytvoření obstarává prostřednictvím AppStore Connect API. Potřebnou dvojici certifikátu i *provisioning profilu* vytváří pouze jednou a pro možnost jejich sdílení v týmu je ukládá v separátním Git repozitáři. V případě jejich expirace automaticky provede opětovné vytvoření, takže odpadá nutnost manuální správy v rozhraní AppStore Connect. Vytvořil jsem tedy privátní Git repozitář, do kterého nástroj Match certifikáty a *provisioning profily* ukládá.

Po stažení certifikátu a *provisioning profilu* z Git repozitáře již může dojít k samotnému sestavení aplikačního balíčku včetně jeho podepsání a nahrání do AppStore Connect. V případě Fastlane lze toto realizovat pomocí volání dvojice příkazů `build_app()` a `upload_to_app_store()`. Výsledný běh skriptu ve Službě GitHub Actions je vyobrazen na obrázku 5.1

5.2.3 Android

Google Play Console je webové administrační prostředí pro správu vydání aplikací na Google Play. Principy práce s AppStore Connect a Google Play Console jsou vzhledem k jejich stejnému určení podobné.

Kromě distribuce aplikace prostřednictvím Google Play umožňuje webové rozhraní vývojářům uskutečnit testování aplikace pro omezené skupiny testerů. Google Play Console nabízí několik prostředí (Google Play používá termín „track“), v rámci kterých lze aplikaci vydat. Prvním je tzv. *internal track*, které je určeno pro snadné a rychlé sdílení sestavení aplikace v rámci dané organizace. Vydání aplikace v rámci tohoto prostředí nevyžaduje kontrolu aplikace ze strany firmy Google a není pro něj nutné vyplňovat ani základní údaje jako je název aplikace nebo její ikonka.

Uzavřené (alpha) a otevřené (beta) prostředí jsou už určena k distribuci aplikace i těm uživatelům, kteří nejsou členy dané organizace. Proto tato prostředí již vyžadují, aby byla každá verze schválena ze strany společnosti Google. Aplikace publikovaná v otevřeném beta prostředí je na Google Play veřejně viditelná a uživatelé se do ní mohou volně registrovat (pokud není naplněn maximální limit testerů). Alpha prostředí veřejné není a přístup k němu mají pouze pozvaní testeři.

V rámci automatizace buildů aplikace Tichý jazyk jsem na začátku využil právě interní prostředí, protože umožňuje rychlou distribuci nových sestavení aplikace, které obzvlášť v brzkých fázích vývoje před vydáním bývá mnoho.

Google Play při nahrání každého sestavení vyžaduje, aby dané sestavení obsahovalo číslo sestavení (*build number*), které je vždy větší než číslo sestavení předchozích sestavení. Aplikace je dále označena verzí, která je definována centrálně v souboru `pubspec.yaml`. Tu je nutné zvyšovat pouze v případě nového

vydání aplikace, a to jak pro produkční vydání, tak pro uzavřené i otevřené testování.

Standardní sestavení aplikace pro Android má na starost nástroj Gradle. Jeho vstupem je Android projekt, který má standardní strukturu. Podobně jako u sestavení aplikace pro iOS, vše začíná sestavením samotného Android projektu příkazem `flutter build`.

Pro sestavení výsledné aplikace existují dvě možnosti. První možností je sestavení APK souboru, který funguje jako instalační soubor na konkrétní zařízení a je tedy vždy určen pro jednu konkrétní procesorovou architekturu. Pro sestavení aplikace pro všechny dostupné architektury pro OS Android by to znamenalo sestavení několika APK souborů.

Novější a zároveň doporučovanou možností je sestavit AAB (Android App Bundle) soubor, který v sobě zapouzdřuje všechny architektury. Ve výsledku pak tedy stačí nahrát do Google Play Console pouze tento soubor a Google už se následně postará o jeho zpracování a distribuci správné verze aplikace pro správné zařízení.

Aby bylo zaručeno, že aplikace pochází opravdu od daného vývojáře a že nebyla nijak změněna, i se zde řeší podepisování aplikace. Google se snaží tento proces pro vývojáře co nejvíce usnadnit, a proto nabízí podepisování automaticky na jeho straně. Jediným požadavkem na straně vývojáře je aplikaci podepsat svým vlastním takzvaným *upload klíčem*, který slouží pouze pro ověření, že aplikaci nahrál do Google Play Console opravdu daný vývojář. Při jeho ztrátě je možné zažádat o použití nového *upload klíče*. Po nahrání aplikace ji Google podepíše znovu sám – právě pro účely distribuce mezi uživatele. Ponecháním podepisování aplikace na Googlu mizí riziko toho, že by mohlo dojít ke ztrátě hlavního klíče, a tedy ztráty možnosti aplikaci aktualizovat. [58]

K podepsání pomocí *upload klíče* je nutné vygenerovat tzv. *keystore* – soubor obsahující zašifrované klíče. Nástroj Gradle jej využívá právě při podepisování aplikací a k jeho použití potřebuje tři údaje:

- heslo samotného *keystore*,
- alias klíče v *keystore*,
- heslo klíče *keystore*.

Výsledkem práce nástroje Gradle je tedy AAB soubor, který lze nahrát do Google Play Console.

Aby se proces sestavení i nahrání aplikace do Google Play Console zautomatizoval, používám opět nástroj Fastlane. V adresáři `android` jsem tedy nástroj Fastlane inicializoval, a došlo tak k vytvoření souboru `Fastfile` a několika dalších souborů s informacemi o aplikaci, podobně jako v případě iOS verze.

Pro sestavení a nahrání do Google Play Console vznikla Fastlane procedura s názvem „internal“. Ta začíná spuštěním příkazu `flutter build`. Ačkoliv by

se mohlo volání tohoto příkazu vyskytovat mimo `Fastfile` například jako jeden z kroků GitHub Actions jobu, umístil jsem jej sem. Nástroj Fastlane totiž obsahuje užitečný příkaz `number_of_commits`, který vrátí počet commitů v daném repozitáři. To mi umožňuje získat unikátní a postupně rostoucí hodnotu, kterou používám jako číslo sestavení dané verze aplikace. Kromě nastavení čísla sestavení používám přepínače `--obfuscate` a `--split-debug-info` pro odstranění co nejvíce informací, které by mohly pomoci škodolibému analytikovi při reverzní analýze aplikace.

Druhým krokem je volání funkce `upload_to_play_store()` pro nahrání artefaktu do Google Play Console. Tato funkce je volána s několika parametry, které určují cestu k sestavenému AAB souboru, cílové prostředí daného artefaktu, a také parametr s názvem `json_key_data`.

Google totiž pro komunikaci s Google Play Console API vyžaduje zřízení takzvaného *Service účtu*. To umožňuje jasně oddělit jeho oprávnění od ostatních uživatelských účtů, které mají do Google Play Console přístup.

Jeho zřízení je mírně složitější proces, který může navíc absolvovat pouze vlastník vývojářského účtu. Spočívá ve vytvoření tohoto účtu, přidělení správné role a stažení souboru s přihlašovacími údaji ve formátu JSON. Celý proces navíc probíhá ve webovém rozhraní Google Cloud Console, která může být pro uživatele Google Play Console kompletně nová. Proto jsem sepsal podrobný návod pro administrátora vývojářského účtu Tichého světa, aby proběhlo vytvoření *Service účtu* bez problémů. Výstupem bylo získání právě popisovaného JSON souboru, který Fastlane využívá pro komunikaci s Google Play Console API.

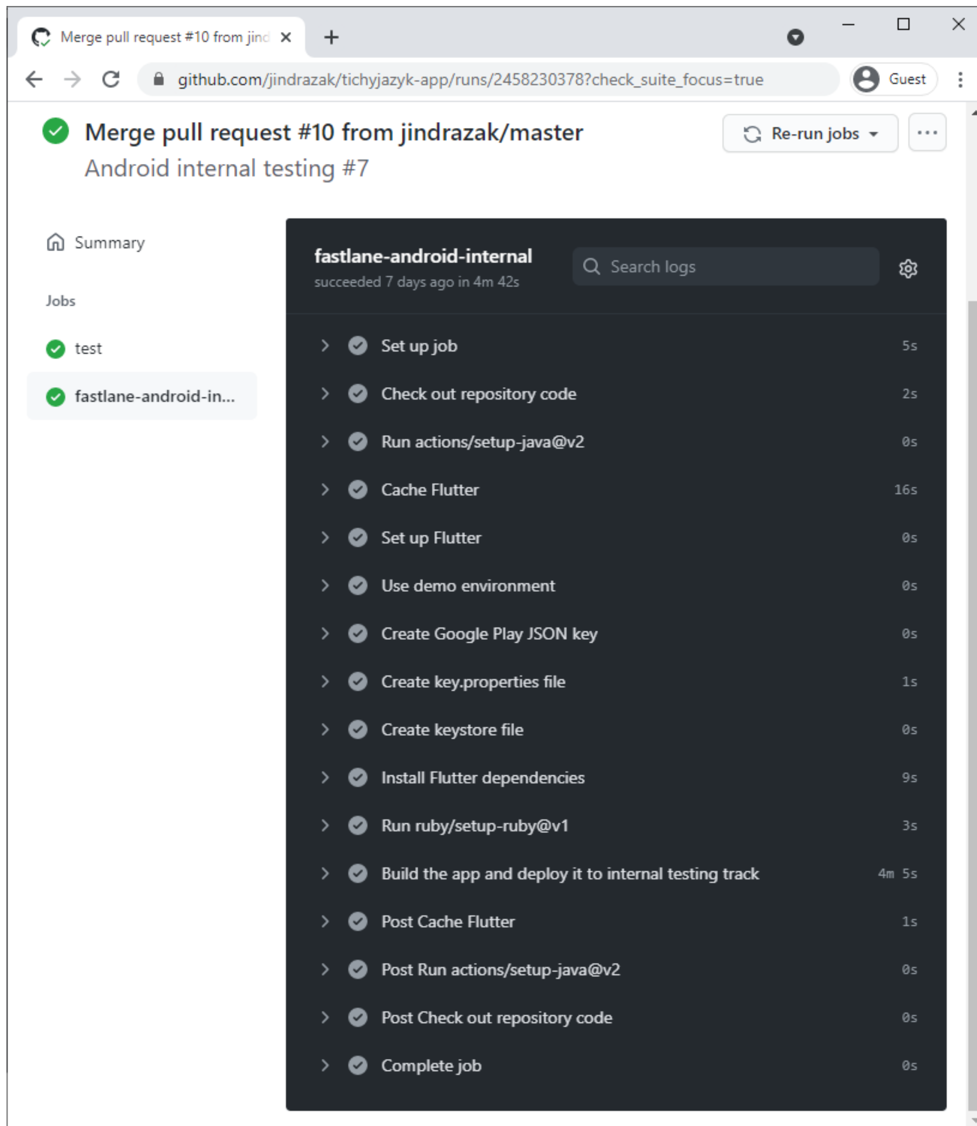
Vytvořená Fastlane procedura je spouštěna v rámci GitHub Actions workflow, jejíž průběh je zachycen na obrázku 5.2

5.2.4 Skrytí tajemství v CI/CD systému

Oba popsané procesy pracují s řadou důvěrných informací, hesel a souborů. V rámci lokálního běhu sestavení aplikace to není problém, ale pro použití v nějakém CI/CD systému, ke kterému mohou mít potenciálně přístup další vývojáři z týmu, je vhodné hesla a tajné soubory nezahrnout do repozitáře. Místo toho je dobrou praxí tato tajemství uložit v rámci tajných hodnot, což je běžná funkce CI/CD systémů.

V mém případě jsem tedy v administraci projektu na GitHubu vytvořil následující tajné parametry týkající se Google Play:

- `ANDROID_KEYSTORE` – obsah *keystore* souboru zakódovaný v base64,
- `ANDROID_KEYSTORE_PASSWORD` – heslo *keystore* souboru,
- `ANDROID_KEYSTORE_KEY_ALIAS` – název *upload klíče*,
- `ANDROID_KEYSTORE_KEY_PASSWORD` – heslo *upload klíče*,



Obrázek 5.2: Průběh GitHub Actions *jobu* sestavujícího Android aplikaci s nahráním do Google Play Console

- `GOOGLE_PLAY_JSON_KEY` – obsah JSON klíče pro přístup k *Service účtu.*

Tajné parametry pro sestavení a distribuci iOS aplikace jsou následující:

- `FASTLANE_PASSWORD` – heslo k Apple Developer účtu,
- `FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD` – heslo potřebné pro autorizaci k nahrání aplikace do AppStore Connect,
- `MATCH_GIT_BASIC_AUTHORIZATION` – řetězec ve formátu *HTTP Basic Authentication* pro přihlášení ke Git repozitáři s certifikátem a *provisioning profilem* prostřednictvím nástroje Match,
- `MATCH_PASSWORD` – heslo, kterým jsou šifrovány certifikáty a *provisioning profily* v Git repozitáři.

Na tyto parametry se v GitHub Actions *workflow* odkazuji na místech, kde jsou potřeba a používám je jako proměnné prostředí. Díky tomu zůstávají všechny důležité údaje tajné a k projektu tak mohou bez problémů přistupovat i ostatní vývojáři. Tato konfigurace by byla obzvlášť důležitá, kdyby se jednalo o open source projekt.

Celý proces automatizace sestavení a distribuce mobilní aplikace se ukázal jako nezanedbatelná časová investice. Dosažení automatizace totiž vyžaduje seznámení se s nástrojem Fastlane, zajištění infrastruktury na spouštění skriptů, tvorbu samotných Fastlane procedur a jejich propojení s obchody s aplikacemi. Přesto se však jedná o investici, která se s počtem nových vydání aplikace může poměrně rychle vrátit.

5.3 Monitoring chyb

Pro zajištění úspěšnosti mobilní aplikace je nutné sledovat a analyzovat její kvalitu. Jednou z nejjednodušších metrik, kterou je dobré v průběhu času sledovat, je průměrné uživatelské hodnocení v obchodech s aplikacemi. Tato metrika může posloužit jako jednoduchý nástroj pro sledování vývoje kvality napříč jednotlivými verzemi aplikace. Uživatelské recenze aplikace mohou být navíc velmi důležitým vodítkem při hledání problémů, které aplikace má, ale v rámci vývoje a testování nebyly odhaleny.

Průměrné skóre uživatelských hodnocení má zásadní vliv na umístění v žebříčcích a ve vyhledávání v obchodech s aplikacemi. [59] Hodnocení bývají ovlivněna faktory jako je kvalita uživatelského rozhraní, rychlost aplikace nebo její funkčnost. Kromě uživatelských hodnocení má ale na dobrém umístění v obchodu s aplikacemi vliv stabilita aplikace, [60] jejíž měření se v této kapitole zabývám.

Aby byla implementace aplikace stabilní, je potřeba během vývoje dbát na kvalitní pokrytí testů, a to jak jednotkových, integračních, tak i uživatelských.

Ani to však neodstraní riziko pádu aplikace při produkčním nasazení. Aby bylo možné zjistit, jak dobře aplikace funguje na produkčním prostředí, je potřeba mít schopnost chyby zaznamenávat a analyzovat.

5.3.1 Mobilní aplikace

Schopnost zaznamenávat chybové incidenty je poměrně samozřejmá v případě běhu aplikace na serveru, na kterém je možný přístup k aplikačním logům. Mobilní aplikace, která může běžet na obrovském počtu zařízení, však vyžaduje složitější řešení.

Existuje několik služeb, které nabízejí řešení tohoto problému jako například Instabug, Sentry a další. Službu, kterou jsem se rozhodl v aplikaci Tichého jazyka použít, je Crashlytics. Crashlytics je webová služba na sběr a analýzu chyb v rámci běhu aplikace. Původně se jednalo o separátní službu, po její akvizici firmou Google je od roku 2017 součástí systému Firebase, který zahrnuje řadu nástrojů pro analytiku a podporu běhu mobilních a webových aplikací bez nutnosti implementovat *backendový* systém.

Firebase poskytuje velmi dobrou podporu pro Flutter projekty prostřednictvím projektu FlutterFire. Jde o sadu balíčků přinášející do aplikace jednotlivé produkty Firebase. Spojení s Firebase a dobrá integrace do Flutter projektů jsou hlavní důvody, proč jsem se rozhodl použít právě Crashlytics.

Prvním krokem v zakomponování Crashlytics do aplikace je založení projektu na webu Firebase. Projekt, jak jej chápe Firebase, je entita, která v sobě zapouzdřuje celou službu, která se pak může skládat z konkrétních aplikací jako například aplikací pro Android, iOS, webové aplikace a dalších.

Co se týče implementace, v mém konkrétním případě šlo nejdříve o nainstalování balíčků `firebase_core` a `firebase_crashlytics` do Flutter projektu. Aby se mobilní aplikace dokázala identifikovat v rámci Firebase, do Android a iOS projektu je potřeba vložit soubory s identifikačními údaji. Oba tyto soubory je možné stáhnout z webového rozhraní Firebase. Kromě zakomponování těchto souborů do projektů je ještě potřeba upravit *build skripty* daných platforem tak, aby do výsledného sestavení zakomponovaly i potřebné Firebase knihovny. Celý proces je dobře popsaný v dokumentaci FlutterFire.

Inicializace Firebase služeb v aplikaci se provádí asynchronním voláním `await Firebase.initializeApp()` a pro základní konfiguraci už prakticky není třeba nic dalšího řešit. Jednoduchým nasimulováním pádu pomocí volání funkce `FirebaseCrashlytics.instance.crash()` jsem ověřil, že pád iOS i Android aplikace se opravdu propíše až do webového rozhraní Firebase.

Aby v rámci vývoje nebyla generována chybová hlášení, omezil jsem fungování Crashlytics pro začátek pouze na testovací prostředí. Vzhledem k tomu, že Crashlytics je službou, která sbírá data ze zařízení a posílá je na vzdálený server, je potřeba mít tuto činnost právně ošetřenou. Tyto otázky zatím s Tichým světem nebyly blíže konzultovány, proto jsem odesílání chybových hlášení pomocí Crashlytics zatím nezahrnul pro produkčního sestavení aplikace. Jeho

případná budoucí aktivace pro produkční sestavení je otázkou úpravy jedné `if` podmínky.

5.3.2 Webová služba

Chybová hlášení je samozřejmě potřeba sledovat i na straně webové aplikace poskytující mobilní aplikaci REST API. Symfony aplikace ve výchozím stavu automaticky ukládají chybové záznamy do souboru, což ale samozřejmě vyžaduje jeho pravidelné kontroly. Proto jsem chtěl vyřešit monitoring chyb podobným způsobem jako v případě mobilní aplikace.

Zaznamenávání chyb běhu Symfony aplikace má na starosti knihovna Monolog. Vývoj Monologu sahá až do roku 2011 a jde o *de-facto* standard co se týče logování chybových záznamů v PHP aplikacích. Kromě běžného logování do souboru umožňuje nakonfigurovat řadu dalších způsobů pro záznam chybových incidentů. Vývojář tak může například použít zaznamenávání chyb do databáze, emailu, do konverzace v chatovací aplikaci Slack nebo například do systému Elasticsearch.

Záznam chybových hlášení do souboru je jistě dobrý základ, ale pro vývojáře je přeci jen pohodlnější, aby byl o chybových incidentech informován přímo a nemusel se tak o soubor s chybovými záznamy zajímat. Pro případ webové služby Tichého jazyka jsem se proto rozhodl použít integraci se službou Sentry, která (podobně jako Crashlytics) provádí záznam chybových hlášení. O chybách dokáže informovat například pomocí emailu a v samotném webovém rozhraní služby nabízí různé statistiky ohledně stability aplikace.

Odlišný výběr služby pro monitorování chyb v mobilní aplikaci a webovém API odůvodňuji hlavně jednoduchostí integrace dané služby s danou aplikací. Crashlytics, které se soustředí na mobilní aplikace, totiž nenabízí integraci s PHP frameworkem Symfony. Sentry sice nabízí podporu i pro framework Flutter, oproti Crashlytics mu ale chybí úzké napojení na další služby Firebase jako jsou Analytics nebo Cloud Messaging pro zasílání notifikací. Tyto služby sice momentálně nejsou v aplikaci využívány, lze ale očekávat, že v budoucnu se může objevit požadavek na jejich integraci.

Závěr

Práce úspěšně analyzuje současný stav prototypu mobilní aplikace od Alžbety Gogolákové. V rámci analýzy se práce dále seznamuje s prostředím pro výuku kurzů českého znakového jazyka v organizaci Tichý svět a jejích požadavků kladených na fungování mobilní aplikace. V rámci analýzy je věnován prostor i technickým možnostem integrace mobilní aplikace se systémem Moodle, na kterém existující e-learningový portál běží.

Na základě analýzy práce navrhuje řešení postavené na separátní webové službě implementované s použitím PHP frameworku Symfony. Implementované řešení se skládá z webové služby, která implementuje autorizaci a autentizaci uživatelů a poskytuje mobilní aplikaci data pro prohlížení studijních materiálů. Data jsou webovou službou čtena z databáze systému Moodle, autentizace je realizována s použitím JWT tokenů.

Prototyp mobilní aplikace prošel zásadními změnami, co se týče implementace. Postupně se dostal do podoby mobilní aplikace, která dokáže komunikovat s webovou službou, zprostředkovává autentizaci uživatelů a je připravena pro nasazení do produkčního prostředí. Během implementace mobilní aplikace navíc vznikl způsob pro rychlou distribuci verzí aplikace k zadavateli díky možnosti sestavit a spouštět Flutter aplikaci ve webovém prostředí.

Ačkoliv zadání práce výslovně zmiňuje použití fakultní instance GitLab pro správu zdrojového kódu a implementaci *pipelines* pro sestavení variant aplikace, v práci se od systému GitLab odkláním a místo něj je použit GitHub se službou GitHub Actions. Tato změna je v textu obhájena zejména možností spouštět automatizované skripty na strojích s operačním systémem macOS.

S použitím GitHub Actions byly implementovány skripty, které provádějí sestavení verzí aplikace pro iOS i Android pro různá prostředí pro nasazení. Aplikace byla pomocí těchto skriptů automatizovaně sestavena a nahrána do testovacích prostředí na Google Play a AppStore[20]. V nasazení mobilní apli-

[20] Ačkoliv aplikace byla nahrána na AppStore, v rámci AppStore TestFlight nebyla schválena, protože obsahovala testovací data kvůli tomu, že v době odevzdání závěrečné práce dosud nebyl Tichým světem dodám přístup pro nasazení webové služby

kace do produkčního prostředí brání skutečnost, že v době odevzdání závěrečné práce nebyl zřízen přístup pro nasazení webové služby.

V rámci práce vzniklo zajímavé *know-how*, které je podle mého názoru využitelné v praxi během vývoje mobilních aplikací. Vývojářskému týmu umožňuje rychle prezentovat nové změny zadavateli a samotným uživatelům snadno publikovat nové verze aplikace.

Seznam literatury

1. GOGOLÁKOVÁ, Alžbeta. *Návrh a implementácia mobilnej aplikácie pre študentov znakového jazyka v organizácii Tichý svět.* České vysoké učení technické v Praze, Fakulta informačních technologií, 2019. Dostupné také z: https://dspace.cvut.cz/handle/10467/83026.
2. *Parse JSON in the background* [online]. Flutter documentation, 2021-03 [cit. 2021-05-04]. Dostupné z: https://flutter.dev/docs/cookbook/networking/background-parsing.
3. GREEN, Marcus. *Moodle Database by Marcus Green* [online] [cit. 2021-05-04]. Dostupné z: https://www.examulator.com/er/.
4. *Password salting* [online]. MoodleDocs, 2014-06 [cit. 2021-05-04]. Dostupné z: https://docs.moodle.org/310/en/Password_salting.
5. PROVOS, Niels; MAZIERES, David. *Bcrypt Algorithm* [online]. 1999-04 [cit. 2021-05-04]. Dostupné z: https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html.
6. NTANTOGIAN, Christoforos; MALLIAROS, Stefanos; XENAKIS, Christos. Evaluation of Password Hashing Schemes in Open Source Web Platforms. *Computers & Security*. 2019, roč. 84, s. 206–224. ISSN 0167-4048. Dostupné z DOI: 10.1016/j.cose.2019.03.011.
7. *Modular Crypt Format* [online]. Passlib 1.7 Documentation, 1999-10 [cit. 2021-05-04]. Dostupné z: https://passlib.readthedocs.io/en/stable/modular_crypt_format.html.
8. *Using web services* [online]. MoodleDocs, 2021-02 [cit. 2021-05-04]. Dostupné z: https://docs.moodle.org/310/en/Using_web_services.
9. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures*. University of California, Irvine, 2000. ISBN 0599871180. Dostupné také z: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Dis. pr.

10. *Adding a web service to a plugin* [online]. MoodleDocs, 2021-04 [cit. 2021-05-04]. Dostupné z: https://docs.moodle.org/dev/Adding_a_web_service_to_a_plugin.
11. *Creating a web service client: How to get a user token* [online]. MoodleDocs, 2021-01 [cit. 2021-05-04]. Dostupné z: https://docs.moodle.org/dev/Creating_a_web_service_client#How_to_get_a_user_token.
12. DIERKS, Thomas; RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol* [Internet Requests for Comments]. RFC Editor, 2008-08. RFC, 5246. RFC Editor. ISSN 2070-1721. Dostupné také z: https://tools.ietf.org/html/rfc5246.
13. LEEK, Tom. *What layer is TLS?* [Online]. 2015-07 [cit. 2021-05-04]. Dostupné z: https://security.stackexchange.com/questions/93333/what-layer-is-tls/93338#93338.
14. RESCORLA, Eric. *HTTP Over TLS* [Internet Requests for Comments]. RFC Editor, 2000-05. RFC, 2818. RFC Editor. ISSN 2070-1721. Dostupné také z: https://tools.ietf.org/html/rfc2818.
15. KOLOUCH, Jan; BAŠTA, Pavel. *CyberSecurity.* 1. vyd. Praha: CZ.NIC, 2019. ISBN 978-80-88168-34-8. Dostupné také z: https://knihy.nic.cz/files/edice/cybersecurity.pdf.
16. RESCHKE, Julian. *The 'Basic' HTTP Authentication Scheme* [Internet Requests for Comments]. RFC Editor, 2015-09. RFC, 7617. RFC Editor. ISSN 2070-1721. Dostupné také z: https://tools.ietf.org/html/rfc7617.
17. JONES, Mike; HARDT, Dick. *The OAuth 2.0 Authorization Framework: Bearer Token Usage* [Internet Requests for Comments]. RFC Editor, 2012-10. RFC, 6750. RFC Editor. ISSN 2070-1721. Dostupné také z: https://tools.ietf.org/html/rfc6750.
18. PEYROTT, Sebastian. *Refresh Tokens: When to Use Them and How They Interact with JWTs* [online]. 2020-02 [cit. 2021-05-04]. Dostupné z: https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/.
19. PEYROTT, Sebastian. *Auth0 Docs* [online] [cit. 2021-05-04]. Dostupné z: https://auth0.com/docs/tokens/access-tokens.
20. JONES, Mike; BRADLEY, John; SAKIMURA, Nat. *JSON Web Token (JWT)* [Internet Requests for Comments]. RFC Editor, 2015-05. RFC, 7519. RFC Editor. ISSN 2070-1721. Dostupné také z: https://tools.ietf.org/html/rfc7519.
21. FOWLER, Martin. *Continuous Integration* [online]. 2006-05 [cit. 2021-05-04]. Dostupné z: https://www.martinfowler.com/articles/continuousIntegration.html.

22. FOWLER, Martin. *Patterns for Managing Source Code Branches* [online]. 2020-05 [cit. 2021-05-04]. Dostupné z: https://www.martinfowler.com/articles/branching-patterns.html.
23. FOWLER, Martin. *ContinuousDelivery* [online]. 2013-05 [cit. 2021-05-04]. Dostupné z: https://www.martinfowler.com/bliki/ContinuousDelivery.html.
24. GREIF, Sacha; BENITTE, Raphaël. *Data Layer* [online]. State of JS 2020 [cit. 2021-05-04]. Dostupné z: https://2020.stateofjs.com/en-US/technologies/datalayer/.
25. DRIESSEN, Vincent. *A successful Git branching model* [online]. 2010-01 [cit. 2021-05-04]. Dostupné z: https://nvie.com/posts/a-successful-git-branching-model/.
26. GADZINOWSKI, Konrad. *Trunk-based Development vs. Git Flow* [online] [cit. 2021-05-04]. Dostupné z: https://www.toptal.com/software/trunk-based-development-git-flow.
27. *Adopt a Git branching strategy* [online]. Microsoft Docs, 2020-18 [cit. 2021-05-04]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance.
28. HAMMANT, Paul. *Branch for release* [online]. Trunk Based Development, 2020 [cit. 2021-05-04]. Dostupné z: https://trunkbaseddevelopment.com/branch-for-release/.
29. HAMMANT, Paul. *Context* [online]. Trunk Based Development, 2020 [cit. 2021-05-04]. Dostupné z: https://trunkbaseddevelopment.com/context/.
30. MERKEL, Dirk. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* [online]. 2014, roč. 2014, č. 239 [cit. 2021-05-04]. ISSN 1075-3583. Dostupné z: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.
31. *About storage drivers* [online]. Docker Documentation, 2021-04 [cit. 2021-05-04]. Dostupné z: https://docs.docker.com/storage/storagedriver/.
32. *Compose file version 3 reference: build* [online]. Docker Documentation, 2021-04 [cit. 2021-05-04]. Dostupné z: https://docs.docker.com/compose/compose-file/compose-file-v3/#build.
33. *Use Docker to build Docker images: Enable Docker commands in your CI/CD jobs* [online]. GitLab Docs, 2021-04 [cit. 2021-05-04]. Dostupné z: https://docs.gitlab.com/ee/ci/docker/using_docker_build.html.

34. COBLENZ, Michael; SUNSHINE, Joshua; ALDRICH, Jonathan; MYERS, Brad; WEBER, Sam; SHULL, Forrest. Exploring Language Support for Immutability. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin, Texas: Association for Computing Machinery, 2016, s. 736–747. ICSE '16. ISBN 9781450339001. Dostupné z DOI: 10.1145/2884781.2884798.

35. *Security: Authentication & Firewalls* [online]. Symfony Docs [cit. 2021-05-05]. Dostupné z: https://symfony.com/doc/current/security.html#a-authentication-firewalls.

36. *Type declarations* [online]. PHP Manual [cit. 2021-05-05]. Dostupné z: https://www.php.net/manual/en/language.types.declarations.php.

37. *PHPDocs Basics* [online]. PHPStan User Guide, 2020-11 [cit. 2021-05-05]. Dostupné z: https://phpstan.org/writing-php-code/phpdocs-basics.

38. *Rule Levels* [online]. PHPStan User Guide, 2020-04 [cit. 2021-05-05]. Dostupné z: https://phpstan.org/user-guide/rule-levels.

39. *A tour of the Dart language: Constructors* [online]. Dart Docs, 2021-04 [cit. 2021-05-05]. Dostupné z: https://dart.dev/guides/language/language-tour#constructors.

40. *List of state management approaches* [online]. Flutter Docs, 2021-01 [cit. 2021-05-05]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/state-mgmt/options.

41. *Rendering library: RenderObject class* [online]. Dart API, 2021-04 [cit. 2021-05-05]. Dostupné z: https://api.flutter.dev/flutter/rendering/RenderObject-class.html.

42. *Flutter architectural overview: Widget state* [online]. Flutter Docs, 2021-04 [cit. 2021-05-05]. Dostupné z: https://flutter.dev/docs/resources/architectural-overview#widget-state.

43. *Start thinking declaratively* [online]. Flutter Docs, 2020-10 [cit. 2021-05-05]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative.

44. *Differentiate between ephemeral state and app state* [online]. Flutter Docs, 2020-10 [cit. 2021-05-05]. Dostupné z: https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app.

45. *What's new: Sept 10, 2019* [online]. Flutter Docs, 2020-03 [cit. 2021-05-05]. Dostupné z: https://flutter.dev/docs/whats-new#sept-10-2019.

46. HICKSON, Ian. *Web SQL Database*. 2010-11. W3C Note. W3C. Dostupné také z: https://www.w3.org/TR/2010/NOTE-webdatabase-20101118/.
47. RANGANATHAN, Arun. *Beyond HTML5: Database APIs and the Road to IndexedDB* [online]. Mozilla Hacks - the Web developer blog, 2010-06 [cit. 2021-05-05]. Dostupné z: https://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/.
48. ORLOW, Jeremy; SICKING, Jonas; POPESCU, Andrei; BELL, Joshua; MEHTA, Nikunj; GRAFF, Eliot. *Indexed Database API*. 2015-01. W3C Recommendation. W3C. Dostupné také z: https://www.w3.org/TR/2015/REC-IndexedDB-20150108/.
49. GROUP, WebAssembly Community. *Introduction* [online]. WebAssembly 1.1, 2021-04 [cit. 2021-05-05]. Dostupné z: https://webassembly.github.io/spec/core/intro/introduction.html.
50. ROUX, Alexandre. *Web Support* [online]. 2020-03 [cit. 2021-05-05]. Dostupné z: https://github.com/tekartik/sqflite/issues/212#issuecomment-598782187.
51. RUDERMAN, Jesse. *Same-origin policy: Definition of an origin* [online]. 2021-03 [cit. 2021-05-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.
52. LAINE, Emil. *Are there languages without "null"?* [Online]. 2016-01 [cit. 2021-05-05]. Dostupné z: https://stackoverflow.com/a/34570644/4646698.
53. MEYER, Bertrand; KOGTENKOV, Alexander; STAPF, Emmanuel. Avoid a Void: The Eradication of Null Dereferencing. In: *Reflections on the Work of C.A.R. Hoare*. Ed. ROSCOE, A.W.; JONES, Cliff B.; WOOD, Kenneth R. London: Springer London, 2010, s. 189–211. ISBN 978-1-84882-912-1. Dostupné z DOI: 10.1007/978-1-84882-912-1_9.
54. HOARE, Tony. *Null References: The Billion Dollar Mistake* [online]. 2009-08 [cit. 2021-05-05]. Dostupné z: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.
55. C, Yang. *Running Android Instrumented Tests on CI - from Bitrise.io to GitHub Actions* [online]. 2020-02 [cit. 2021-05-05]. Dostupné z: https://dev.to/ychescale9/running-android-emulators-on-ci-from-bitrise-io-to-github-actions-3j76.
56. *App Store Connect API* [online]. fastlane docs, 2021-03 [cit. 2021-05-05]. Dostupné z: https://docs.fastlane.tools/app-store-connect-api/.

57. *About Code Signing* [online]. Apple Inc., 2016-08 [cit. 2021-05-05]. Dostupné z: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.
58. *Sign your app: Play App Signing* [online]. Android Developers, 2021-03 [cit. 2021-05-05]. Dostupné z: https://developer.android.com/studio/publish/app-signing#app-signing-google-play.
59. SEFFERMAN, Ashley. *The App Store Optimization Checklist: Top 10 Tips: Play App Signing* [online]. 2016-03 [cit. 2021-05-05]. Dostupné z: https://moz.com/blog/app-store-optimization-checklist.
60. PEREZ, Sarah. *https://techcrunch.com/2017/08/03/google-play-will-now-downrank-poorly-performing-apps/* [online]. 2017-08 [cit. 2021-05-05]. Dostupné z: https://techcrunch.com/2017/08/03/google-play-will-now-downrank-poorly-performing-apps/.

Příloha **A**

Seznam použitých zkratek

AAB Android App Bundle

API Application programming interface

APK Android Package

ASCII American Standard Code for Information Interchange

BLoC Business Logic Component

CLI Command-line interface

ČVUT České vysoké učení technické

FIT Fakulta informačních technologií

FUP Fair Use Policy

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated development environment

IP Internet Protocol

JSON JavaScript Object Notation

JWT JSON Web Token

MAC Message authentication code

MVC Model–view–controller

NPM Node Package Manager

ORM Object–relational mapping

PDF Portable Document Format

REST Representational state transfer

SDK Software development kit

SOAP Simple Object Access Protocol

SQL Structured Query Language

SSH Secure Shell Protocol

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

URL Uniform Resource Locator

WYSIWYG What You See Is What You Get

XML-RPC Extensible Markup Language - Remote Procedure Call

YAML YAML Ain't Markup Language

Příloha B

Testovací scénáře

Vyhledání slovíčka ve Videoslovníku

1. Přejděte na obrazovku „Videoslovník velký“.
2. Pomocí scrollování nalezněte slovíčko „čaj“ a přehrajte si video.
3. Po přehrání video zavřete a vyhledejte si slovíčko „káva“. Pro jeho nalezení použijte seznam písmen abecedy u pravého okraje displeje.
4. Pomocí scrollování nebo seznamu písmen nalezněte ještě slovíčko „mléko“ a znovu si přehrajte video.

Vyhledání slovíčka v modulu

1. Pomocí levého vysouvacího panelu si otevřete seznam lekcí druhého modulu.
2. Najděte lekci „Jídlo a pití“ a rozklikněte ji.
3. Zajímá vás, jak se ve znakovém jazyce řekne slovíčko „žízeň“. V seznamu videí jej nalezněte a přehrajte si znak na videu. Slovíčko si uložte do knihovny.
4. Pomocí šipek na displeji přejděte na slovíčko „banán“ a také si jej uložte.
5. Jako poslední uložte do knihovny slovíčko „zelenina“.

Procházení slovíček v knihovně

1. Chcete si zopakovat uložená slovíčka. Přejděte do „Mojí knihovny“ a ověřte, že obsahuje váš uložený seznam ve stejném pořadí jako v samotné lekci, tedy „zelenina“, „banán“ a „žízeň“.

2. Slovíčko „žízeň“ si ale již pamatujete, v seznamu videí v knihovně jej odstraňte.
3. Rozklikněte si slovíčko „banán“ a po přehrání videa jej také odstraňte.
4. Posledním slovíčkem v knihovně by měla být „zelenina“, odstraňte i toto slovíčko.
5. Ověřte, že vás aplikace přesunula zpět z detailu videí do knihovny. Knihovna by měla být prázdná, o čemž by vás měla krátkým textem informovat.

Vymazání knihovny po odhlášení

1. Přesuňte se ještě jednou do lekce „Jídlo a pití“ v druhém modulu.
2. Uložte si do knihovny například slovíčko „maso“
3. Přejděte do „Mojí knihovny“ a ověřte, že slovíčko se v pořádku uložilo.
4. Odhlašte se a znovu se přihlašte.
5. Znovu navštivte mojí knihovnu a ověřte, že po odhlášení došlo k jejímu vymazání.

Příloha C

Obsah přiloženého USB flash disku

```
├── readme.txt ............ stručný popis obsahu flash disku
├── thesis.pdf ............ text práce ve formátu PDF
└── src
    ├── php-symfony-docker .... soubory pro sestavení Docker obrazu s PHP
    ├── thesis ................ zdrojové soubory dokumentu ve formátu LaTeX
    ├── tichyjazyk-api ........ zdrojový kód webové služby
    └── tichyjazyk-app ........ zdrojový kód mobilní aplikace
```