FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Backend webové aplikace na učení se angličtiny
Student:	Jan Hlaváč
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je ve spolupráci s Janem Jeníčkem realizovat webovou aplikaci na učení se angličtiny. Z důvodu velkého rozsahu a také aby byla zajištěna forma izolace od práce Jana Jeníčka je předmětem této konkrétní práce backendová část aplikace, tedy REST API a databáze.

Postupujte v následujících krocích:
1. Na základě analýzy konkurence od Jana Jeníčka analyzujte funkční a nefunkční požadavky s důrazem na backendovou část aplikace. Dále analyzujte algoritmy na učení se jazyků.
2. Na základě analýzy proveďte důkladný návrh a zvolte vhodné technologie backend části.
3. Implementujte backend část aplikace s ohledem na vhodné algoritmy učení.
4. Navrhněte a realizujte vhodné formy testů backend části aplikace.
5. Nasaďte svou část práce do produkčního prostředí.
6. Zhodnoťte výsledné řešení, navrhněte úpravy do budoucna.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 4. února 2024 v Praze.

Bakalářská práce

BACKEND WEBOVÉ APLIKACE NA UČENÍ SE ANGLIČTINY

Jan Hlaváč

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Hunka
16. května 2024

Odkaz na tuto práci: Hlaváč Jan. *Backend webové aplikace na učení se angličtiny.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam algoritmů

Děkuji panu Ing. Jiřímu Hunkovi za vedení této práce, konkstruktivní zpětnou vazbu a vstřícnou neformální komunikaci. Dále bych rád poděkoval kolegovi Janu Jeníčkovi za příležitost pracovat na smysluplném projektu se skutečným potenciálem a za dlouhé společné chvíle strávené mimo jiné i navrhováním aplikace. Rovněž velké díky patří kolegovi Jakubu Vondráčkovi, který mi během vývoje aplikace poskytl nespočet nedocenitelných rad k její technické realizaci. V neposlední řadě děkuji také svým dalším přátelům-kolegům za vzájemnou podporu, inspiraci a motivaci a slečně Tereze Papíkové za lingvistickou asistenci.

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, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

Práce se zabývá návrhem a implementací backendu aplikace na učení se angličtiny. Na začátku práce byly analyzovány algoritmy efektivního učení a specifikovány požadavky na aplikaci. Na jejich základě byl proveden návrh architektury, doménového modelu, REST API a systému správy předplatného. Následně byla aplikace naimplementována, přičemž byl kladen důraz na implementaci algoritmu řídícího postup učení uživatele. Aplikace byla podrobena testování ve formě jednotkových a systémových testů a nasazena do produkčního prostředí formou průběžného nasazování. Ve spolupráci s frontendovou částí tak tvoří ucelený systém, jenž je volně přístupný méně i více náročným samoukům, kteří se chtějí efektivní formou zdokonalit v anglickém jazyce.

Klíčová slova aplikace, rest api, backend, učení se angličtiny, C#, ASP.NET Core

Abstract

This thesis deals with the design and implementation of the backend of an English learning application. At the beginning of the work, the algorithms of effective learning were analyzed and the requirements of the application were specified. Based on these, the architecture, domain model, REST API and subscription management system were designed. Subsequently, the application was implemented, with emphasis on the implementation of the algorithm controlling the user learning procedure. The application was tested in the form of unit and system tests and deployed to the production environment in the form of continuous deployment. In cooperation with the frontend part, it forms a complete system that is freely accessible to less and more demanding self-taught users who want to improve their English in an efficient way.

Keywords application, rest api, backend, english learning, C#, ASP.NET Core

Seznam zkratek

SDLC	Software Development Lifecycle
REST	Representational State Transfer
API	Application Programming Interface
SQL	Structured Query Language
ORM	Object–relational mapping
CRUD	Create, Read, Update, Delete
KISS	Keep It Simple Stupid
POGE	Principle Of Good Enough
URI	Uniform Resource Identifier
JSON	JavaScript Object Notation
JWT	JSON Web Token
DTO	Data Transfer Object
PaaS	Platform as a Service
CI/CD	Continuous Integration / Continuous Deployment
DDL	Data Definition Language

Úvod

Jednou z mnoha oblastí, které byly zásadně ovlivněny příchodem internetu a následně i chytrých telefonů, je vzdělávání. Díky nepřebernému množství dostupných informačních zdrojů se může v dnešní době každý vzdělávat svým vlastním tempem, a nemusí být závislý pouze na školní výuce či předražených kurzech. Snad nejvíce patrné je to v oblasti učení se jazyků, pro které dnes najde využití prakticky každý. A tak není divu, že už od 90. let vznikají tematické webové stránky a jazykové online kurzy a s rozmachem chytrých telefonů pak i mnohé mobilní aplikace.

I v tématu nezběhlému čtenáři se pravděpodobně v této souvislosti ihned vybaví Duolingo, které ovládá více než 60 % trhu s jazykovými aplikaci a 12,5 % trhu se všemi vzdělávacími aplikacemi. Méně významných aplikací však existuje celá řada – některé se silně inspirovaly Duolingem, kde průchod aplikací téměř připomíná hraní videohry, jiné zase naopak dávají větší volnost uživateli a nechávají ho procházet učením podle svých možností a představ, ovšem často za cenu nižší uživatelské přívětivosti, zejména pro méně zkušené uživatele.

Málokterá aplikace však kombinuje všechny žádoucí vlastnosti tak, aby byla vhodným nástrojem pro širší spektrum uživatelů. Jednak pro náročnější uživatele, kteří už mohou mít s daným jazykem předchozí zkušenosti, chtějí mít větší kontrolu nad tím, v jakých tematických okruzích se budou při učení zdokonalovat a potrpí si na to, aby aplikace využívala efektivní učicí algoritmy, tak i pro běžného volnočasového samouka, který naopak ocení jednoduchost, přehlednost a případně i hravost aplikace. Tyto důvody vedly k iniciativě zahájit vývoj vlastní aplikace na učení se angličtiny, která by tyto vlastnosti vhodně kombinovala a vyvažovala.

Tato práce se zabývá backendovou částí této aplikace, v teoretické části se zaměřuje zejména na specifikaci požadavků, které má aplikace splňovat, a na rozbor algoritmů efektivního učení. Kolega Jan Jeníček se naopak ve své bakalářské práci *Frontend webové aplikace na učení se angličtiny* věnuje frontendové části a analýze konkurenčních řešení s rozborem případů užití.

Cílem této práce je tedy vytvořit funkční backend aplikace na učení angličtiny jako výsledek kompletního softwarového procesu. Nejprve bude provedena

nezbytná analýza sestávající z analýzy požadavků a algoritmů učení, dále na základě toho bude vytvořen návrh řešení backendové části aplikace a následně bude implementována základní část aplikace tak, aby bylo možné ji otestovat a nasadit do reálného produkčního prostředí.

Kapitola 1

Teorie softwarového procesu

Předtím, než začnu popisovat průběh vývoje aplikace v rámci jednotlivých fází softwarového procesu, vysvětlím význam tohoto pojmu v současném softwarovém inženýrství a specifikuji, čím se zabývají samotné jeho fáze.

Softwarový proces, v anglické literatuře *software development process* nebo také *software development lifecycle (SDLC)*, je množina aktivit nutných k tomu, aby software vznikl [1]. Jde o opakovatelný proces tvorby softwaru, díky kterému bude moct celý produktový tým systematicky, efektivně a co možná nejpředvídatelněji pracovat na dodání softwaru, který co nejlépe odpovídá představám zákazníka. Přestože existuje několik různých modelů SDLC – jejichž analýze se ve své práci věnuje kolega Jeníček – všechny sestávají z několika následujících fází, které jsou pro úspěch projektu nutné [2].

1.1 Analýza

První fází softwarového procesu, která je obecně považována i za nejdůležitější, je analýza. Smyslem této fáze je rozmyslet a naplánovat, co a jak se bude v dalších fázích dělat. Zahrnuje zejména sběr požadavků, analýzu případů užití, analýzu konkurenčních řešení a případně i analýzu dosavadního stavu systému. V business prostředí by na úrovni projektového managementu do této fáze také spadalo odhadování ceny a potenciálních zisků, přidělování zdrojů a sestavení časového rámce projektu.

1.2 Návrh

Po analýze následuje fáze návrhu. Jejím cílem je na základě analýzy navrhnout, jakým způsobem budou naplněny jednotlivé požadavky. Během návrhové fáze je potřeba představy zákazníka transformovat do konkrétních podkladů pro vývojový tým tak, aby bylo co nejvíce eliminováno riziko nejednoznačnosti a vzájemného nepochopení. Je důležité obsáhnout všechny části navrhovaného

systému, neboť odhalení nedostatků ve fázi návrhu je pro rozpočet projektu řádově levnější, než až během implementace. Obvykle tato fáze zahrnuje volbu programovacího jazyka a frameworků, způsobu uložení dat, tvorbu databázového modelu a návrhu architektury, a v případě tvorby uživatelského rozhraní je pak obvykle vytvořen i prototyp, na základě kterého získá zákazník jasnou představu o podobě rozhraní a může navrhovat jeho úpravy.

1.3 Implementace

Na základě důkladného návrhu je již možné software naimplementovat. V případě backendového projektu je do této fáze zahrnuto i vytvoření databázového schématu a propojení jednotlivých komponent. Po celou dobu vývoje je třeba dbát na kvalitní organizaci kódu, tedy využívat vhodné architektonické a návrhové vzory, které zajistí, aby byl výsledný kód udržovatelný, rozšiřitelný a testovatelný [3]. V případě vícečlenného vývojového týmu je nezbytné práci efektivně koordinovat a synchronizovat [4], k čemuž jsou využívány systémy na správu verzí a nástroje pro řízení projektů a správu úkolů.

1.4 Testování

Vytvořený, nebo stále ještě vznikající, software je vhodné náležitě otestovat, aby bylo zajištěno splnění zadání a aby byl dodán co nejspolehlivější produkt [5]. V závislosti na zvoleném modelu SDLC probíhá testování v různém časovém horizontu, obecně se ale doporučuje zahájit testování v odpovídající formě co nejdříve již během návrhové fáze, průběžně testovat během vývoje a pokračovat s ním i po nasazení [6]. Významnými typy testů jsou ku příkladu *jednotkové testy*, které testují software po co nejmenších částech, *integrační testy*, které se zaměřují na spolupráci jednotlivých komponent, *systémové testy* zajišťující, že kompletní systém funguje dohromady jako celek, a *akceptační testy*, které ověřují, že software splňuje podmínky zadání a je připraven k nasazení [7]. Některé fáze testování je vhodné automatizovat, aby bylo možné testy provádět často a rychle.

Nasazení

V jistou chvíli je již možné software vydat k užívání. To může proběhnout buď jednorázově, nebo postupně s přidáváním jednotlivých změn. V takovém případě je vhodné zavést proces průběžného nasazování (*Continuous Delivery*), díky kterému lze nové změny rychle a automatizovaně aplikovat na produkční prostředí [8]. Nasazený software je pak vhodné neustále monitorovat, aby bylo možné odhalovat chyby, které se projeví až v reálném provozu při použití skutečnými uživateli.

Kapitola 2
Analýza

V této kapitole se budu věnovat specifikaci požadavků a analýze algoritmů učení. Podrobný rozbor případů užití, které navazují na analýzu požadavků, a analýzu konkurenčních řešení zpracoval kolega Jeníček ve své práci.

2.1 Specifikace požadavků

Před začátkem vývoje každého systému je nezbytné vydefinovat, co se od výsledku očekává. V případě vývoje softwaru na zakázku by specifikace požadavků probíhala společně se zákazníkem, aby dodavatel vytvořil dílo přesně podle zákazníkových představ a aby bylo možné stanovit rozsah dodávky a od toho se odvíjející odhadovanou náročnost a cenu. V případě této aplikace byly požadavky specifikovány přímo vývojáři na základě rozsáhlé analýzy konkurenčních řešení, kterou zpracoval kolega Jeníček.

Požadavky se standardně dělí do dvou hlavních skupin, funkční a nefunkční [9]. Pro vyšší granularitu lze použít framework *FURPS*, který rozděluje nefunkční požadavky ještě do několika dalších kategorií. Jsou jimi:

- Functionality
- Usability
- Reliability
- Performance
- Supportability

Díky tomu je méně pravděpodobné, že by došlo k opomenutí některé z důležitých kategorií a nebyly tak vyspecifikovány požadavky, které se jí týkají. Tento framework je velice hojně používaný nejen v softwarovém inženýrství a není přehnaně komplikovaný ani nepřehledný. Jelikož FURPS vznikl ve společnosti Hewlett-Packard již v 90. letech minulého století [10, s. 159], došlo

od té doby k jeho rozšíření. *FURPS+* slouží jako zastřešující akronym pro různá rozšíření základního frameworku, aby bylo možné klást zvýšený důraz na specifičtější atributy nebo přidávat další oblasti, na které je při návrhu daného systému potřeba pamatovat [11, s. 32]. Například ve společnosti IBM se mezi tyto rozšiřující kategorie řadí požadavky designové, implementační, fyzické a požadavky na rozhraní [12].

Jak je také zmiňováno ve výše uvedených zdrojích, je zásadní určit prioritu jednotlivých požadavků. Zřídkakdy se podaří v daném časovém horizontu uspokojit všechny specifikované požadavky, a je tedy žádoucí, aby byly zahrnuty alespoň ty nejzásadnější. Nejjednodušší způsoby prioritizace obvykle uvažují nějakou relativní škálu, například *vysoká*, *střední* a *nízká* priorita, případně číselnou škálu, což ale často vede k nejistotám a neshodám v týmu ohledně toho, kam který požadavek zařadit. Navíc takové hodnoty většinou postrádají sémantický význam: je požadavek s prioritou 4 opravdu nutné implementovat, aby byl systém vůbec provozuschopný, nebo vůbec zásadní není, jen je o něco důležitější, než požadavky s prioritou 3? Na tyto nedostatky reaguje technika *MoSCoW*, která je s oblibou uplatňována v agilním vývojovém přístupu [13, Kapitola 10]. Pomocí čtyř klíčových slov poskládaných do úderného akronymu jednoduše popisuje, jak jsou které požadavky důležité, a jednoznačně určuje, jak nutné je jejich naplnění pro úspěch projektu. Těmito prioritami jsou:

- Must have
- Should have
- Could have
- Won't have this time

V této podkapitole tedy budu požadavky kategorizovat dle frameworku FURPS+ a přiřazovat jim priority z MoSCoW.

2.1.1 Funkční požadavky

FR1 Registrace a přihlašování uživatelů (must have)

Noví uživatelé se budou moct volně zaregistrovat pomocí e-mailové adresy a zvoleného hesla. Následně bude uživateli umožněno zvolit si uživatelské jméno a provést další nastavení uživatelského účtu. K existujícímu uživatelskému účtu se bude moct uživatel přihlásit uvedením e-mailové adresy a hesla.

FR2 Změna hesla (must have)

V případě, že uživatel zapomene své stávající heslo nebo ho bude chtít změnit, bude možné zadáním e-mailové adresy příslušející uživatelskému účtu požádat o změnu hesla.

FR3 Uživatelské role (must have)

Systém bude každému uživateli přiřazovat uživatelské role. Konkrétně půjde o role *uživatel*, *prémiový uživatel* a *admin*. Po registraci dostane každý roli uživatel, prémiovým uživatelem se stane po objednání předplatného (viz dále). Administrátorskou roli nemůže běžný uživatel nijak získat, slouží administrátorům systému pro správu obsahu.

FR4 Kurzy (must have)

Administrátor bude moct vytvářet nové kurzy. U kurzu bude systém evidovat název, odkaz na náhledový obrázek a kódy mateřského a cílového jazyka. Uživatelé si budou moct zobrazit tyto informace o kurzu a případně se do libovolného kurzu zapsat.

FR5 Lekce (must have)

Administrátor bude moct v každém kurzu vytvářet lekce a případně je zařazovat do různých témat. Systém bude rozlišovat několik typů lekcí, například lekce na gramatiku, poslech a slovíčkové lekce. Slovíčkové lekce budou sloužit jako uspořádaná množina slovíček, zatímco ostatní typy už budou rovnou obsahovat jednotlivá cvičení.

Lekce budou v rámci kurzu lineárně uspořádány, aby bylo možné evidovat uživatelův postup kurzem. Uživatel si bude moct ve svých kurzech lekce libovolně skrývat a zobrazovat a označovat je jako oblíbené.

FR6 Témata (must have)

Administrátor bude moct v každém kurzu vytvářet témata, která budou sloužit pro seskupování lekcí. U témata bude systém evidovat název, odkaz na náhledový obrázek a kategorii. Dále bude moct administrátor určit, zda bude téma výchozí pro daný kurz, tedy zda bude ve výchozím stavu zapnuté. Uživatel si bude moct ve svých kurzech témata libovolně zapínat a vypínat, čímž ovlivní viditelnost lekcí obsažených v tématu.

FR7 Zpětná vazba lekce (could have)

Uživatel bude moct lekci ohodnotit příznakem „líbí se“/„nelíbí se“ a případně přidat textový komentář.

FR8 Uživatelské lekce (should have)

Uživatel si bude moct vytvářet vlastní slovíčkové lekce a seskupovat tak již existující slovíčka dle libosti.

FR9 Slovíčka (must have)

Administrátor bude moct vytvářet slovíčka. U slovíčka bude systém evidovat překlady v mateřském a cílovém jazyce a odkazy na ilustrační obrázek a nahrávku výslovnosti. Uživatel si bude moct slovíčko označit jako oblíbené.

Ke každému slovíčku budou také evidovány příklady jeho použití ve větách.

FR10 Cvičení (must have)

Administrátor bude moct k lekcím a slovíčkům vytvářet cvičení, která budou sloužit k samotnému procvičení dané problematiky. Bude existovat několik druhů cvičení, například *fill in the blank*, *překlad věty*, *vyplňování tabulky* a další.

FR11 Denní učení (must have)

Uživatel bude moct v rámci kurzu kdykoliv spustit tzv. denní učení s požadovanou dobou trvání. Systém prostřednictvím sofistikovaného algoritmu vybere cvičení, která si má uživatel zopakovat, a k nim přidá nová cvičení z lekce, kterou se má uživatel učit jako následující. Počet vybraných cvičení by měl být zvolen tak, aby celková odhadovaná doba jejich plnění odpovídala požadované době trvání. Následně systém cvičení vyhodnocená od uživatele zpracuje a zaznamená jeho postup.

FR12 Učení lekce (must have)

Uživatel bude moct spustit i učení konkrétní lekce. Systém vybere cvičení, která si má v rámci této lekce uživatel zopakovat a přidá k nim nová cvičení z této lekce. Následně systém cvičení vyhodnocená od uživatele zpracuje a zaznamená jeho postup.

FR13 Učicí algoritmus (must have)

Algoritmus, který vybírá cvičení k učení, bude využívat metodu spaced repetition tak, aby co nejefektivněji využil čas a učící potenciál každého daného uživatele. Konkrétní vlastnosti celého algoritmu jsou rozebrány dále v této práci.

FR14 Úvodní test (could have)

Při zápisu do kurzu bude mít uživatel možnost projít úvodním testem, který určí jeho úroveň. Na základě toho bude rovnou upraven jeho postup kurzem, aby nezačínal od úplného začátku.

FR15 Placení předplatného (should have)

V aplikaci bude fungovat systém předplatného, v rámci kterého budou některé funkce dostupné pouze prémiovým uživatelům.

FR16 Uživatelský profil (should have)

Uživatel si bude moct zobrazit a upravovat svůj uživatelský profil. Bude si moct nastavovat například profilový obrázek, jméno a cíle učení.

FR17 Statistiky učení (should have)

Uživatelům bude systém zaznamenávat, kolik cvičení udělali v rámci jednotlivých dní za dobu používání aplikace.

FR18 Sledování uživatelů (could have)

Uživatelé se budou moct navzájem sledovat. Úspěchy sledovaných uživatelů se budou zobrazovat na hlavní stránce.

FR19 Ocenění (could have)

Uživatel bude moct v aplikaci získávat tzv. ocenění. Ocenění budou jasně definované úkoly, které se po splnění budou zobrazovat na veřejném profilu daného uživatele. Za splnění ocenění uživatel bude moct získat dané množství virtuální měny.

FR20 Výzvy (won't have this time)

Uživatelé budou moct plnit výzvy. Konkrétní výzva se bude zobrazovat vždy v určeném časovém rozmezí mezi dvěma daty. Výzva se může týkat počtu dokončených cvičení za dané období nebo splnění konkrétních lekcí.

FR21 Obchod (won't have this time)

V aplikaci bude uživatel moct učením získávat virtuální měnu. V aplikaci bude stránka obchodu, kde bude uživatel moct za virtuální měnu nakupovat různé položky, které přizpůsobí vzhled aplikace nebo uživatelského profilu.

2.1.2 Nefunkční požadavky

NR1 Bezpečné přihlašování a nakládání s hesly (must have, R)

Systém bude používat kvalitní a prověřené kryptografické protokoly pro registraci, přihlašování uživatelů a obnovu hesla. S uživatelskými hesly bude nakládáno v souladu s doporučeními NIST tak, aby bylo co nejvíce eliminováno riziko jejich zneužití.

NR2 Rest API (must have, U)

Systém bude mít vhodně navržené REST API, se kterým bude komunikovat frontendová část aplikace.

NR3 Dokumentace (should have, U)

Systém bude dostatečně vhodnou formou zdokumentován alespoň na úrovni koncových bodů API.

NR4 Rozšiřitelnost (must have, S)

Architektura systému bude navržena tak, aby bylo možné jakoukoliv funkcionalitu snadno rozšířit či doplnit. Tomu bude odpovídat i použití vhodných návrhových vzorů a samotný způsob implementace.

NR5 Testovatelnost (should have, S)

Architektura a způsob implementace systému bude umožňovat testování všech funkcionalit, a to jak jednotkovými testy, tak i komplexnějšími typy testů.

2.2 Algoritmy učení

Pro určování, kdy si má která cvičení uživatel procvičovat, je nezbytné zvolit vhodný algoritmus. Ten by měl zajistit, aby se uživatel naučil jazyk co nejefektivněji, tedy v co nejlepším čase a na co nejdelší dobu. Již přes sto let se ukazuje, že pro tyto účely je nejvhodnější metoda spaced repetition [14].[1]

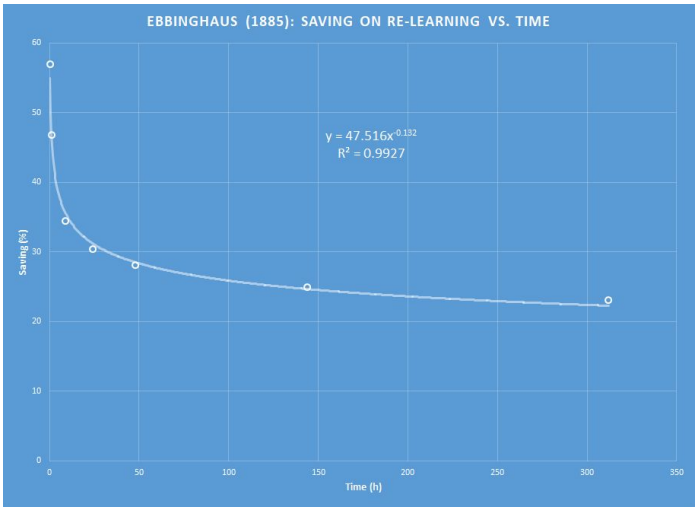
Zároveň bude algoritmus v aplikaci využívat principu aktivního vybavování [15]. Tento v zásadě intuitivní, ačkoliv školami dodnes často opomíjený, princip říká, že si látku student zapamatuje mnohem lépe, když si ji sám aktivně snaží připomenout, než když informace pouze pasivně vstřebává. Alternativně se označuje také jako testovací efekt, což poukazuje na to, že testování vlastních znalostí oproti správným odpovědím je primárně klíčové pro učení, nikoliv pro známkování. Tento princip byl známý učencům již v 17. století, a je tak obecně užitečný nezávisle na metodě spaced repetition [16].

2.2.1 Spaced repetition

Základem této metody byla práce H. Ebbinghause na konci 19. století, který zkoumal vliv opakovaného učení dané látky na míru její zapamatovatelnosti [17]. Mimo jiné se zabýval i tím, jak dobře si člověk látku pamatuje v závislosti na době, která uplynula od učení – to ukazuje dnes již známá Ebbinghausova křivka (obr. 2.1), z níž je patrné, že k největší míře zapomínání dochází hned

[1] dříve také někdy nazývaná *distributed practice*

po ukončení učení a postupně zpomaluje. Tato křivka se však v Ebbinghausově práci nikde nevyskytuje, poněvadž sám provedl pouze několik diskrétních pozorování, z nichž nevyvozoval žádné větší závěry.



Obrázek 2.1 Křivka zapomínání vykreslená na základě dat tabelovaných H. Ebbinghausem [18]

Pokud si tedy studující chce látku v paměti udržet co nejdéle, je potřeba ji pravidelně opakovat s tím, že začne kratšími intervaly, které se budou postupně prodlužovat [19]. Značné množství výzkumu během 20. století bylo věnováno hledání optimálních hodnot těchto intervalů [20] a jednotlivé algoritmy, které metodu spaced repetition využívají, se liší v podstatě pouze v tom, jak tyto intervaly určují v závislosti na dosavadním studiu.

Všechny následující algoritmy pracují se studijním materiálem rozděleným na co nejmenší páry: otázka a odpověď (e.g. v případě učení slovíček překlady v mateřském a cizím jazyce). Tyto páry se v literatuře obvykle označují jako *items*, a dále v textu tak bude užíváno tvarů *item, pl. itemy.*

2.2.1.1 Leitner

Systém Leitnerovy kartotéky z roku 1972 je založen na pěti krabičkách, do kterých jsou jednotlivé itemy rozdělené [21]. Na začátku jsou všechny itemy v krabičce číslo 1, a při opakování jsou po správné odpovědi posunuty o krabičku dále. Takto probíhají opakování v rámci všech krabiček, jen při špatné odpovědi se itemy z vyšších krabiček vrací zpět do krabičky první. Každá krabička má pevně stanovenou konstantu, po jak dlouhé době má probíhat opakování jednotlivých itemů. Zásadní faktorem je pak to, že pro vyšší krabičky jsou tyto intervaly čím dál tím delší.[2] Nevýhodou tohoto algoritmu však je, že je

[2]Leitner je navrhoval v poměru 1 : 2 : 5 : 8 : 14

založen na pouhé heuristice s předem určenými intervaly, a nijak nezohledňuje náročnost daného itemu pro konkrétního studenta.

2.2.1.2 SuperMemo

Jednou z hlavních postav vývoje spaced repetition je polský výzkumník Piotr Woźniak, který ve svých dvaceti letech začal hledat efektivní způsob, jak si látku ze školy zapamatovat na delší dobu, než jen kvůli zkoušce [22]. Jsa neobeznámen s předchozím výzkumem v této oblasti, postupoval podobně jako Ebbinghaus a sám na vlastních studijních materiálech testoval, které intervaly mezi opakováními by měly být nejvhodnější. Poznatky z jeho experimentu vedly k algoritmu SM-0.

Algoritmus 1 Algoritmus SM-0 [23]

1. Rozděl látku na co nejmenší itemy otázka–odpověď.
2. Seskup itemy do skupin po 20–40 prvcích. Tyto skupiny jsou dále označovány jako stránky.
3. Opakuj si celé stránky dle následujících intervalů: $I(1) \leftarrow 1$ den $I(2) \leftarrow 7$ dnů $I(3) \leftarrow 16$ dnů $I(4) \leftarrow 35$ dnů, pro $i > 4$: $I(i) \leftarrow I(i-1) \times 2$, kde: I(i) je interval použitý po i-tém opakování.
4. Zkopíruj všechny itemy zapomenuté po 35denním intervalu do nově vytvořených stránek (aniž bys je odebral z již dříve používaných stránek). Tyto nové stránky se budou opakovat stejným způsobem jako stránky s novými itemy.

Algoritmus SM-0 je velice podobný tomu Leitnerovu, jen s tím rozdílem, že nabízí rozšíření (zobecnění) pro libovolné množství krabiček a používá jiné odhady pro hodnoty intervalů. Zároveň Woźniak z počátku pracoval s látkou rozdělenou na stránky, které se učily najednou, ale tento přístup následně sám zavrhl.

Když Woźniak přešel ze studia mikrobiologie na informatiku, pochopitelně se rozhodl svůj algoritmus implementovat na počítači, aby měl své studijní materiály digitalizované a učení bylo plánované automaticky. V programu *SuperMemo 1.0 for DOS (1987)* se ale rozhodl algoritmus podstatně vylepšit, a to tak, aby byla zohledněna složitost každého konkrétního itemu. Tento algoritmus se nazývá SM-2.

Pro čtenáře preferujícího čtení zdrojového kódu před pseudokódem algoritmu je dostupná komentovaná implementace všech zde diskutovaných algoritmů v jazyce Java.[3]

[3] https://github.com/nickhnsn/facharbeit-spaced-repetition/tree/master

Algoritmus 2 Algoritmus SM-2 [23]

1. Rozděl látku na co nejmenší itemy.
2. Všem itemům přiřaď E-Faktor rovný 2,5.
3. Opakuj si itemy dle následujících intervalů: $I(1) \leftarrow 1$ den $I(2) \leftarrow 6$ dnů, pro $i > 2$: $I(i) \leftarrow I(i-1) \times EF$, kde:
   - I(i) – interval použitý po i-tém opakování,
   - EF – E-Faktor určující snadnost zapamatování a udržení daného itemu v paměti.

   Je-li interval zlomek, zaokrouhli ho nahoru na celé číslo.
4. Po každém opakování ohodnoť kvalitu odpovědi na škále od 0 do 5:
   - 5 – perfektní odpověď
   - 4 – správná odpověď po zaváhání
   - 3 – správná odpověď vybavená se začnou námahou
   - 2 – nesprávná odpověď, přičemž ta správná se zpětně zdá snadná na vybavení
   - 1 – nesprávná odpověď, ale ta správná byla zapamatována
   - 0 – naprostý výpadek
5. Po každém opakování uprav E-Faktor právě opakovaného itemu podle vzorce: $EF \leftarrow EF + (0.1 - (5-q) \times (0.08 + (5-q) \times 0.02))$, kde:
   - EF' – nová hodnota E-Faktoru,
   - EF – původní hodnota E-Faktoru,
   - q – kvalita odpovědi na škále od 0 do 5.

   Je-li EF menší než 1,3, pak nechť EF je 1,3.
6. Je-li kvalita odpovědi nižší než 3, pak zahaj opakování itemu od začátku beze změny E-Faktoru (t. j. použij intervaly I(1), I(2) atd. jako by byl item učen jako nový).
7. Po každém kole opakování v daném dni si znovu zopakuj všechny itemy, které získaly v hodnocení kvality odpovědi méně než 4. Pokračuj v opakování, dokud všechny tyto itemy nezískají hodnocení alespoň 4.

Během let dalšího výzkumu postupně Woźniak algoritmus ještě vylepšoval. K dynamickým úpravám E-faktoru přibyla i možnost, jak modifikovat celou funkci optimálních intervalů podle konkrétního uživatele, tedy tak, aby i nově učené itemy mohly rovnou benefitovat ze zkušenosti starších [23, Kapitola 3.3]. Nejnovější verzí je aktuálně algoritmus SM-18 z roku 2019, ale zde je třeba zvážit efektivitu algoritmu proti náročnosti implementace. Ačkoliv stojí SM-2 prakticky na počátku vývoje SuperMemo algoritmů, ukazuje se jako mimořádně efektivní a v kombinaci s jednoduchostí implementace je ideálním kompromisem. O tom svědčí i fakt, že je na něm založen i učicí algoritmus populární aplikace Anki [24].

2.2.1.3 FSRS

Algoritmus FSRS *(Free Spaced Repetition Scheduler)* je založen na třísložkovém modelu paměti DSR, se kterým jako první pochopitelně přišel Piotr Woźniak [25]. Tento algoritmus z roku 2022 je ale dílem komunity *Open Spaced Repetition* a vychází z poznatků výzkumu algoritmu SSP-MMC [26]. Samotný algoritmus je již o poznání složitější než předchozí uvedené, a zde tak není rozepsán.[4]

Na základě experimentálních výsledků se z hlediska schopnosti odhadnout pravděpodobnost vybavení itemu po dané době vyrovná algoritmu SM-17 a algoritmus SM-2 jednoznačně předčí [27]. Aplikace Anki ho nativně podporuje od 1. listopadu 2023 [28].

2.2.1.4 Volba algoritmu

Z diskutovaných (a v době psaní práce nejspíše i z veřejně dostupných) algoritmů se jako nejefektivnější algoritmus pro učení metodou spaced repetition jeví algoritmus FSRS. Zejména kvůli vyšší náročnosti jeho implementace bude však v první verzi aplikace zvolen algoritmus SM-2, který je implementačně podstatně jednodušší a je stále více než konkurenceschopný.

Vlastní algoritmus aplikace bude tedy využívat tento spaced repetition algoritmus. Konkrétní způsob jeho implementace bude diskutován dále v textu v rámci dalších fází softwarového procesu.

[4] https://github.com/open-spaced-repetition/fsrs4anki/wiki/The-Algorithm

Kapitola 3
Návrh

V návrhové části se nejprve zaměřím na volbu technologií, architekturu systému jako celku a pak i samotné backendové části. Následně se budu věnovat problémové doméně aplikace, REST API a poté dalším netriviálním aspektům systému, které vyžadují důsledný návrh. Konkrétně popíšu návrh procesu správy předplatného a způsoby, jak bude aplikace zajišťovat autentizaci a autorizaci uživatelů.

3.1 Metodika vývoje

Výběru vhodného modelu softwarového procesu se ve své práci věnuje kolega Jeníček, které pro naši spolupráci zvolil formu agilního vývoje *Kanban*. Ta zahrnuje použití virtuální nástěnky, na kterou jsou umisťovány jednotlivé úkoly rozdělené do sloupců podle fází vývoje, v nichž se zrovna nachází [29]. Jako implementaci této nástěnky použijeme nástroj *Jira*.[1] V něm budeme s kolegou vytvářet úkoly spojené s tímto projektem, mé úkoly týkající se backendové části aplikace budou obvykle odpovídat jednotlivým koncovým bodům jejího API.

3.2 Volba technologií

V rámci návrhové části softwarového procesu je důležité zmapovat, jaké technologie pro vývoj aktuálně existují a vybrat z nich ty nejvhodnější pro zadaný projekt. Jelikož se tato práce zaměřuje na backendovou část aplikace, budu se zde věnovat volbě databázového úložiště a programovacích jazyků a frameworků pro vývoj API.

[1] https://www.atlassian.com/software/jira

3.2.1 Programovací jazyky a frameworky

Existuje nepřeberné množství programovacích jazyků, ve kterých je lépe či hůře možné vyvíjet webové aplikace. Ze všech těchto jazyků jsem se rozhodl vybrat čtyři nejpoužívanější pro backendový vývoj [30]. Jsou jimi *Python*, *Java*, *PHP* a *C#*. Prakticky vždy se místo čisté verze programovacího jazyka k vývoji aplikací využívá nějaký framework, a tak je vhodné spojit volbu jazyka už i s konkrétním frameworkem a porovnávat rovnou ty. Opět z těch nejpoužívanějších jsem pro porovnání zvolil *Django* (Python),[2] *Spring Boot* (Java),[3] *Symfony* (PHP)[4] a *ASP.NET Core* (C#)[5].

3.2.1.1 Oficiální dokumentace a podpora tvůrců

Ke všem frameworkům samozřejmě autoři poskytují oficiální dokumentaci. Ta má vesměs vždy stejnou formu (webové stránky se stromovou strukturou, navzájem provázané odkazy), ale liší se v rozsáhlosti a kvalitě.

Jako nejpřehlednější se jeví dokumentace Symfony. Má přívětivý moderní design, je přehledně strukturovaná a pokrývá snad všechny oblasti, které by mohl vývojář v dokumentaci hledat. Zároveň je ke každému tématu uvedeno mnoho příkladů s ukázkami kódu a jsou uvedeny různé možnosti, jak řešit konkrétní problémy.

Podobně obsáhlá je i dokumentace Django, jen je o trochu těžší najít řešení konkrétního problému.

Dokumentace Spring Boot, potažmo Spring, už působí méně obsáhle, zejména v oblasti poskytování odpovědí na časté problémy a vysvětlování běžných postupů. Nicméně alespoň v samostatné sekci obsahuje několik užitečných návodů, jak provést některé základní úkony.

V případě ASP.NET Core je situace srovnatelná, jen užitečných návodů je k dispozici snad ještě méně. Celkově je tak tato dokumentace spravovaná společností Microsoft ve srovnání s ostatními nejslabší.

Všechny frameworky jsou tvůrci pravidelně aktualizovány a vylepšovány, Django, Symfony a Spring jsou open-source projekty. Celá technologie .NET je vyvíjena a udržována společností Microsoft, což má na jednu stranu výhodu v zázemí velké a stabilní firmy, která poskytuje i kvalitní nástroje pro vývojáře, na druhou stranu je vývojář v mnoha ohledech závislý na rozmarech této společnosti.

3.2.1.2 Popularita a podpora komunity

Dalším zásadním faktorem pro výběr frameworku je míra jeho rozšířenosti a popularity mezi vývojáři a s tím související velikost a kvalita komunity kolem

[2] https://www.djangoproject.com
[3] https://spring.io
[4] https://symfony.com
[5] https://dotnet.microsoft.com/en-us/apps/aspnet

Tabulka 3.1 Volba frameworku: popularita dle Stack Overflow

	Django	Spring Boot	Symfony	ASP.NET Core
Počet vývojářů[a]	8k	8,5k	2k	12k
Počet zodpovězených otázek na Stack Overflow[b]	205k	210k[c]	49k	313k[d]

[a] Z celkových 71802 respondentů
[b] Stav k 16. 1. 2024
[c] včetně výsledků pro Spring
[d] včetně výsledků pro ASP.NET

něj. Málokdy si totiž vývojář vystačí s oficiální dokumentací, a je tak potřeba spoléhat na rady zkušenějších kolegů. Pro porovnání tak uvádím tabulku 3.1, která ukazuje, kolik vývojářů daný framework používalo podle *Stack Overflow Developer Survey* [31] v roce 2023 a kolik se ho na této stránce týkalo otázek.

Z tabulky je patrné, že frameworky Spring Boot a ASP.NET Core jsou mezi vývojáři nejpoužívanější, a zároveň je k nim na komunitním fóru zodpovězeno nejvíce otázek. To ve výsledku znamená nejlepší šanci, že budou na fóru k nalezení i řešení týkající se problémů specifických pro tento projekt.

3.2.1.3 Přehlednost kódu

Obzvlášť ve velkých projektech je důležité, aby bylo možné kód dobře organizovat dle ověřených architektonických a návrhových vzorů a psát kód čitelně pro budoucí úpravy či jiné vývojáře.

Ačkoliv si jazyky jako PHP [32] a Python [33] zakládají na snadném používání a široké škále využití, činí tak na úkor robustní architektury, která na větších projektech přispívá ke spolehlivějšímu běhu a nižší chybovosti kódu. Zároveň jsou tyto jazyky dynamicky typované, což může leckterý čtenář na první pohled považovat za výhodu, poněvadž psaní kódu bez explicitního nominálního typování nebo deklarování tříd bývá rychlejší. Tyto rozdíly jsou však – zejména ve větších projektech – více než vyváženy dobou, o kterou se může protáhnout debugování takového dynamicky typovaného kódu. Zároveň se v rozsáhlém kódu mnohem lépe orientuje novým vývojářům, pokud je typován staticky.

3.2.1.4 Rychlost

Obecně platí, že interpretované jazyky (Python, PHP) jsou ze své podstaty za běhu nejpomalejší, o něco lépe je na tom Java se svým *JVM* [34] a ještě lépe kompilované jazyky (C#). V případě webových aplikací lze toto srovnání prakticky zanedbat, neboť mnohem větší vliv na dobu zpracování požadavku budou mít databázové operace a rychlost sítě [35]. Naopak vliv na rychlost může mít efektivita vícevláknového zpracování, což například Java a .NET

Tabulka 3.2 Volba frameworku: bodové ohodnocení frameworků

	Django	Spring Boot	Symfony	ASP.NET Core
Dokumentace	3	2	4	1
Popularita a podpora	2	3	1	4
Přehlednost kódu	1	3	1	3
Rychlost	1	3	2	3
Bodů celkem	7	11	8	11

technologie zvládají oproti Pythonu lépe [33].

3.2.1.5 Nasazení a provoz

Jelikož bude aplikace provozována na nějaké cloudové platformě, bude třeba ji vhodným způsobem nasadit. V dnešní době je velice snadné téměř jakoukoliv aplikaci uzavřít do kontejneru a takto ji nasadit prakticky do libovolného prostředí, nicméně je dobré poznamenat, že v případě aplikací v ASP.NET Core lze využít integrace s ostatními službami Microsoftu, který poskytuje rozsáhlou a poměrně kvalitní podporu pro nasazení .NET aplikací přímo na svou platformu *Azure*.[6]

3.2.1.6 Závěr průzkumu

Pro usnadnění uceleného srovnání převádím výše popsaná pozorování na číselnou škálu, kdy porovnávaným technologiím přiděluji bodové hodnocení, které je v rámci jednotlivých kritérií relativně srovnává, viz tab. 3.2. Z celkového hlediska se tak nejvhodněji jeví Spring Boot a ASP.NET Core, a to zejména díky popularitě mezi vývojáři, přehlednosti kódu a rychlosti vícevláknového zpracování. Po zvážení bodově nehodnocené úvahy o nasazení a provozu se nakonec přikloním k frameworku ASP.NET Core.

3.2.2 Databáze

Při výběru vhodné databáze již budu počítat s volbou ASP.NET Core jako frameworku pro backend aplikace. Obecně existují dvě hlavní skupiny databází: *relační* a *NoSQL* [36]. NoSQL databáze se zaměřují zejména na řešení problémů, kterými historicky trpěly ty relační – efektivně ukládají velká a nestrukturovaná data a často jsou distribuované. V tomto projektu těchto výhod moc nevyužiji, protože budu pracovat především se strukturovanými daty a často budu chtít pracovat jen s konkrétními částmi ukládaných objektů [37]. Naopak ocením rigidní schéma s kontrolou integritních omezení [38] a v neposlední řadě je zde také nezanedbatelný faktor toho, že jsem díky svému dosavadnímu studiu mnohem lépe obeznámen s prací v SQL databázích než

[6] https://azure.microsoft.com/en-us/resources/developers/net

v NoSQL, a jsem tak už poměrně zvyklý přemýšlet nad daty ve smyslu relací. Z těchto důvodu jsem se tedy rozhodl jít cestou relační databáze.

Co se týče výběru konkrétního databázového stroje, z SQL databází přichází v úvahu řešení jako *MySQL*, *PostgreSQL* a *Microsoft SQL Server*. První dvě zmiňovaná jsou open-source, naopak Microsoft SQL Server je proprietárním softwarem. Jeho nespornou výhodou je to, že je přímo vyvíjen pro .NET aplikace, a tak funguje perfektně dohromady s ORM[7] frameworkem *Entity Framework Core*,[8] který ASP.NET aplikace využívají. Logicky je tak pro tuto kombinaci dostupná největší podpora ze strany Microsoftu a je také nejpoužívanější mezi vývojáři [39]. Zejména kvůli této synergii s ASP.NET jsem se rozhodl pro databázový stroj Microsoft SQL Server.

3.3 Architektura

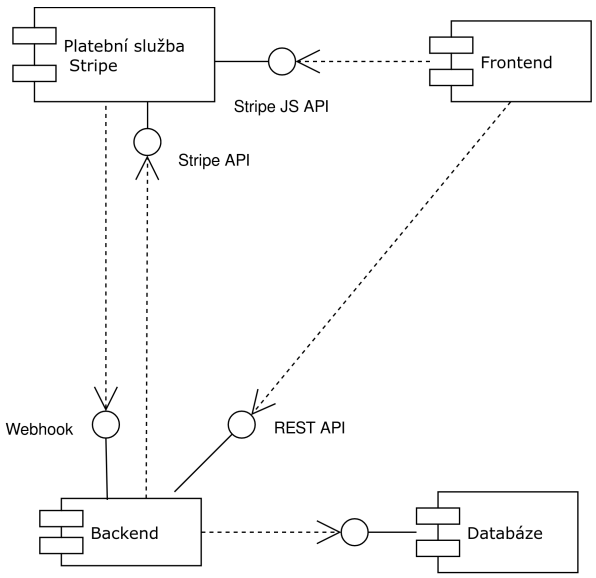
Dále se v této kapitole zaměřím na to, z jakých logických komponent se bude celý systém skládat. Jak již bylo zmíněno v úvodu práce, základními dvěma celky aplikace bude frontend, který má na starost komunikaci s uživatelem aplikace a na kterém pracuje kolega Jeníček, a backend, který zpracovává samotnou logiku aplikace. Dále je zodpovědný za perzistentní ukládání dat do databáze, se kterou komunikuje. Během vývoje bude tato komunikace probíhat lokálně v rámci jednoho zařízení, v produkčním prostředí pak mohou být tyto části nasazeny k libovolným hostitelům, a komunikace tak bude probíhat po síti. Frontend bude s backendem komunikovat prostřednictvím REST API, jehož návrhem se zabývám níže.

Aplikace bude nabízet režim předplatného, v rámci kterého uživatel získá přístup k rozšířeným funkcionalitám aplikace výměnou za pravidelné platby. Ke správě předplatného využiji službu třetí strany, konkrétně službu *Stripe*. Stripe nabízí klasické API, se kterým bude backend komunikovat prostřednictvím příslušné knihovny, a navíc také rozhraní pro Javascript, pomocí kterého bude se službou komunikovat frontend [40]. Zároveň bude Stripe samovolně kontaktovat i backend aplikace, aby ho informoval o důležitých událostech, k čemuž bude připraven koncový bod na zpracovávání tzv. *webhooks*. Ucelený návrh toho, jak bude systém předplatného ve spolupráci s touto službou fungovat, je popsán v dalších podkapitolách.

3.3.1 Architektura backendu

Po rozmyšlení návrhu architektury celého systému přichází na řadu architektura samotné backendové části. Architektura programu popisuje, jak je organizovaný jeho zdrojový kód. Základním požadavkem na architekturu bývá, aby umožňovala co nejsnazší změny, rozšíření a opravy kódu [41]. K tomu je

[7] Object–-relational mapping
[8] https://learn.microsoft.com/en-us/ef/core

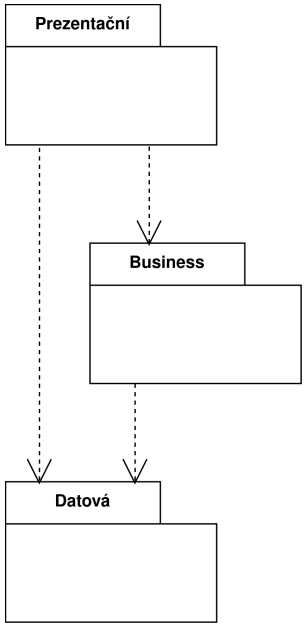


Obrázek 3.1 Diagram komponent celého systému

zejména potřeba, aby byl kód čitelný a aby dodržoval principy vysoké soudržnosti a nízké provázanosti *(high cohesion and loose coupling)* [42]. Díky tomu na sobě nebudou jednotlivé části zbytečně moc záviset a půjde tak opravit či rozšířit jednu část kódu bez toho, aby se vývojář musel orientovat i ve zbytku kódu nebo riskoval, že nedopatřením způsobí chybu v jiném místě.

Vhodným kandidátem je vícevrstvá architektura [43]. Ta rozděluje komponenty programu do několika vrstev, každá z nich pak odpovídá za jinou část celého systému. Důležité je, že komponenty z nižších vrstev nemohou být závislé na vyšších vrstvách. To zajišťuje značnou modularitu a nízkou provázanost, neboť konkrétní implementaci nižší vrstvy by teoreticky mělo být možné jednoduše vyměnit bez toho, aby si toho komponenty vyšší vrstvy všimly.

Dále je potřeba rozmyslet konkrétní počet vrstev. Protože bude aplikace obsahovat i sofistikovanější business logiku a nepůjde jen o základní CRUD rozhraní, budou zapotřebí alespoň tři vrstvy. A to konkrétně *prezentační*, *businessová* a *datová* [44]. V souladu se zásadními akronymy softwarového inženýrství jako *KISS (Keep It Simple Stupid)* a *POGE (Principle Of Good Enough)* zatím nebudu architekturu aplikace více komplikovat, ale pokud by bylo v budoucnu užitečné nějakou vrstvu přidat, mělo by to být možné. Konkrétně ještě třívrstvou architekturu použiji v relaxované verzi, tedy nejen že bude prezentační vrstva záviset na businessové a ta na datové, ale umožním prezentační vrstvě i přímo záviset na datové (obr. 3.2). Tím například umožním prezentační vrstvě přímo využívat datové modely nebo některé služby, které budou z jistého důvodu umístěny v datové vrstvě.



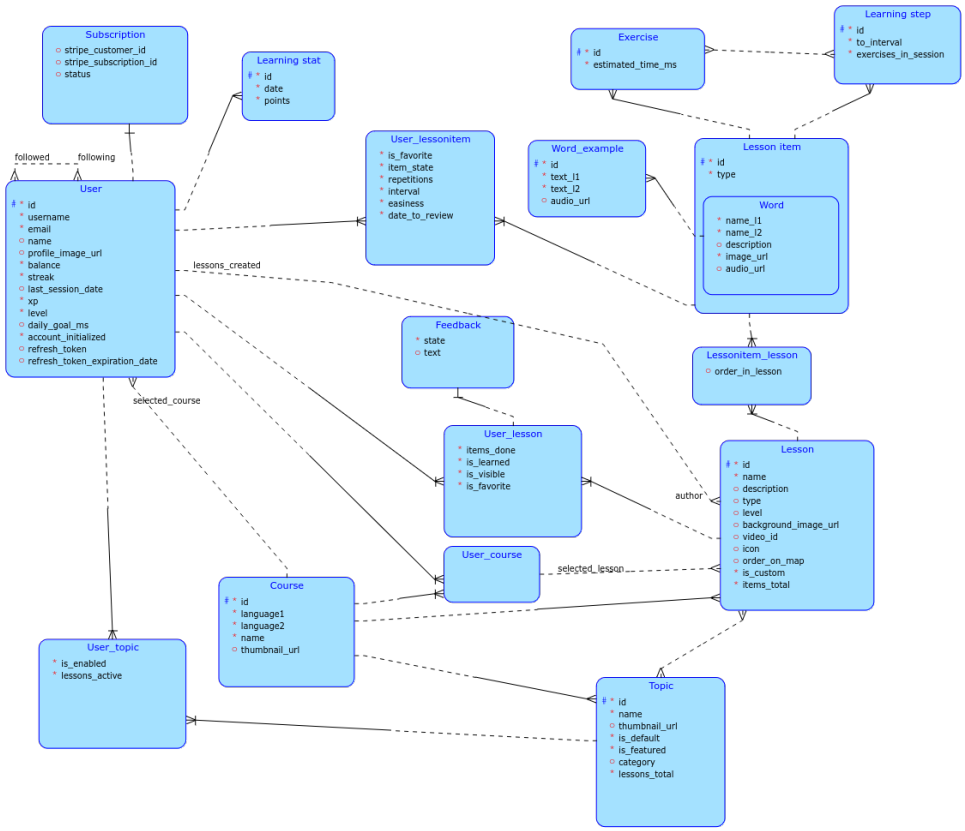
Obrázek 3.2 Diagram závislostí relaxované třívrstvé architektury

3.4 Problémová doména

Zásadním předpokladem pro úspěšný vývoj aplikace je kvalitní modelování domény. Tím je myšlen popis entit a jejich vzájemných vztahů, které společně reprezentují problémový doménový prostor [45]. Prakticky jsem postupoval tak, že jsem na základě funkčních požadavků vytipoval klíčové entity (jako jsou `User`, `Course`, `Lesson`, …) a rozhodl, jak na sebe budou vzájemně navázány. Následně jsem pokračoval přidáváním méně očividných či pomocných entit, které jsou ale pro funkční provázání celého modelu nezbytné, a atributů, které je potřeba u jednotlivých entit evidovat. Na následujících řádkách rozeberu netriviální části výsledného modelu a odůvodním, proč jsou navrženy zrovna takovým způsobem. Vizuálně je problémová doména představena v konceptuálním modelu na obr. 3.3.

3.4.1 Uživatel (User)

Entita uživatele stála na počátku tvorby celého modelu. Kromě atributů souvisejících s identitou uživatele (`username`, `email`) a uživatelovým nastavením (`profile_image_url`, `daily_goal_ms`) budou evidovány i informace o statistikách učení. K tomu kromě atributů jako `streak`, `balance` a `last_session_date` slouží i entita `LearningStat`, jejíchž instancí bude mít typicky každý uživatel přiřazeno více a každá z nich bude uchovávat informace o jednom dni, během kterého uživatel studoval. V neposlední řadě bude moct mít každý uživatel aktivované předplatné (viz dále) a sledovat libovolné množství uživatelů,



Obrázek 3.3 Konceptuální model

k čemu slouží relace `followed-following`.

Uživatel bude muset mít vazbu na všechny další klíčové entity, neboť bude potřeba kupříkladu evidovat, které kurzy a lekce uživatel studuje a pro každý takový vztah udržovat další atributy, které samotný vztah dále parametrizují – uživatel bude například mít možnost označit lekci jako oblíbenou.

3.4.2 Kurz (Course)

Jak už ostatně stanovuje i název samotné práce, aplikace se bude zaměřovat na učení se angličtiny, neboť výukové metody, které bude využívat, nemusí nutně fungovat pro všechny jazyky. Nicméně by mělo být pro administrátory možné přidávat i další kurzy, které by se týkaly dalších jazyků nebo byly nějak specificky zaměřeny. Povinnými atributy každého kurzu bude kód mateřského jazyka a nově učeného jazyka. Pro identifikaci těchto jazyků bude aplikace využívat označení `language1`, resp. `language2` vycházející z běžné praxe na poli studia učení jazyků.

Nejvýše jeden ze svých zapsaných kurzů může mít uživatel označený jako aktuálně vybraný (`selected_course`) - díky tomu bude moct frontend uživa-

teli ihned nabízet informace o kurzu, který má aktuálně rozstudovaný.

3.4.3 Lekce (Lesson)

Lekce je zásadním pojmem pro výukový model aplikace. Každá lekce bude muset náležet k právě jednomu kurzu, a kromě mnoha triviálních atributů u ní bude evidováno pořadí na studijní mapě (`order_on_map`). V rámci kurzu budou lekce lineárně seřazeny na tzv. studijní mapu, po které bude uživatel postupovat – aby bylo administrátorům umožněno průběžně přidávat nové lekce i mezi již existující a nebylo nutné měnit jejich pořadí, bude mít index `order_on_map` formát čísla s plovoucí desetinnou čárkou, aby mohl nabývat hodnot z hustě uspořádané množiny racionálních čísel $\mathbb{Q}$.

Toto pořadí na mapě plní důležitou roli pro určení aktivní lekce daného uživatele (viz dále v textu), se kterou pracuje algoritmus učení.

Každou lekci bude moct uživatel označit jako oblíbenou a přidat k ní zpětnou vazbu (`Feedback`) ve formě textového komentáře a příznaku `LIKE/DISLIKE`. Také bude umožněno každou lekci jednotlivě skrýt (`is_visible`), čímž přestane být zobrazována na studijní mapě a nebude brána v úvahu při učení.

Dále bude možné vytvářet i vlastní uživatelské lekce, které se budou v zásadě chovat jako ty běžné, jen budou odlišeny příznakem `is_custom`, budou mít uvedeného svého autora a nebudou zahrnuty do studijní mapy.

3.4.4 Téma (Topic)

Témata slouží jako zastřešující množiny lekcí. Hlavní motivací je, aby uživatel nemusel jednotlivě vypínat všechny lekce, které ho nezajímají, pokud se týkají všechny jednoho konkrétního tématu, a mohl si jednoduše rovnou zvolit, která témata ho zajímají a která se naopak učit nechce.

Některá témata budou od začátku administrátory doporučená (`is_featured`) a některá budou automaticky zapnutá (`is_default`). Uživatel si bude moct jakékoliv téma sám zapnout nebo vypnout (`is_enabled`), což ve výsledku ovlivní viditelnost jednotlivých lekcí v něm obsažených.

Jednou z výzev během návrhu bylo vymyslet, jak se budou zapínání témat a změny viditelnosti lekcí vzájemně ovlivňovat, neboť je mezi těmito entitami umožněn vztah *many-to-many (M:N)*. Nakonec se jako nejrozumnější ukázalo chování, které splňuje tyto invarianty:

- Téma se všemi skrytými lekcemi je vypnuté.
- Téma se všemi viditelnými lekcemi je zapnuté.
- Lekce v alespoň jednom zapnutém tématu je viditelná.
- Lekce v pouze vypnutých tématech je skrytá.

Toho se docílí následujícím přístupem. Jestliže uživatel zapne téma, zviditelní se všechny lekce v něm. Jakmile jej vypne, skryjí se všechny lekce, jejichž všechna ostatní témata jsou již také vypnutá. Pokud uživatel skryje konkrétní lekci, vypnou se všechna témata, v nichž toto byla poslední viditelná lekce. Jakmile konkrétní lekci zviditelní, zapnou se témata, v nichž jsou všechny lekce viditelné. Z tohoto chování tedy vyplývá, že pokud jsou v tématu viditelné pouze některé lekce, ale ne všechny, může být jak zapnuté, tak vypnuté, podle toho, do kterého stavu bylo vlivem výše uvedených pravidel naposledy uvedeno.

Aby bylo takové chování možné, je potřeba evidovat ke každému tématu celkový počet lekcí (`lessons_total`) a ve vztahu k uživateli také počet aktuálně viditelných lekcí (`lessons_active`) – oboje by samozřejmě bylo možné počítat dynamicky, ale poměrně rychle by se to mohlo stát výpočetně náročnou operací.

3.4.5 Lesson item

K zavedení této entity vede požadavek, že mají lekce jednak umožňovat seskupování slovíček, a jednak mají sloužit jako samostatné učební celky zaměřující se např. na konkrétní gramatický jev nebo na mluvení v konkrétní situaci. Ukazuje se, že tento problém úzce souvisí s fundamentální otázkou, co bude v učicím algoritmu považováno za elementární jednotky učené látky, tedy již dříve zmiňované *itemy*. U slovíčkových lekcí to budou samotná slovíčka (*Word, pl. Vocabulary*), zatímco ostatní lekce budou item obsahovat pouze jeden, který bude odpovídat látce celé lekce. Aby potom bylo možné se všemi typy lekcí pracovat stejným způsobem, zavádím entitu `Lesson item`, která tuto roli plní. Slovíčko pak bude jejím podtypem, díky čemuž bude možné se všemi lesson itemy pracovat polymorfně.

3.4.5.1 Slovíčko (Word)

Slovíčko bude kromě triviálních atributů jako překlad v obou jazycích (`name_l1` a `name_l2`) obsahovat také příklady použití ve větách (`Word example`), které budou uživateli k dispozici pro lepší pochopení jejich významu.

3.4.6 Cvičení (Exercise)

Samotné učení a procvičování lesson itemů bude probíhat prostřednictvím cvičení. Cvičení bude patřit k právě jednomu lesson itemu a samo o sobě bude abstraktní entitou, aplikace tedy nebude pracovat s instancemi samotného `Exercise`. Jednotlivé typy cvičení se mohou ve svých atributech značně lišit, vždy budou ale obsahovat nějakou formu zadání a správné odpovědi, podle které bude moct být hodnocena uživatelova znalost příslušného lesson itemu.

Tabulka 3.3 Datová struktura cvičení typu *FillInSentence*

Atribut	Datový typ	Popis
TextL1	String	věta v jazyce L1
TextL2	String	věta v jazyce L2
BlankIndices	List[Integer]	indexy slov chybějících ve větě v L2
Options	List[String]	nabídka slov na doplnění do věty
ImageUrl	String	případný ilustrační obrázek

Tabulka 3.4 Datová struktura cvičení typu *Listening*

Atribut	Datový typ	Popis
AudioUrl	String	nahrávka v jazyce L2
QuestionL2	String	otázka k zodpovězení v jazyce L2
AnswerL2	String	správná odpověď na otázku
ImageUrl	String	případný ilustrační obrázek

Jediným společným atributem získaným od rodičovského typu bude odhad časové náročnosti cvičení (`estimated_time_ms`), která by měla umožňovat co možná nejpřesnější přiblížení k uživatelovu dennímu cíli učení při sestavování studijní relace (*study session*).

Tabulky 3.3 a 3.4 popisují strukturu dvou vybraných typů cvičení *FillInSentence*, resp. *Listening*. V diagramu 3.3 nejsou v zájmu zachování předhlednosti jednotlivé typy znázorněny.

Databázový model vytvořený na základě tohoto návrhu je ztvárněn na obr. 3.4. Dědičnost – v konceptuálním modelu modelovaná vztahem ISA – je do databáze převedena podle vzoru *table-per-hierarchy* [46]. Všechny podtypy `Exercise` tak sdílí jednu tabulku, do které je přidán sloupec `Discriminator`, který od sebe podtypy na jednotlivých řádcích odlišuje. V každém řádku jsou pak vyplněné pouze relevantní sloupce.

3.5 Návrh REST API

Jak je zřejmé z návrhu architektury, s backendem aplikace bude vnější svět komunikovat přes API *(Application Programming Interface)*. Konkrétně bude aplikace používat REST API (někdy také RESTful API), což je způsob návrhu rozhraní na základě šesti návrhových principů REST (*Representational State Transfer*) [47]. Nejzásadnějšími principy pro samotný návrh jsou pak:

- Jednotnost rozhraní – požadavky na stejný zdroj by měly vypadat stejně, nehledě na to, odkud požadavek přichází.
- Oddělení serveru a klienta – server a klient by na sobě měli být kompletně nezávislí a neměli by spolu interagovat jinak než přes požadavky na příslušné zdroje.

- Bezstavovost – každý požadavek je nezávislý na ostatních, musí v sobě obsahovat všechny potřebné informace pro jeho zpracování a server pro klienta nezakládá žádnou relaci (*session*).

Pro úplnost dalšími zbývajícími principy REST jsou: možnost ukládání do mezipaměti (*cacheability*), vrstvená systémová architektura a volitelně kód na požádání.

3.5.1 Obecné koncové body

Pro přístup k celé kolekci dostupných objektů bude API používat URI (*Uniform Resource Identifier*) ve tvaru množného čísla názvu daného objektu. Tedy například zdroj pro získání všech kurzů bude označen jako `/courses`. K získání celého objektu konkrétního kurzu je pak URI rozšířeno ještě o příslušný identifikátor, tedy `/courses/:courseId`.

3.5.2 Uživatelské koncové body

Většina koncových bodů potřebuje vracet data v závislosti na konkrétním uživateli, který o ně požádá. Může jít přímo o výpis kurzů, které má uživatel zapsané nebo informace o nastavení jeho profilu, ale také o výpis všech lekcí v kurzu s příznaky závislými na uživateli, jako jsou viditelnost a oblíbenost lekce.

Při určování URI těchto koncových bodů jsem zvažoval více variant. Ta, která by pravděpodobně nejstriktněji odpovídala požadavku REST na jednotnost rozhraní, znamená použití předpony `/users/:userId`, čímž skutečně dojde k vytvoření unikátního zdroje pro každého uživatele. Není to však moc praktický přístup pro implementaci, neboť je při přijetí každého požadavku ještě potřeba ověřovat, že identifikátor v URI odpovídá uživatelově identitě, nehledě na to, že se tím celý URI v zásadě zbytečně prodlužuje.

Proto jsem se nakonec inspiroval REST API platformy *GitHub* [48], kde tyto uživatelské koncové body obsahují pouze předponu `/user`. Nastavení aktuálně přihlášeného uživatele se tak nachází na zdroji `/user/settings`, seznam slovíček v daném kurzu s informací o oblíbenosti na `/user/courses/:courseId/vocabulary` a seznam témat v daném kurzu se stavem zapnutí/vypnutí na `/user/courses/:courseId/topics`.

3.5.3 Změna objektu

Bude-li chtít uživatel některý objekt změnit, využije k tomu HTTP metodu `PATCH` a v těle požadavku uvede nové hodnoty atributů, které bude chtít změnit. Na zdroji `/user/settings` tak bude moct upravit atributy `name`, `username` a `daily_goal`. V případě koncového bodu `/user/courses/:courseId/lessons/:lessonId`, který vrací kromě atributů dané lekce i informace o její viditelnosti

a oblíbenosti z pohledu uživatele, pak bude metoda `PATCH` sloužit k úpravě těchto uživatelsky specifických vlastností, zatímco k úpravě názvu, popisu a dalších atributů vlastních uživatelských lekcí bude využívána metoda `PUT`.

3.5.4 Přidání a odebrání objektu v rámci kolekce

Důležitou operací, kterou bude muset REST API umožňovat, je přidávání objektů do kolekce, stejně jako jejich případné odebírání. Tím je myšleno například přidání a odebrání tématu v rámci daného kurzu přes koncový bod `/user/courses/:courseId/topics/:topicId`. Jednotlivé operace budou prováděny pomocí metod `PUT`, resp. `DELETE` s prázdným tělem požadavku. Stejným způsobem bude probíhat i zahájení a ukončení sledování uživatele, což lze také chápat jako přidání do, resp. odebrání z kolekce sledovaných uživatelů (přes koncový bod `/user/following/:followed_user_id`).

3.5.5 Filtrování

U některých koncových bodů lze očekávat, že budou uživateli na požadavky typu `GET` vracet poměrně početné kolekce objektů. Je tedy žádoucí uživateli umožnit přidat kritéria, podle kterých budou výsledky jeho dotazu filtrovány. Filtrovací kritéria budou předávána prostřednictvím parametrů dotazu (*query parameters*). Na konec URI definující příslušný zdroj tak bude moct uživatel přidat libovolný počet kritérií ve tvaru *klíč–hodnota* a tím omezit výsledky dotazu pouze na ty objekty, které všechny podmínky splňují. Toto filtrování bude dostupné například pro koncový bod `/user/courses/:courseId/lessons`, který vrací všechny lekce v daném kurzu. Filtrovat bude možné jak podle atributů samotné lekce jako `type` a `level`, tak podle jejich vztahu k uživateli, tedy booleovských hodnot `visible` a `favorite`.

3.5.6 Stránkování

K zamezení zahlcení uživatele nadměrným počtem objektů v odpovědi slouží také stránkování. Z několika možných přístupů zvolím tzv. *page-based pagination* [49], které využívá hodnoty `page` a `limit` předávané prostřednictvím parametrů dotazu. Výsledky dotazu tak budou rozdělené do stránek, které uživatel bude moct získávat postupně, a jejich velikost nastaví prostřednictvím parametru `limit`.

3.5.7 Zabezpečení koncových bodů

K jednotlivým koncovým bodům bude potřeba nastavit vhodná oprávnění přístupu. Přístup bude aplikace udělovat na základě uživatelovy role. Některé koncové body budou určené pouze administrátorům aplikace – ty budou sloužit zejména k přidávání výukového materiálu. Většina ostatních koncových

bodů bude přístupná všem přihlášeným uživatelům, vyjma těch, které budou zprostředkovávat funkcionality dostupné pouze prémiovým uživatelům.

Několik koncových bodů bude umožňovat anonymní přístup i nepřihlášeným uživatelům. To konkrétně ty, které budou zprostředkovávat služby jako registrace uživatele, jeho přihlášení a obnova hesla.

3.6 Autentizace a autorizace uživatelů

Pro správu uživatelských účtů a s tím související autentizaci a autorizaci uživatelů existuje několik řešení, z nichž je možné vybírat. Jednou z možností jsou zcela externí služby třetích stran, které komunikují přímo s uživatelem a samy odpovídají za uchovávání uživatelských dat, která by od nich vlastní aplikace musela získávat přes jejich API. Po prostudování a zvážení jednoho takového řešení *Auth0*[9] jsem dospěl k závěru, že nenabízí dostatečnou volnost v přizpůsobení celého přihlašovacího procesu vlastním potřebám, takže by se spíše musela aplikace přizpůsobit autentizační službě.

3.6.1 Knihovna ASP.NET Core Identity

Z toho důvodu jsem se rozhodl pro řešení, které je pro ASP.NET aplikace poměrně běžné, a tou je vlastní správa uživatelů přímo v databázi aplikace za pomoci knihovny *ASP.NET Core Identity* [50]. Ta zajistí vygenerování tabulek v databázi pro správu uživatelů a rolí a nabízí rozhraní pro vytvoření uživatele, ověření hesla a vytvoření tokenů pro obnovu e-mailu a hesla. V nejnovější verzi .NET 8 lze za pomocí Identity vygenerovat dokonce celé koncové body obsluhující autentizační a autorizační služby, což kromě základní konfiguraci nevyžaduje žádnou dodatečnou práci [51]. Bohužel však nelze upravit, které koncové body budou vytvořeny a jak budou vypadat jimi přijímané požadavky a jejich odpovědi. To je poměrně zásadní nedostatek, a z toho důvodu jsem se rozhodl implementovat samotné koncové body dle vlastního návrhu s využitím služeb nižších vrstev, které Identity nabízí. K tomu navíc bude potřeba samostatně generovat autentizační tokeny, neboť to Identity bez vygenerování kompletních koncových bodů vůbec nenabízí.

3.6.2 Generování tokenů

Po úspěšném přihlášení heslem je potřeba vygenerovat uživateli *access token* [52], který prokazuje uživatelovu identitu vůči aplikaci a je podepsán jejím tajným klíčem, což zaručuje jeho integritu a autenticitu. Tento token má určitou dobu platnosti, po kterou jej může klient posílat se svými požadavky a bude mu na základě něj poskytován přístup k aplikaci. Vyvinu tedy vlastní imple-

[9] https://auth0.com

mentaci generování tohoto tokenu ve formátu *JWT (JSON Web Token)* [53] – následnou autentizaci na základě tokenu již zvládá knihovna Identity sama.

Druhým typem tokenu, který bude v aplikaci využit, je *refresh token* [52]. Jelikož mívá access token poměrně krátkou dobu platnosti (v řádu minut, maximálně hodin), nebylo by pohodlné, aby po jejím vypršení musel uživatel vždy znovu zadávat heslo. Refresh token uživatel získá také po přihlášení, má ale delší dobu platnosti (řádově dny až měsíce) a uživatel, resp. frontend na pozadí, ho posílá za účelem získání nového access tokenu, pokud je již ten starý neplatný nebo se tak brzy stane. Pro jeho implementaci je potřeba vygenerovat dostatečně dlouhý řetězec kryptograficky bezpečným pseudonáhodným generátorem, který bude následně uložen do databáze společně s datem vypršení platnosti. Ačkoliv to dle specifikací není nutné, bude token před uložením ještě hešován, aby při případném úniku dat z databáze nebyla kompromitována uživatelova bezpečnost až do doby vypršení platnosti tokenu. Jakmile uživatel refresh token jednou použije, musí být kromě nového access tokenu vygenerován a uložen nový refresh token, čímž dojde k zneplatnění původního tokenu a alespoň částečné mitigaci útoků přehráním (*replay attack*).

3.6.3 Zapomenuté heslo

V případě, že uživatel zapomene své heslo, bude mu na e-mail, kterým se registroval, zaslán odkaz pro obnovu hesla. Ten jej odkáže na stránku frontendu a bude obsahovat unikátní token vygenerovaný knihovnou Identity, který následně frontend pošle v požadavku backendu společně s uživatelovým novým heslem. Poté, co knihovna tento token ověří, se uživateli nastaví nové heslo.

3.7 Správa předplatného

3.7.1 Platební služba Stripe

Ke správě předplatného (FR15) bude aplikace využívat službu *Stripe*. Tu jsem zvolil zejména z toho důvodu, že je z konkurenčních řešení nejpoužívanější – spolupracuje například se společnostmi Google, Amazon, Spotify a Shopify [54] – je tak velice důvěryhodná, což je pro službu zpracovávající platby zásadní požadavek. Zároveň nabízí velice rozsáhlou a přehlednou dokumentaci [55] s ukázkami integrací s mnoha různými technologiemi a umožňuje snadné přizpůsobení vlastním potřebám.

3.7.2 Služby v rámci předplatného

Převážná většina funkcionalit aplikace bude uživateli dostupná zcela zdarma bez jakýchkoliv závazků, jako zadávání platebních údajů nebo časového omezení. Pokud si uživatel aktivuje předplatné, stane se prémiovým uživatelem a bude mu zpřístupněné i vytváření vlastních lekcí (FR8) a možnost studovat

pokročilejší lekce (úrovně C1 a C2). Každý uživatel bude mít možnost si nejprve předplatné vyzkoušet na omezenou dobu, pochopitelně však pouze jednou za celou dobu používání aplikace.

3.7.3 Proces tvorby a správy předplatného

Samotnou položku předplatného – dle Stripe terminologie *price* – s nastavenou frekvencí plateb a samotnou částkou lze vytvořit přímo ve webovém rozhraní Stripe, aplikace se pak na ni v požadavcích může rovnou odkazovat. Stripe si potřebuje vést vlastní evidenci uživatelů, před vytvořením samotného předplatného pro naše uživatele je potřeba vytvořit objekt uživatele ve Stripe. Jedním z možných přístupů by bylo vytvářet Stripe uživatele automaticky ihned po jeho registraci v aplikaci, ale protože nepředpokládám, že bude mít obzvlášť v prvních měsících používání značná část uživatelů zájem o předplatné, budu je ve Stripe vytvářet až po objednání jejich prvního předplatného.

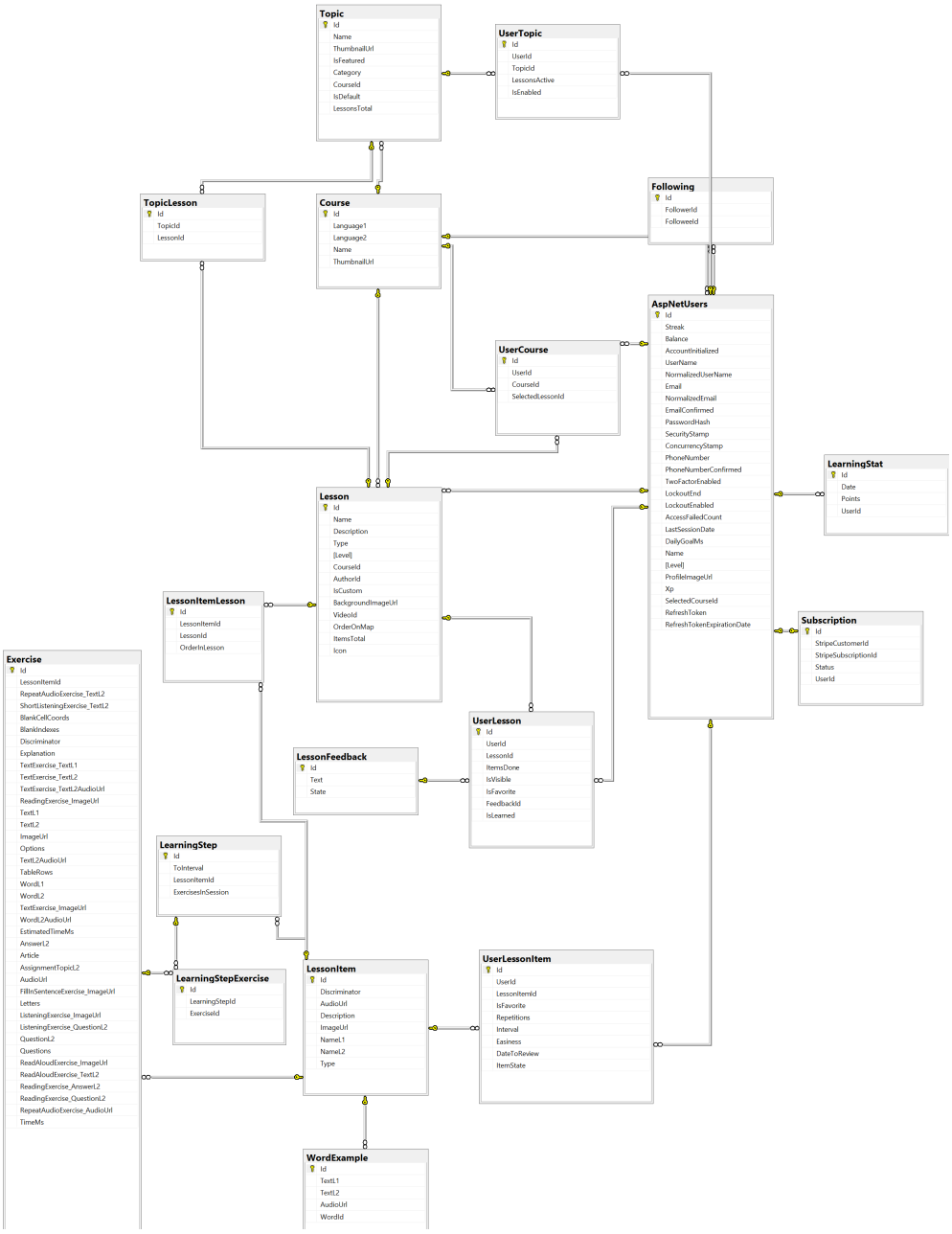
Pokud bude mít uživatel zájem o objednání předplatného se zkušební dobou, není potřeba od něj vyžadovat platební údaje. Okamžitě po objednání mu tak bude umožněn přístup k prémiovému obsahu. Objednat placené předplatné může uživatel již během zkušební doby – to je doporučené zejména proto, aby si uživatel zajistil plynulý přístup k obsahu bez přerušení, které by s ukončením zkušební doby mohlo nastat. V takovém případě bude vyzván k poskytnutí platební metody, která bude po konci zkušební doby použita k hrazení předplatného. V opačném případě dojde s vypršením zkušební doby k pozastavení předplatného a uživatel může následně kdykoliv požádat o zahájení placeného plánu.

Po objednání placeného předplatného bude uživatel rovnou požádán o uhrazení platby za první období. Po úspěšném přijetí platby bude předplatné aktivováno.

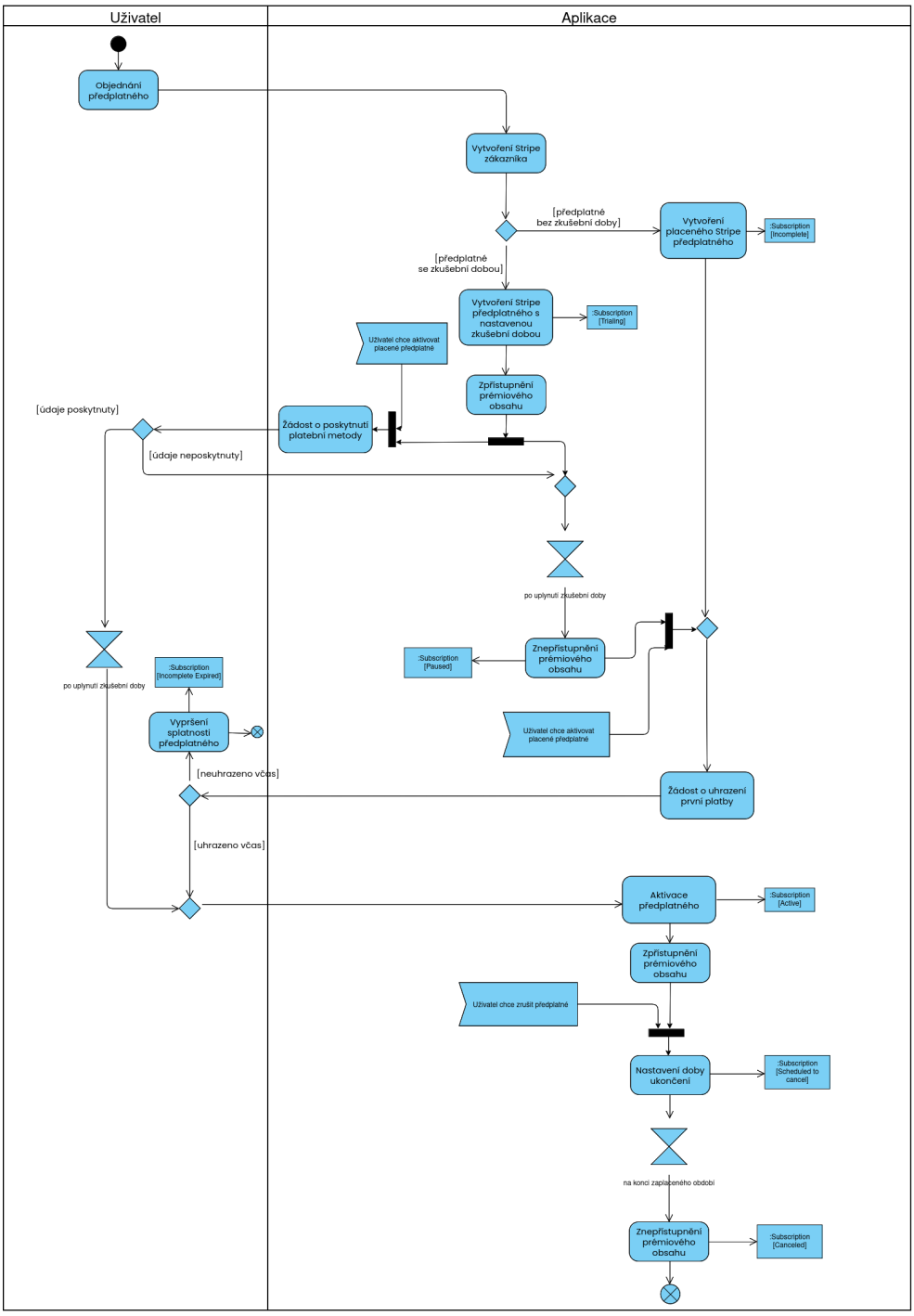
Uživatelé také budou moct své předplatné kdykoliv zrušit. V takovém případě dojde k ukončení předplatného, a tedy i přístupu k prémiovému obsahu na konci posledního období, za které uživatel ještě zaplatil.

Ke všem těmto operacím provedeným na žádost uživatele bude aplikace využívat Stripe API. Stripe bude naopak aplikaci informovat o změnách stavu každého předplatného, aby bylo možné udělovat a odpírat přístup k prémiovému obsahu na základě zpracování plateb. K tomu bude aplikace implementovat *webhook endpoint* [56], na který bude Stripe posílat události obsahující informace o změnách.

Výše popsaný proces tvorby a navazujícího životního cyklu předplatného je demonstrován diagramem na obr. 3.5.



Obrázek 3.4 Model databázového schématu



Obrázek 3.5 Diagram aktivit životního cyklu předplatného

Kapitola 4
Implementace

Po vypracování všech potřebných návrhů jsem přešel k samotné implementaci. Na virtuální nástěnce *Kanban* jsme s kolegou Jeníčkem vytvořili úkoly, které odpovídaly klíčovým navrženým koncovým bodům API. Těmto úkolům jsme přiřadili priority odvozené z funkčních požadavků, které jednotlivé koncové body realizují. Také jsme navrhli strukturu požadavků, které budou přes tyto koncové body posílány. Následně jsme tuto specifikaci iterativně upravovali na základě společných rozprav.

Implementoval jsem tedy postupně jednotlivé koncové body od nejvyšší priority. Nejprve jsem se věnoval registraci a autentizaci uživatelů, poté organizaci látky do kurzů, lekcí a témat a následně procesu učení. Všechny změny v kódu jsem průběžně verzoval a replikoval na vzdáleném úložišti platformy *GitHub*.

Dále se v této kapitole zaměřím na několik problémů, na které jsem během vývoje narazil, a popíšu jejich výsledné řešení. Jde zejména o organizaci kódu a návrhové vzory, implementaci učicího algoritmu, mapování objektů a zpracování výjimek.

4.1 Vývojové prostředí

Ihned na počátku implementování jsem narazil na problém, jak během vývoje spouštět databázi a k ní připojený backend v lokálním prostředí. Jelikož je .NET proprietární technologií společnosti Microsoft, obvykle se doporučuje k vývoji použít operační systém Windows s vývojovým prostředí Visual Studio, které jsou rovněž dílem této společnosti. Databázový stroj Microsoft SQL Server, zvolený v návrhové fázi, ale poněkud překvapivě nezvládá provoz nad souborovým systémem nových Windows 11. Ten totiž na některých novějších discích vytváří sektory o velikosti větší než 4 KB, zatímco SQL Server umí pracovat nejvýše s velikostmi 4 KB [57].

Tuto nepříjemnost jsem se rozhodl vyřešit kontejnerizací databáze přes

technologii *Docker*.[1] To databázovému stroji umožní přistupovat k souborovému systému počítače virtualizovaně prostřednictvím tzv. *volumes*, čímž se problém s velikostí sektorů vyřeší.

Abych nemusel při každém spuštění aplikace v lokálním prostředí startovat kontejner s databází ručně, přidal jsem podporu Dockeru i pro samotnou aplikaci v prostředí Visual Studio. To dále umožňuje zavést orchestraci s využitím nástroje *Docker Compose*, čímž dojde k propojení kontejneru aplikace a databáze. Díky tomu je Visual Studio schopné již při svém spuštění sestavit orchestrované kontejnery a propojení s databází tak funguje naprosto automaticky, bez nutnosti manuálního spouštění a jakéhokoliv jiného zasahování do kontejnerů.

4.2 Organizace kódu a vkládání závislostí

Celý zdrojový kód je rozdělený do tří projektů dle jednotlivých vrstev třívrstvé architektury, které jsou dohromady spojeny do jednoho řešení (*solution*). Projekt s prezentační vrstvou (projekt *API*) obsahuje kromě balíčků se samotným kódem aplikace i konfigurační soubory `appsettings.json` a také balíček spravující vkládání závislostí.

Za vkládání závislostí (*dependency injection*) je označován návrhový vzor, který realizuje princip *Inversion of Control* [58]. Ten umožňuje strukturovat kód tak, aby byl vysoce modulární, testovatelný a udržitelný, a to díky zavedení vyšší úrovně abstrakce, na které jednotlivé konkrétní implementace tříd mohou záviset, namísto přímých závislostí mezi sebou.

Aplikace tento návrhový vzor implementuje pomocí abstrahování veřejných metod do rozhraní, registrováním těchto rozhraní jako tzv. služeb (*service*) a vkládáním těchto služeb jako závislostí do konstruktoru třídám, které je potřebují využívat. Registrování služeb provádím pomocí extension metod z oficiální knihovny `Microsoft.Extensions.DependencyInjection`, kde jim zároveň přiřazuji konkrétní implementace. Vždy je přitom potřeba určit jejich životnost, kterou jsem zpravidla zvolil jako *scoped*, tedy omezenou na daný požadavek klienta. To z toho důvodu, že taková je životnost i databázového kontextu z *Entity Framework*[2] a dle doporučení Microsoft [58] by služby daného životního cyklu neměly používat kontext databáze s dobou života kratší než životnost služby.

Registrované služby jsou do tříd, které je využívají, vkládány jako parametry konstruktoru. Přitom musí platit, že třídy z projektů nižší vrstvy nemohou záviset na službách, které jsou definované ve vyšších vrstvách, aby byla dodržena pravidla vrstvené architektury.

[1] https://www.docker.com

[2] Framework, který zajišťuje komunikaci s databází a ORM mapování.

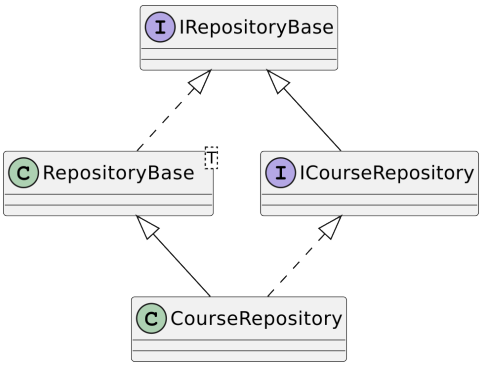
4.3 Návrhové vzory

Kromě *dependency injection* využívá aplikace několik dalších návrhových vzorů, které přispívají k přehlednější organizaci kódu, jeho modularitě a udržitelnosti.

4.3.1 Repository

Návrhový vzor Repository zprostředkovává komunikaci mezi businessovou a datovou vrstvou tak, aby aplikační logiku co nejlépe odstínil od detailů přístupu k datům v databázi [59]. Jednotlivé třídy implementující tento vzor – *repozitáře* – poskytují rozhraní, které umožňuje čtení i zápis dat nehledě na konkrétní implementaci perzistentního úložiště.

V aplikaci konkrétně využívám tento vzor ještě ve variantě s generickým repozitářem, který definuje základní CRUD operace, které jsou pro všechny entity implementovány stejně. Pro všechny zásadní entity z domény jsou pak implementovány samostatné repozitáře, které tento generický rozšiřují o metody, které zajišťují operace specifické pro danou entitu. Tyto metody typicky zapouzdřují jednotlivé dotazy v jazyce *LINQ*,[3] které Entity Framework pro komunikaci s databází překládá do SQL.



Obrázek 4.1 Diagram závislostí třídy `CourseRepository`

4.3.2 Unit of Work

Repozitáře poskytují abstrakci nad dotazy pracující s konkrétní entitou, obvykle je však v rámci zpracování jednoho požadavku potřeba provést takových dotazů větší množství. Je velice vhodné všechny tyto dotazy provádět v rámci transakce, aby bylo možné atomické zpracování – to umožňuje právě návrhový vzor Unit of Work [60].

V aplikaci ho realizuji rozhraním `IUnitOfWork`, resp. jeho implementací `UnitOfWork`. Ta zajišťuje správnou inicializaci všech repozitářů a poskytuje

[3] https://learn.microsoft.com/en-us/dotnet/standard/linq

veřejný přístup k jejich rozhraní. Navíc nabízí metodu `SaveChanges()`, která potvrdí změny v databázovém kontextu – transakční commit. Prakticky je tedy rozhraní `IUnitOfWork` injektováno do služeb businessové vrstvy, které přes něj přistupují k metodám jednotlivých repozitářů a při dokončení transakce na něm volají metodu `SaveChanges()`. Skončí-li zpracování dotazu s chybou, je vyhozena výjimka a k potvrzení transakce nedojde, žádná z provedených operací v rámci tohoto požadavku se do databáze nezapíše.

4.3.3 Role Guard

Je zřejmé, že aplikace musí uživatelům poskytovat přístup pouze ke jejich datům, nikdy k datům ostatních uživatelů. K tomu je zapotřebí mít po celou dobu zpracovávání požadavku přehled o identitě uživatele, který jej vytvořil, a buď přístup ke koncovému bodu úplně odepřít, nebo vracet pouze ta data, ke kterým má daný uživatel přístup.

V případě, že uživatel přistupuje ke koncovému bodu, který přímo ve svém URI obsahuje `userId`, jež má odpovídat identifikátoru uživatele, lze provést porovnání rovnou v prezentační vrstvě. Identifikátor uživatele je součástí access tokenu, který je možné snadno získat ze třídy `HttpContext` dostupné pro každý požadavek. V případě jeho neshody s hodnotou v URI může být uživateli přístup rovnou odepřen formou vrácení odpovědi s vhodným stavovým kódem.

Pokud se identifikátory shodují, je potřeba informaci o uživatelově identitě dát k dispozici i dalším vrstvám, které se budou podílet na zpracování požadavku, aby mohly zajistit přístup k odpovídajícím datům. K tomuto kroku je nutné přistoupit i v případě, který je v aplikaci mnohem častější, kdy koncové body v URI žádný identifikátor uživatele neobsahují. Pro zajištění této funkcionality používám návrhový vzor RoleGuard.

RoleGuard v mém pojetí spočívá v rozhraní třídy `IRoleGuard`, které je definované v datové vrstvě, a třídy `RoleGuard`, která toto rozhraní implementuje na úrovni prezentační vrstvy. Rozhraní poskytuje read-only přístup k vlastnostem `User` a `Roles`, které obsahují objekt autentikovaného uživatele, resp. jeho rolí v rámci aplikace. Toto rozhraní mohou využívat ostatní třídy na úrovni datové vrstvy (repozitáře), které tak mohou informace o uživatelově identitě zohlednit již na úrovni komunikace s databází, a nedochází tím k narušení závislostí mezi vrstvami v rámci třívrstvé architektury.

Implementace `RoleGuard` v prezentační vrstvě pak využívá třídu `HttpContext`, která zapouzdřuje všechny informace o konkrétním požadavku. Poté, co dojde k validaci tokenu knihovnou *ASP.NET Core Identity*, získá z něj `RoleGuard` identifikátor uživatele a na základě něj pak příslušný objekt uživatele a jeho role přiřadí odpovídajícím vlastnostem.

4.4 Implementace učicího algoritmu

Klíčovou částí celé aplikace je řízení uživatelova učení. Na nejvyšší úrovni většinu souvisejících funkcionalit zprostředkovává koncový bod `/user/courses/:courseId/study-session`, který na požadavek typu GET pro uživatele zajistí sestavení uspořádané množiny cvičení, která by měl uživatel splnit – v aplikaci ji nazýváme *study session*. Volitelně může být v požadavku formou parametru specifikovaná i konkrétní lekce, které se má study session týkat. Pokud se tak nestane, sestaví se study session pro denní učení.

V požadavku typu POST pak uživatel zasílá informace o tom, která cvičení zrovna dokončil a s jakým úspěchem, a aplikace je následně zpracovává a vyhodnocuje. V této podkapitole dále rozeberu, jak jsou tyto funkcionality implementovány.

4.4.1 Uchovávání postupu uživatele

Pro řízení celého učicího procesu je nezbytné uchovávat, jak uživatel postupuje při učení jednotlivých lesson itemů. Pro každý vztah lesson itemu a uživatele tak aplikace eviduje tyto hodnoty související s postupem učení:

- `item_state` – stav učení itemu: je buď `NEW`, nebo `REVIEW`
- `repetitions` – počet opakování itemu, tedy kolikrát již uživatel viděl některé z jeho cvičení
- `interval` – interval mezi posledním a následujícím opakováním
- `date_to_review` – ideální datum, kdy by se měl item zopakovat
- `easiness` – E-Faktor v kontextu algoritmu SM-2

4.4.2 Sestavení study session

Nehledě na to, zda jde o učení konkrétní lekce nebo denní učení, algoritmus se vždy pokusí vybrat cvičení tak, aby celková doba jejího trvání odpovídala časovému cíli. Ten je pro denní učení zvolen uživatelem v rámci jeho nastavení, pro konkrétní lekci jde o danou konstantu. Doba trvání session je pak počítána jako součet odhadů časových náročností jednotlivých cvičení (`estimated_time_ms`).

V konfiguraci aplikace je zároveň dán poměr, kolik času v každé session by v ideálním případě mělo být věnováno novým lesson itemům, které uživatel ještě nikdy neviděl, a itemům k opakování. Každý lesson item má navíc podle svého typu (slovíčko, gramatika, poslech, …) určené, kolik z jeho cvičení má být do session zařazeno.

Na základě těchto hodnot tedy algoritmus postupně vybírá vhodné lesson itemy. Pro každý z nich pak přidá příslušný počet cvičení do výsledné množiny

a jakmile je naplněn časový cíl, zastaví se. Takto postupuje nezávisle na sobě pro itemy k opakování a pro nové.

4.4.2.1 Výběr lesson itemů

Nové lesson itemy se v rámci jedné session vybírají vždy pouze z jedné lekce. V případě denního učení je vybrána zrovna aktivní lekce z daného kurzu. Tyto lesson itemy jsou pak ještě seřazeny podle pořadí v rámci lekce, na základě kterého jsou pak postupně procházeny. Není-li pořadí itemu definované, je zařazen až na konec. Řazení je zavedeno proto, že je obvykle vhodné učit se slovíčka postupně od lehčích k těžším, což může administrátor ovlivňovat nastavováním pořadí itemů v lekci.

K opakování jsou vybírány ty lesson itemy, jejichž datum k zopakování již vypršelo. Řazené jsou pak podle tohoto data tak, aby byly přednostně vybírány ty se starším datem.

Není-li dostupný dostatek nových lesson itemů k naplnění jim vyhrazeného časového cíle, je study session doplněna o itemy k opakování. Pokud není při učení konkrétní lekce dostatek itemů, jejichž čas k opakování se již naplnil, je session doplněná o itemy, které se mají opakovat až v budoucnu, ale opět se nejprve vybírají ty s nejdřívějším datem.

4.4.2.2 Výběr cvičení

Způsob výběru jednotlivých cvičení pro daný lesson item může probíhat více způsoby. V nejpřímočařejší variantě je vybrán počet náhodných cvičení v závislosti na typu itemu. Jinak je tomu ale v případě, kdy jsou definovány tzv. *learning steps*.

4.4.2.3 Learning steps

Learning steps umožňují prioritizovat vybraná cvičení tak, aby byla uživateli vybírána v prvních fázích učení daného lesson itemu. Každý learning step v aplikaci obsahuje množinu cvičení a k tomu informaci, do které hodnoty intervalu mezi opakováními příslušného lesson itemu se má daný learning step použít. Pokud je tedy v postupu uživatele v rámci učení lesson itemu interval mezi opakováními nastaven na hodnotu i, budou vybrána cvičení z nejbližšího learning stepu, jehož hodnota `to_interval` je vyšší než i. Pokud takovou podmínku žádný z definovaných learning stepů nesplňuje, a interval itemu je tedy již vyšší, než hodnoty všech learning stepů, jsou cvičení vybírána náhodně, tedy stejně, jako by learning steps nebyly definovány.

4.4.2.4 Řazení cvičení v session

Poté, co je množina cvičení vybrána, proběhne ještě její uspořádání. Aby se vedle sebe nezobrazovala cvičení stejného typu a neshlukovala se cvičení podle

```
if (filter.LessonId != null)
{
  if (await _unitOfWork.LessonRepository.UserHasAccess(filter.LessonId.Value))
  {
    // new
    long totalTimeNew = (long)(_configuration.SessionLengthMs *
↪   _configuration.TimeForNewItems);
    var itemsNew = await
↪   _unitOfWork.LessonItemRepository.GetNewInLessonOrdered(filter.LessonId.Value);
    (IEnumerable<GetExerciseDTO> exercises, long time) res = await
↪   GetExercisesForOrderedItems(itemsNew, totalTimeNew, true);
    exerciseDTOs = res.exercises.ToList();
    exerciseDTOs.ForEach(e =>
    {
      .IsNew = true;
      e.LessonId = filter.LessonId.Value;
      });

    // to review
    long totalTimeReview = _configuration.SessionLengthMs - res.time;
    var itemsToReview = await _unitOfWork.LessonItemRepository
    .GetToReviewInLessonOrdered(filter.LessonId.Value);
    exerciseDTOs.AddRange((await GetExercisesForOrderedItems(itemsToReview,
↪   totalTimeReview, false)).Item1);
  }
  else throw new AccessDeniedException("You have no access to this lesson.");
}
```

Výpis kódu 4.1 Implementace sestavení study session pro konkrétní lekci

lesson itemů, dojde k jejich náhodnému promíchání.

4.4.3 Zpracování výsledků study session

Úspěšnost splnění jednotlivých cvičení vyhodnocuje uživatel sám, resp. frontend. Toto ohodnocení na škále od 0 do 100 uživatel posílá společně s identifikátory cvičení a příslušných lesson itemů. Algoritmus nejprve seskupí výsledky podle lesson itemu a všechna odpovídající ohodnocení zprůměruje.

Následně prochází postupně všechny dotčené lesson itemy a upravuje jejich postup učení na základě algoritmu SM-2. Průměrná ohodnocení jsou přepočítána na škálu 0–5, aby mohla být v algoritmu použita jako kvalita odpovědi. Proběhne výpočet nového E-Faktoru a intervalu a nakonec se ještě zapíše nové datum k zopakování.

Pokud se již uživatel po této session seznámil se všemi itemy v lekci – jsou ve stavu `REVIEW` – je pro něj lekce označena jako naučená (`IsLearned`). Do tohoto stavu ji může uživatel uvést i manuálně přes příslušný požadavek, pokud má pocit, že již látku lekce ovládá a nechce se ji učit od začátku. V takovém případě je postup učení všech jejích lesson itemů uměle přepsán a hodnoty E-

```
1  private void UpdateUserProgress(UserLessonItem progress, double averageRating)
2  {
3    int responseQuality = (int)Math.Round(averageRating / 20.0);
4    if (responseQuality < 3)
5    {
6    progress.Repetitions = 1;
7    progress.Interval = 1;
8    }
9    else
10   {
11     switch (progress.Repetitions)
12     {
13       case 0:
14         progress.Interval = 1; break;
15       case 1:
16         progress.Interval = 6; break;
17       default:
18         progress.Interval = (int)Math.Ceiling(progress.Interval *
   ↪  progress.Easiness); break;
19     }
20     progress.Repetitions++;
21     progress.Easiness = SM2.ComputeEF(progress.Easiness, responseQuality);
22   }
23
24   if (progress.ItemState == LessonItemState.NEW)
25   {
26     progress.ItemState = LessonItemState.REVIEW;
27   }
28
29   progress.DateToReview = DateTime.Today.AddDays(progress.Interval);
30 }
```

Výpis kódu 4.2 Implementace úpravy postupu učení dle algoritmu SM-2

Faktoru, opakování a intervalu jsou nastaveny dle předdefinovaných konstant, které jsem odhadl na základě jednoduché heuristiky.

Na závěr ještě dojde k úpravě učicích statistik uživatele, přičtení bodů a případnému zvýšení jeho úrovně.

4.4.4 Aktivní lekce

Jako aktivní lekce je zpravidla určena první taková lekce na studijní mapě, kterou uživatel ještě nemá naučenou a zároveň je pro něj viditelná. Uživatel se však po studijní mapě může kromě postupu při učení posouvat i manuálně, a tím si přímo zvolit některou lekci za aktivní. Pokud se pak taková lekce stane naučenou či neviditelnou, bude za aktivní lekci vybírána vždy ta první po ní následující, která podmínky splňuje.

4.5 Mapování vstupních a výstupních objektů

V zájmu uživatelsky přívětivé komunikace přes API je nezbytné, aby objekty, které v tělech požadavků do aplikace vstupují nebo z ní naopak vystupují, nebyly předávané přesně v takové podobě, jak jsou reprezentované v databázi. Uživatel například při vytváření objektu nebude chtít zadávat některé nepovinné údaje a při čtení objektů zase nepotřebuje obdržet všechny atributy, které databáze eviduje, zejména pak není žádoucí zahrnovat všechny vnořené objekty – to může vést k zacyklení při serializaci objektu. Proto aplikace pro komunikaci s vnějším světem používá objekty DTO (*Data Transfer Object*), které jsou posílány přes API a v businessové vrstvě jsou mapovány na vlastní doménové objekty aplikace, které definuje vrstva datová.

K mapování objektů jsem se rozhodl využít knihovnu *AutoMapper*.[4] Ta dokáže na základě shodných názvů atributů sama převádět mezi doménovými objekty a DTO a přitom vynechávat prázdné atributy. Z hlediska konfigurace stačí v hlavičce DTO objektu použít anotaci, která mu přiřadí příslušný doménový objekt, a pak už je rovnou možné v businessové logice volat metodu `Map()`, která samotné mapování provede. Stejně tak lze mapovat celé kolekce.

Velice užitečnou schopností knihovny AutoMapper se ukázalo být mapování polymorfních kolekcí – tedy takových kolekcí, které obsahují různé podtypy jednoho společného nadtypu. Příkladem mohou být různé typy cvičení, které jsou v rámci sestavování study session přidávány do jedné kolekce a se všemi se pracuje jako se společným nadtypem `Exercise`. Uživateli je ale třeba předávat cvičení jako instance jejich skutečných typů, resp. příslušných DTO, a tak musí být při mapování zachována polymorfní vlastnost kolekce. To vyžaduje již pár řádků konfigurace a vytvoření tzv. *mapovacího profilu*, samotné mapování kolekce je pak ale provedeno stejně jednoduchým voláním, jako v běžných případech:

```
var exerciseDTOs = _mapper.Map<IEnumerable<Exercise>,
↪  List<GetExerciseDTO>>(exercises);
```

4.6 Zpracování výjimek

V neposlední řadě musí aplikace vhodným způsobem reagovat na chyby za běhu, odchytávat výjimky a informovat o tom uživatele. Po obeznámení se s několika možnými způsoby, jak toto řešit, jsem se rozhodl implementovat *middleware*, který zpracovává výjimky vyhozené aplikací a převádí je na návratové kódy HTTP.

Nejprve bylo potřeba zadefinovat různé typy výjimek, které aplikace využívá, jako podtypy třídy `System.Runtime.Exception`. Těmito typy jsou například `AccessDeniedException` (uživatel nemá oprávnění k dané operaci),

[4] https://automapper.org

`InvalidIDException` (uživatel zadal neexistující identifikátor) nebo `UserNotInCourseException` (uživatel chce provádět operaci v rámci kurzu, který nemá zapsaný). Všem těmto výjimkám je možné v konstruktoru předat chybovou hlášku, která se ve výsledku zobrazí uživateli.

Následně jsem v prezentační vrstvě definoval třídu `ErrorHandlingMiddleware` a zaregistroval ji jako middleware v souboru `Program.cs`, který slouží jako vstupní bod celé aplikace. Registrací se middleware zařadí do *pipeline*, kterou prochází každý zpracovávaný požadavek, a jednotlivé middleware mají možnost do tohoto zpracování v jistou chvíli vstupovat. Samotný `ErrorHandlingMiddleware` pouze předá zpracování požadavku dále, ale zaobalí toto volání do *try–catch* bloku, který zachytává všechny neošetřené výjimky vyvstanuvší během zpracování. Takto odchyceným výjimkám jsou na základě jejich typu přiřazené návratové kódy HTTP – middleware tedy následně vytvoří odpověď na požadavek s tímto kódem a do jejího těla zapíše chybovou hlášku z výjimky.

4.7 Konfigurace

Konfigurace aplikace se v ASP.NET aplikacích standardně udržuje v souboru `appsettings.json`. K tomu navíc používám ještě soubory `appsettings.Development.json` a `appsettings.Production.json`, kde jsou zapsány hodnoty, které jsou odlišné pro různá prostředí. K ukládání tajných konfiguračních údajů jako přístupová hesla nebo připojovací řetězec k databázi využívám .NET nástroj *Secret Manager*, který tyto údaje ukládá do lokálního úložiště. Při spuštění v prostředí vývoje je pak toto úložiště automaticky připojeno ke zdrojům konfigurace [61].

Jednotlivé části konfiguračních souborů rozděluji do sekcí dle jejich využití v programu, rozlišuji tak například konfiguraci učicího algoritmu a posílání e-mailů. K těmto sekcím pak s využitím návrhového vzoru *Options* přistupuji prostřednictvím samostatných tříd, na které jsou namapovány v kontejneru vkládání závislostí. Do tříd, které za běhu programu potřebují k jednotlivých konfiguračním sekcím přístup, jsou pak tyto namapované třídy vkládány jako závislosti a poskytují tak přístup jen k hodnotám, které daná třída potřebuje, nikoliv k celé konfiguraci aplikace. Takové chování je žádoucí v zájmu dodržení principů oddělení zodpovědností (*Separation of Concerns*) a zapouzdření [62].

Kapitola 5
Testování

Naimplementovanou aplikaci bylo nutné vhodným způsobem otestovat. Rozhodl jsem se backend testovat na dvou úrovních, jednak prostřednictvím jednotkových testů, a jednak testů systémových. Jelikož rozhraní backendové části není primárně určeno pro koncového uživatele, není ji samotnou možné testovat uživatelsky. V budoucnu ale bude možné realizovat uživatelské *end-to-end* testování celého systému s připojeným frontendem.

5.1 Jednotkové testy

Jednotkové testy (*unit tests*) se vyznačují tím, že izolovaně testují co možná nejmenší stavební bloky aplikace, tedy ideálně jednotlivé metody. Jejich výhodou je, že tak umožňují velice přesně určit, kde v kódu se případná chyba nachází, a tím zásadně usnadňují údržbu kódu. Zároveň jsou snadno automatizovatelné a jejich běh je poměrně rychlý. Na druhou stranu je jejich vytváření značně pracné, obzvlášť je-li cílem skutečně otestovat každou metodu a pokrýt všechny možné vstupy. Z toho důvodu nebylo v rozsahu této práce možné dosáhnout významného pokrytí jednotkovými testy, bylo však připraveno prostředí pro jejich vytváření, vytvořeny testy pro část aplikace a jejich automatické spouštění bylo zavedeno do procesu průběžné integrace.

5.1.1 Realizace testů

Pro testy jsem ve svém řešení vytvořil nový projekt, ve kterém jsem definoval jednotlivé testovací případy jako metody rozdělené do tříd podle toho, kterou ze tříd aplikace testují. Tento projekt musí záviset na ostatních projektech aplikace, vyjma prezentační vrstvy, která není vhodná pro jednotkové testování. V projektu jsem využil knihovnu *NUnit*, která je ve vývojářské komunitě .NET hojně využívaná. V první fázi každého testu jsem vydefinoval vstupní objekty pro testovanou metodu a zejména pak připravil instance tříd, které

testovaná třída vyžaduje jako své závislosti. V případě businessové vrstvy jsou těmito závislostmi především repozitáře z datové vrstvy, ale také mapovací rozhraní IMapper nebo objekty zapouzdřující konfigurační proměnné. Pro realizaci těchto závislostí jsem použil tzv. testovací dvojníky (*test doubles*), konkrétně dvojníky typu *stub* vytvořené knihovnou *Moq*. Na těchto objektech jsem předdefinoval chování při volání jednotlivých metod a vložil je jako závislosti do konstruktoru testované třídy.

Následně jsem zavolal testovanou metodu a výsledek jsem porovnal s předdefinovým očekávaným výsledkem. K tomu jsem využil univerzální metodu z knihovny NUnit `Assert.That`, která má mnoho využití pro různé typy objektů.

5.1.2 Automatizace a průběžná integrace

Správně napsané jednotkové testy lze spouštět přímo pomocí terminálového nástroje `dotnet test`. Toto spouštění je pak také možné zautomatizovat v rámci procesu průběžné integrace (*continuous integration*). K nastavení tohoto procesu jsem využil službu GitHub Actions, která při každém nahrání změny do vzdáleného repozitáře aplikaci automaticky sestaví a spustí testy. O výsledku těchto testů je pak vývojář informován. Jednotkové testy jsou tak spouštěny po každé provedené změně v kódu, což umožňuje i okamžité regresní testování po přidání nových funkcionalit.

5.2 Systémové testy

Pro otestování backendu jako celku jsem připravil sadu HTTP požadavků určených k otestování funkčnosti kompletního API. Ty je možné ručně či automatizovaně zasílat do aplikace v testovacím prostředí a porovnávat odpovědi s očekávaným chováním. Požadavky simulují zejména běžný průchod aplikací tak, jak je očekáváno od uživatelů, aby byla primárně zajištěna podpora hlavních funkcionalit. Důkladně se také zaměřují na testování komponent zodpovědných za autentizaci a autorizaci uživatelů. Dojde tak k otestování celého backendu se zapojením všech jeho částí, tedy na všech vrstvách architektury.

5.2.1 Příprava prostředí

Při přípravě testovacích požadavků jsem narazil na předpokládaný problém konzistence počátečního stavu databáze. Před každým testováním je žádoucí začínat se stejným stavem databáze, aby bylo možné jednoznačně určit očekávané odpovědi. Zároveň je vhodné, aby databáze nebyla úplně prázdná, ale byla již naplněna základním obsahem, aby každému testování nemuselo předcházet rutinní zaslání množství požadavků v roli administrátora, které by toto plnění jinak zajistilo. Vytvořil jsem proto obraz testovací databáze pro Docker založený na stroji Microsoft SQL Server, ve kterém je po sestavení spuštěn

SQL skript, jenž zajistí vytvoření schématu a naplnění databáze daty. Pro orchestraci testovacího prostředí jsem přidal nový profil nástroji Docker Compose, který zajistí připojení lokálně spouštěné aplikace k testovací databázi, namísto standardně používané databáze pro vývoj. Kontejner testovací databáze záměrně nemá namapovanou paměť na souborový systém hostitelského systému, při novém sestavení kontejneru tak databáze obsahuje stále stejná původní data, a zápisy provedené v předchozích testovacích bězích se nezachovají.

5.2.2 Možnosti automatizace

S konzistencí databáze a podporou jejího automatického spouštění přes Docker Compose se nabízí možnost automatizace i systémových testů. Zatím jsem testovací požadavky spouštěl pouze manuálně přes nástroj *Postman*,[1] v tomtéž nástroji lze ale vytvořit i testovací skripty, které automaticky spustí danou sekvenci požadavků a porovnají odpovědi s očekávanými výsledky. Alternativně je možné využít některou z testovacích knihoven pro .NET (například již zmiňovaný NUnit), a zasílat požadavky v kódu. Takto napsané testy by pak mohly být, podobně jako jednotkové, spouštěny a vyhodnocovány zcela automaticky v procesu průběžné integrace, což by ale z důvodu jejich časové náročnosti nebylo praktické, a tak tuto variantu ani do budoucna neplánuji realizovat.

[1] https://www.postman.com

Kapitola 6
Nasazení

Pro nasazení aplikace do produkčního prostředí bylo potřeba vybrat vhodný hosting. Ten by měl vzhledem k povaze aplikace poskytovat spolehlivou dostupnost a nabízet možnost snadného škálování. Z toho důvodu jsem se soustředil na velké známé poskytovatele cloudových PaaS (*Platform as a Service*), kteří disponují spolehlivou vysoce distribuovanou infrastrukturou a jsou bez problému schopné přidělovat více výpočetních prostředků, je-li to potřeba. Dále v mém rozhodování hrála velkou roli kompatibilita v rámci ekosystému služeb Microsoft, aby bylo nasazení backendu v ASP.NET i databáze Microsoft SQL Server co nejméně komplikované. Z toho důvodu jsem nakonec zvolil řešení Microsoft Azure.

Azure nabízí službu *App Service* pro hostování API a *SQL Database* pro databáze SQL Server. Obě tyto služby jsou zpoplatněny režimem *pay-as-you-go*, tedy průběžnými platbami na základě využitých prostředků. Do jistého – jakkoliv poměrně nízkého – denního, resp. měsíčního limitu prostředků je využívání služeb dokonce zcela bez poplatků.

Díky úzké spolupráci s platformou GitHub nabízí Azure předdefinovou konfiguraci pro průběžné nasazování (*CI/CD*) skrze GitHub Actions. Po připojení repozitáře se zdrojovým kódem aplikace vygeneruje Azure tzv. soubor pracovního postupu (*workflow file*) pro GitHub Actions, který obsahuje instrukce pro sestavení a následné automatické nasazení do cloudu Azure. Poněkud překvapivě se v tomto jednoduchém souboru vyskytují zásadní chyby, které si musí uživatel sám opravit, aby vůbec nasazení fungovalo. Volají se například zastaralé a nekompatibilní verze jednotlivých actions a nejsou ani správně ošetřené řetězce obsahující cestu k souborům, se kterými samo Azure pracuje.

Připojení databáze k aplikaci probíhá poměrně přímočaře poskytnutím správného připojovacího řetězce (*connection string*). V ideálním případě by bylo žádoucí zajistit výhradní přístup k databázi pouze aplikaci a uzavřít ji tak od vnějšího světa. K tomu Azure poskytuje službu *Virtual Network*, která umožňuje propojení jednotlivých služeb virtuální sítí. Tu však není možné připojit k produktům s možností bezplatného užívání, a tak jsem od jejího použití

alespoň prozatím upustil. Z hlediska minimálních oprávnění jsem tedy zvolil druhou nejlepší možnost, a to umožnění přístupu k databázi všem službám v síti Azure. K tomu jsem ještě přidal pravidlo pro firewall, aby bylo možné připojení i z mé domácí sítě za účelem ladění. Samotné vytvoření schématu zajistí aplikace pro připojení k databázi na základě tzv. migrací, které vznikaly průběžně během vývoje. Jde o záznamy změn v datovém modelu aplikace, které vytváří a ukládá do datové vrstvy kódu framework Entity Framework. Tyto záznamy mohou být následně kdykoliv aplikovány ve formě DDL[1] operací do databáze.

6.1 Citlivé konfigurační údaje

Na produkční prostředí bylo třeba správně přenést konfigurační proměnné. Běžné údaje specifické pro produkční prostředí jsou zapsány v souboru `appsettings.Production.json`, citlivé a tajné údaje byly během vývoje udržovány v lokálním úložiště v podobě *user secrets*. Přímo v Azure je možné definovat proměnné prostředí a přiřadit jim hodnoty, které jsou ukládány v šifrované podobě a za běhu aplikace jsou přidány ke zdrojům konfigurace. Tímto způsobem tedy na produkci bezpečně ukládám veškerá hesla, klíče a připojovací řetězec do databáze.

6.2 Monitorování

K monitorování aplikace za běhu a k logování chyb jsem zvolil řešení Azure Application Insights. Oproti jiným konkurenčním řešením (např. Sentry) má tu nespornou výhodu, že je součástí ekosystému služeb Microsoft a je dostupné přímo v portálu Azure. Nabízí rozhraní jak pro procházení aplikačních logů, tak i živé sledování příchozích požadavků a výkonu.

Aplikaci jsem k Insights připojil pohodlně přímo ve vývojovém prostředí Visual Studio, které se postaralo o doplnění potřebných balíčků a správnou konfiguraci. Díky připojenému účtu Microsoft pak i samo nalezlo mé Azure předplatné a zapsalo do produkčního prostředí aplikace vygenerovaný připojovací řetězec k vytvořené instanci Insights.

6.3 Dokumentace

K aplikaci je dostupná automaticky generovaná dokumentace koncových bodů API v nástroji Swagger.[2] Ten na základě implementace koncových bodů, komentářů a anotací vytváří specifikaci dle standardu *OpenAPI* a na základě ní pak i generuje grafické webové rozhraní, které jednotlivé koncové body přehledně dokumentuje a umožňuje na ně i přímo interaktivně posílat požadavky.

[1] Data Definition Language
[2] https://swagger.io

Tuto formu dokumentace jsem využíval již od počátku vývoje, a to zejména pro snadné manuální testování průběžně naimplementovaných funkcionalit. Po nasazení aplikace na Azure jsem dokumentaci zpřístupnil také tam, takže je rovněž veřejně dostupná. Velkou výhodou tohoto přístupu je, že publikovaná verze dokumentace vždy odpovídá aktuálně nasazené verzi API, neboť je znovu generovaná při každém novém nasazení. To bude jistě velice přínosné pro kolegu Jeníčka při integraci jeho frontendové části.

Závěr

Cílem práce bylo navrhnout, implementovat a nasadit backend webové aplikace na učení se angličtiny v podobě REST API a databáze. Zásadní při tom byla důkladná analýza a následně implementace efektivního algoritmu řídícího postup učení uživatele. Frontendové části aplikace se ve své práci věnoval kolega Jan Jeníček.

Výsledkem je serverová část aplikace realizovaná v technologii .NET, konkrétně v jejím frameworku ASP.NET Core, s rozhraním v podobě REST API. Obsahuje všechny funkcionality, které byly v rámci analýzy požadavků ohodnoceny nejvyššími prioritami *must have* a *should have* a některé z kategorie *could have*. Aplikace tedy umožňuje registraci a přihlašování uživatelů a nabízí jim výběr kurzů, tematických okruhů i konkrétních lekcí tak, aby mohli sami co nejvíce ovlivnit, na co se při svém učení zaměří. Předně pak nabízí uživatelsky přívětivý a zároveň konfigurovatelný přístup k učení samotných slovíček a gramatických jevů. Aplikace implementuje vlastní algoritmus, který vybírá, která slovíčka a gramatiku by se měl uživatel v závislosti na jeho dosavadních studijních výsledcích učit, tak, aby si jazyk osvojil v jeho plné šíři za co možná nejkratší dobu. K tomu využívá prokazatelně efektivní algoritmus učení SM-2. Některé rozšiřující funkcionality jsou pak dostupné uživatelům v rámci předplatného, které je spravováno ve spolupráci se službou třetí strany.

V době dopsání práce je backend s databází nasazen na produkčním prostředí cloudového poskytovatele, a REST API je tak veřejně přístupné na adrese `linguino.azurewebsites.net`. Je k němu připojena frontendová část a společně tak tvoří funkční webovou aplikaci, která je volně dostupná všem zájemcům o učení se angličtiny pod názvem Linguino.

Aby byla aplikace na trhu skutečně konkurenceschopná, bude ji v budoucnu potřeba ještě nad rámec základních funkcionalit rozšířit. Navržená rozšíření zajistí zejména zatraktivnění celého procesu učení pro běžného uživatele, například formou získávání ocenění, plnění výzev či obchodu s virtuální měnou. To vše si budou moct uživatelé navzájem sdílet, což ještě více přispěje k jejich motivaci aplikaci pravidelně používat. Navíc by v pozdějších verzích aplikace bylo vhodné zvážit vylepšení učicího algoritmu nahrazením základového al-

goritmu SM-2 ještě efektivnější variantou. V neposlední řadě by bylo velice užitečné zpřehlednit a rozšířit administrátorské rozhraní, neboť za současného stavu je k používání administrátorských koncových bodů nutná jistá znalost implementačních specifik aplikace.

Bibliografie

1. PROFINIT EU, s. r. o. Softwarový proces. In: *Softwarové inženýrství 2* [online]. České vysoké učení technické v Praze, 2023 [cit. 2024-01-19]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/641049/course/section/101354/2023_2024/01_SoftwareProcess.pdf.
2. CLARK, Hannah. *The Software Development Life Cycle (SDLC): 7 Phases and 5 Models* [online]. c2024 [cit. 2023-11-28]. Dostupné z: https://theproductmanager.com/topics/software-development-life-cycle.
3. VELIMIROVIC, Andreja. *What is SDLC? Software Development Life Cycle Defined* [online]. 2022-11-17 [cit. 2024-04-17]. Dostupné z: https://phoenixnap.com/blog/software-development-life-cycle.
4. ALEXANDRA. *What Is SDLC? Understand the Software Development Life Cycle* [online]. 2024-02-28 [cit. 2024-04-17]. Dostupné z: https://stackify.com/what-is-sdlc.
5. RAJKUMAR. *What Is Software Testing | Everything You Should Know* [online]. 2024-03-22 [cit. 2024-05-10]. Dostupné z: https://www.softwaretestingmaterial.com/software-testing.
6. IBM. *What is software testing?* [Online] [cit. 2024-05-10]. Dostupné z: https://www.ibm.com/topics/software-testing.
7. DOSHI, Kalpesh. *Different Types of Testing in Software | BrowserStack* [online]. 2023-03-22 [cit. 2024-05-10]. Dostupné z: https://www.browserstack.com/guide/types-of-testing.
8. CONFIANZ. *SDLC Deployment Phase - A Step-By-Step Guide* [online]. 2023-06-21 [cit. 2024-05-10]. Dostupné z: https://www.confianzit.com/cit-blog/sdlc-deployment-phase-a-step-by-step-guide.
9. GORBACHENKO, Pavel. *Functional vs Non-Functional Requirements* [online]. c2024 [cit. 2023-11-28]. Dostupné z: https://enkonix.com/blog/functional-requirements-vs-non-functional.

10. GRADY, Robert; CASWELL, Deborah. *Software Metrics: Establishing a Company-wide Program*. Prentice Hall, 1987. ISBN 0-13-821844-7.
11. GRADY, Robert. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992. ISBN 978-0137203840.
12. EELES, Peter. *Capturing Architectural Requirements* [online]. 2005-11-15 [cit. 2023-11-28]. Dostupné z: `https://web.archive.org/web/20201112020231/http://www.ibm.com/developerworks/rational/library/4706.html`.
13. CRADDOCK, Andrew. *The DSDM Agile Project Framework*. DSDM Consortium, 2014. ISBN 978-0954483296. Dostupné také z: `https://www.agilebusiness.org/dsdm-project-framework.html`.
14. PYLE, W. H. Economical learning. *Journal of Educational Psychology*. 1913, roč. 4, č. 3, s. 148–158. Dostupné z DOI: `10.1037/h0071752`.
15. WOŹNIAK, Piotr. *Active recall* [online]. 2018-07-04 [cit. 2024-01-25]. Dostupné z: `https://supermemo.guru/wiki/Active_recall`.
16. LOCKE, John. *An Essay Concerning Humane Understanding*. Project Gutenberg, 1690. Dostupné také z: `https://www.gutenberg.org/files/10615/10615-h/10615-h.htm`.
17. EBBINGHAUS, Hermann. *Über das Gedächtnis*. Duncker & Humblot, 1885. Dostupné také z: `https://home.uni-leipzig.de/wundtbriefe/wwcd/opera/ebbing/memory/GdaechtI.htm`.
18. WOŹNIAK, Piotr. *Error of Ebbinghaus forgetting curve* [online]. 2017 [cit. 2024-01-24]. Dostupné z: `https://supermemo.guru/wiki/Error_of_Ebbinghaus_forgetting_curve`.
19. BRANWEN, Gwern. *Spaced Repetition for Efficient Learning* [online]. 2019 [cit. 2024-01-21]. Dostupné z: `https://gwern.net/spaced-repetition`.
20. PETERSON, Lloyd R.; WAMPLER, Richard; KIRKPATRICK, Meredith; SALTZMAN, Dorothy. Effect of spacing presentations on retention of a paired associate over short intervals. *Journal of Experimental Psychology*. 1963, roč. 66, č. 2, s. 206–209. Dostupné z DOI: `10.1037/h0046694`.
21. LEITNER, Sebastian. *So lernt man lernen*. Herder, 1972. ISBN 9783451162657.
22. WOŹNIAK, Piotr. *The true history of spaced repetition* [online]. 2018 [cit. 2024-01-21]. Dostupné z: `https://www.supermemo.com/en/blog/the-true-history-of-spaced-repetition`.
23. WOŹNIAK, Piotr. *Optimization of learning*. Poznan, 1990. Dostupné také z: `https://super-memory.com/english/ol.htm`. Dipl. pr. University of Technology in Poznan, Computer Science Center. Vedoucí práce Prof. Zbigniew KIERZKOWSKI.

24. ANKI. *What spaced repetition algorithm does Anki use?* [Online]. 2023-11-16 [cit. 2024-01-24]. Dostupné z: `https://faqs.ankiweb.net/what-spaced-repetition-algorithm.html`.
25. WOŹNIAK, Piotr. *Three component model of memory* [online]. 2018-08-06 [cit. 2024-01-24]. Dostupné z: `https://supermemo.guru/wiki/Three_component_model_of_memory`.
26. YE, Junyao; SU, Jingyong; CAO, Yilong. A Stochastic Shortest Path Algorithm for Optimizing Spaced Repetition Scheduling. In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, s. 4381–4390. ISBN 9781450393850. Dostupné z DOI: `10.1145/3534678.3539081`.
27. CLARITYINMADNESS. FSRS is now the most accurate spaced repetition algorithm in the world*. In: *Reddit* [online]. 2023 [cit. 2024-01-24]. Dostupné z: `https://www.reddit.com/r/Anki/comments/18csuer/fsrs_is_now_the_most_accurate_spaced_repetition`.
28. ANKI. *Changes in 23.10 - Changes* [online]. 2023-11-01 [cit. 2024-01-24]. Dostupné z: `https://changes.ankiweb.net/changes/23.10.html`.
29. WRIKE, Inc. *What is Kanban Methodology? The Ultimate Guide | Wrike* [online]. c2006–2024 [cit. 2024-04-05]. Dostupné z: `https://www.wrike.com/kanban-guide/what-is-kanban`.
30. ANTALFFY, Paula. *6 Most Popular Back-End Programming Languages* [online]. c2023 [cit. 2024-01-15]. Dostupné z: `https://www.deazy.com/knowledge-hub/top-6-back-end-languages`.
31. STACK EXCHANGE, Inc. *Stack Overflow Developer Survey 2023* [online]. 2023 [cit. 2024-01-15]. Dostupné z: `https://survey.stackoverflow.co/2023`.
32. QSERVICES. *ASP.NET vs PHP: Choosing the Ideal Platform for Website and App Development* [online]. 2023-06-07 [cit. 2024-01-15]. Dostupné z: `https://medium.com/@seoqservicesit/asp-net-vs-php-choosing-the-ideal-platform-for-website-and-app-development-a45f7acc823c`.
33. PASICHNYK, Oleh. *Essential Insights: Asp Net Vs Python For Coders* [online]. 2023-11-15 [cit. 2024-01-15]. Dostupné z: `https://marketsplash.com/tutorials/asp-net/asp-net-vs-python`.
34. RANA, Nishant. *General Workflow of a Java Program* [online]. 2017-09-04 [cit. 2024-01-16]. Dostupné z: `https://www.codementor.io/@nishantrana/general-workflow-of-a-java-program-bme3lpkhg`.
35. NIXSOLUTIONS. *The Main Factors Determining a Speed of the Web App* [online]. 2018-07-24 [cit. 2024-01-16]. Dostupné z: `https://www.nixsolutions.com/blog/main-factors-determining-speed-web-app`.

36. MONGODB, Inc. *Types of Databases* [online]. c2024 [cit. 2024-01-19]. Dostupné z: https://www.mongodb.com/databases/types.

37. MICROSOFT. *Work with data in ASP.NET Core Apps* [online]. 2023-04-25 [cit. 2024-01-19]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/work-with-data-in-asp-net-core-apps#sql-or-nosql.

38. DATASCIENTEST. *SQL vs NoSQL: differences, uses, advantages and disadvantages* [online]. 2023-02-10 [cit. 2024-01-19]. Dostupné z: https://datascientest.com/en/sql-vs-nosql.

39. BULLARD, Brett. What would be the best database other than SQL server to work with ASP.Net MVC? In: *Quora* [online]. 2018 [cit. 2024-01-19]. Dostupné z: https://www.quora.com/What-are-the-pros-and-cons-of-using-MySQL-instead-of-SQL-Server-with-ASP-NET/answer/Brett-Bullard-1. Odpověď na otázku z webového fóra.

40. STRIPE, Inc. *Stripe API Reference* [online]. 2023 [cit. 2024-03-27]. Dostupné z: https://docs.stripe.com/api.

41. NYSTROM, Robert. *Game Programming Patterns*. 2014. Dostupné také z: https://gameprogrammingpatterns.com.

42. MONTIEL, Ivan. *Low Coupling, High Cohesion* [online]. 2018-09-17 [cit. 2024-01-26]. Dostupné z: https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6.

43. BAELDUNG. *Layered Architecture* [online]. 2021-11-11 [cit. 2024-01-26]. Dostupné z: https://www.baeldung.com/cs/layered-architecture.

44. RUBIN, Dean. *The Three Layered Architecture* [online]. 2021-10-19 [cit. 2024-01-27]. Dostupné z: https://medium.com/@deanrubin/the-three-layered-architecture-fe30cb0e4a6.

45. SCALED AGILE, Inc. *Advanced Topic - Domain Modeling - Scaled Agile Framework* [online]. 2023-03-02 [cit. 2024-03-29]. Dostupné z: https://scaledagileframework.com/domain-modeling.

46. MICROSOFT. *Inheritance - EF Core | Microsoft Learn* [online]. 2023-01-12 [cit. 2024-03-30]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/modeling/inheritance.

47. IBM. *What is a REST API? | IBM* [online] [cit. 2024-03-30]. Dostupné z: https://www.ibm.com/topics/rest-apis.

48. GITHUB, Inc. *REST API endpoints for followers - GitHub Docs* [online]. 2022-11-28 [cit. 2024-04-01]. Dostupné z: https://docs.github.com/en/rest/users/followers?apiVersion=2022-11-28.

49. VERMA, Pragati. *Unlocking the Power of API Pagination: Best Practices and Strategies - DEV Community* [online]. 2023-06-06 [cit. 2024-04-01]. Dostupné z: https://dev.to/pragativerma18/unlocking-the-power-of-api-pagination-best-practices-and-strategies-4b49.

50. ANDERSON, Rick. *Introduction to Identity on ASP.NET Core | Microsoft Learn* [online]. 2023-08-25 [cit. 2024-03-27]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity.

51. LOCK, Andrew. *Should you use the .NET 8 Identity API endpoints?* [Online]. 2023-09-19 [cit. 2024-04-01]. Dostupné z: https://andrewlock.net/should-you-use-the-dotnet-8-identity-api-endpoints/#what-are-the-new-identity-api-endpoints.

52. ARIAS, Dan; BELLEN, Sam. *What Are Refresh Tokens and How to Use Them Securely* [online]. 2021-10-07 [cit. 2024-03-27]. Dostupné z: https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them.

53. JONES, Michael B.; BRADLEY, John; SAKIMURA, Nat. *JSON Web Token (JWT)* [RFC 7519]. RFC Editor, 2015. Request for Comments, č. 7519. Dostupné z DOI: 10.17487/RFC7519.

54. STRIPE, Inc. *Stripe | Financial Infrastructure for the Internet* [online]. c2024 [cit. 2024-03-27]. Dostupné z: https://stripe.com/en-cz.

55. STRIPE, Inc. *How subscriptions work | Stripe Documentation* [online] [cit. 2024-03-27]. Dostupné z: https://docs.stripe.com/billing/subscriptions/overview.

56. STRIPE, Inc. *Using webhooks with subscriptions | Stripe Documentation* [online] [cit. 2024-03-27]. Dostupné z: https://docs.stripe.com/billing/subscriptions/webhooks.

57. MICROSOFT. *Troubleshooting operating system disk sector size greater than 4 KB - SQL Server | Microsoft Learn* [online]. 2023-11-24 [cit. 2024-04-05]. Dostupné z: https://learn.microsoft.com/en-us/troubleshoot/sql/database-engine/database-file-operations/troubleshoot-os-4kb-disk-sector-size.

58. LARKIN, Kirk; SMITH, Steve; DAHLER, Brandon. *Dependency injection in ASP.NET Core | Microsoft Learn* [online]. 2023-11-07 [cit. 2024-04-10]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

59. MAKLIN, Cory. *Repository Design Pattern* [online]. 2023-05-31 [cit. 2024-04-11]. Dostupné z: https://www.linkedin.com/pulse/repository-design-pattern-cory-maklin.

60. POPRÓČI, Viliam. *Study materials for the web development in .NET Core*. Brno, 2019. Dostupné také z: `https://is.muni.cz/th/a87c2/bp_full.pdf`. Bak. pr. Masarykova Univerzita, Fakulta informatiky. Vedoucí práce Mgr. Martin MACÁK.

61. ANDERSON, Rick; LARKIN, Kirk. *Safe storage of app secrets in development in ASP.NET Core | Microsoft Learn* [online]. 2024-02-23 [cit. 2024-04-19]. Dostupné z: `https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets`.

62. ANDERSON, Rick. *Options pattern in ASP.NET Core | Microsoft Learn* [online]. 2023-10-26 [cit. 2024-04-19]. Dostupné z: `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options`.

Obsah příloh

- `prilohy.zip` ..... soubory s přílohami
  - `analyza_konkurence.pdf` ..... analýza existujících řešení
  - `open_api_docs.json` ..... specifikace API dle standardu OpenAPI ve formátu JSON
  - `linguino_postman_collection.json` kolekce požadavků pro testování API nástrojem Postman
- `https://github.com/riskin88/LinguinoAPI` ..... repozitář se zdrojovým kódem aplikace
- `https://linguino.azurewebsites.net` ..... nasazená verze aplikace