Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Aplikace na učení gramatik

Gramma

2023

Bc. Vladimír Opluštil

Vedoucí práce:
doc. RNDr. Tomáš Masopust, Ph.D., DSc.

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Vladimír Opluštil
Název práce:	Aplikace na učení gramatik (Gramma)
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	doc. RNDr. Tomáš Masopust, Ph.D., DSc.
Počet stran:	59
Přílohy:	elektronická data v systému katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Vladimír Opluštil
Title:	Learning grammars app
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	doc. RNDr. Tomáš Masopust, Ph.D., DSc.
Page count:	59
Supplements:	electronic data in system of department of computer science
Thesis language:	Czech

Anotace

Obsahem textu je popis webové aplikace Gramma, sloužící k učení gramatik přirozených jazyků. Aplikace byla navržena tak, aby byla použitelná pro libovolné jazyky. Popsány jsou implementační detaily a použité technologie všech částí aplikace. Zvláštní pozornost je pak věnována funkčnosti nabízené administrátorskou a uživatelskou částí webové aplikace a mobilní aplikací. Na vypracování byl použit jazyk C# a technologie .NET, ASP.NET Core a Xamarin.

Synopsis

This text contains a description of Gramma, a web application for learning grammars of natural languages. The app was designed in such a way to be usable with any chosen language. Described are the implementation details and used technologies of all its parts. Special focus is given to the functionality offered by the admin and user sections of the web app, as well as by the mobile app. The application was created using the C# language and the technologies .NET, ASP.NET Core and Xamarin.

Klíčová slova: gramatika; přirozené jazyky; webová aplikace; mobilní aplikace, C#

Keywords: grammar; natural languages; web application; mobile application; C#

Děkuji doc. RNDr. Tomáši Masopustovi, Ph.D., DSc. za vedení práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

1 Úvod 8
1.1 Srovnání s podobnými aplikacemi . . . 8
1.2 Teoretický úvod . . . 8

2 Implementace 13
2.1 Databáze . . . 13
2.2 Projektový diagram . . . 14
2.3 ClassLibrary . . . 14
2.3.1 Category a TableTree . . . 15
2.4 CoreCommon . . . 22
2.5 Compiler . . . 22
2.5.1 Fungování Compileru . . . 22
2.6 API . . . 23
2.7 WebApp . . . 27
2.7.1 Razor . . . 27
2.7.2 Javascriptové knihovny . . . 29
2.7.3 Bootstrap . . . 30
2.7.4 Microsoft Identity . . . 30
2.8 MobileApp . . . 30
2.8.1 MobileApp.Droid a MobileApp.iOS . . . 31

3 Aplikace 32
3.1 Administrátorská část . . . 32
3.1.1 Přihlášení . . . 32
3.1.2 Index . . . 32
3.1.3 Language . . . 32
3.1.4 Category . . . 33
3.1.5 Words . . . 34
3.1.6 Word . . . 34
3.1.7 Exercises . . . 34
3.1.8 Exercise . . . 35
3.2 Uživatelská část . . . 35
3.2.1 Index . . . 35
3.2.2 Languages . . . 35
3.2.3 Exercises . . . 36
3.2.4 Exercise - typ Fill . . . 36
3.2.5 Exercise - typ Find . . . 36
3.2.6 Exercise - typ Select . . . 36
3.2.7 Dictionary . . . 36
3.3 Mobilní aplikace . . . 37
3.3.1 Připojení . . . 37
3.3.2 MainPage . . . 37
3.3.3 ExerciseSelectionPage . . . 37

3.3.4 ExercisePage . . . 38
3.3.5 DictionaryPage . . . 38

Závěr **39**

Conclusions **40**

A Zprovoznění aplikace **41**
A.1 Jádro aplikace v Dockeru (fyzický stroj Windows 10) . . . 41
A.2 Jádro aplikace v Dockeru (virtuální stroj Ubuntu Server 22.04 LTS) 42
A.3 Přístup mobilní aplikací zvenčí pomocí ngrok . . . 44

B Vytvoření latinského cvičení v aplikaci, krok za krokem **45**

C Endpointy Compileru a API **50**
C.1 Compiler . . . 50
C.1.1 CodeController . . . 50
C.1.2 FlexController . . . 50
C.1.3 ArchiveController . . . 50
C.2 API . . . 50
C.2.1 LanguagesController . . . 50
C.2.2 RewriterRulesController . . . 51
C.2.3 CategoriesController . . . 51
C.2.4 ExercisesController . . . 51
C.2.5 ExerciseVocabController . . . 52
C.2.6 TablesPrincipalController . . . 52
C.2.7 TablesCompleteController . . . 52
C.2.8 WordsPrincipalController . . . 52
C.2.9 WordsCompleteController . . . 53
C.2.10 ArchiveController . . . 53
C.2.11 CodeController . . . 54

D VocabTransformer **55**
D.1 Popis . . . 55
D.2 Shrnutí výsledků . . . 56

E Obsah adresářové struktury **58**

Bibliografie **59**

Seznam obrázků

1 Slovníkové a kompletní slovo . . . 9
2 Architektura systému . . . 13
3 Vytvořené projekty . . . 14
4 Tabulka při dvou různých šířkách . . . 15
5 Strom před a po odfiltrování červeně označených listů. . . . 18
6 Příklad cvičení typu Select ve webové aplikaci . . . 37
7 Stránka slovníku a cvičení v mobilní aplikaci . . . 38
8 Přidání nového jazyka. . . . 45
9 Stránka Language po přidání nové kategorie. . . . 46
10 Nastavení struktury kategorie. . . . 47
11 Nastavení slovníkových částí slov kategorie. . . . 48
12 Přidání slova ručně. . . . 48
13 Úprava slova ručně. . . . 49
14 Konfigurace cvičení. . . . 49

1 Úvod

Gramma, z řeckého γράμμα - písmeno, je webová a mobilní aplikace určená na učení gramatik různých přirozených jazyků.

Gramatikou zde rozumím tento pojem v nejužším smyslu, tedy určování různých tvarů slova, jako je skloňování jmen nebo časování sloves.

Uživateli nabízí tři typy cvičení určené k naučení gramatických pravidel - psaní tvarů do tabulky, přiřazování tvarů do tabulky a určení tvaru slova. Dále má uživatel možnost vyhledat libovolné slovo a prohlížet si jeho tvary.

Administrátor má k dispozici rozhraní pro pohodlné přidávání nových jazyků, slovních druhů, slovní zásoby a cvičení.

1.1 Srovnání s podobnými aplikacemi

Při učení jazyků je v začátcích důležité získat přehled ve dvou oblastech - slovní zásobě a gramatice. Moderní jazyky, zejména angličtina, mají gramatiku velice jednoduchou, a proto jsou aplikace, které se zabývají učením těchto jazyků, zaměřené především na osvojení slovní zásoby, zejména opakovaným překladem slov a jednoduchých vět.

To je pro své účely dobré, ale při rozšíření na jiné jazyky už tato metoda není tak vhodná. Míním zde především starověké jazyky - latinu a řečtinu, s poměrně složitými gramatikami, ale také češtinu a jiné slovanské jazyky, které ač moderní, si uchovaly složité systémy gramatiky.

U těchto jazyků často stačí změnit koncovku jediného slova a zároveň se změní i smysl celé věty. Mluvčí tedy musí mít dobrý přehled o gramatice jazyka, aby mohl vybírat správné tvary pro přesné zachycení své myšlenky.

Duolingo je nejrozšířenější aplikace na učení jazyků. Sám jsem ji používal na učení italštiny. Na naučení slovní zásoby byla velmi dobrá, ale způsob fungování sloves (kolem padesáti tvarů u slovesa) předat nezvládla. Když v roce 2019 přibyla na Duolingu latina, zkoušel jsem jejich kurz, ale zdál se pro tento jazyk zcela nevhodným.

Ačkoliv nemám dlouhodobější zkušenosti s jinými aplikacemi, vyzkoušel jsem pro potřeby této práce i další populární - Babbel, Mondly a Memrise. Jejich metoda je veskrz podobná té, kterou používá Duolingo.

Aplikace Gramma se snaží být jiným druhem aplikace na učení jazyků. Je to aplikace zaměřená jen na procvičení gramatiky a dobré pochopení gramatických pravidel. Pro učení jazyků nestačí samotná, a proto by se ve skutečném případě používala nejlépe k doplnění klasické učebnice nebo některé ze zmíněných existujících aplikací.

1.2 Teoretický úvod

Pro lepší pochopení následujících částí zde formálněji vysvětlím tři důležité pojmy - slovníkové slovo, kompletní slovo a gramatické pravidlo. Slovníkové a

Quercus, ûs, *f.* an oak

Case	Singular	Plural
Nominative	quercus	quercūs
Genitive	quercūs	quercuum
Dative	quercuī	quercubus
Accusative	quercum	quercūs
Ablative	quercū	quercubus
Vocative	quercus	quercūs

Obrázek 1: Slovníkové a kompletní slovo

kompletní slovo jsou dva tvary, ve kterých si aplikace Gramma ukládá každé slovo. K převodu z kratšího slovníkového slova na delší kompletní slovo slouží gramatická pravidla. Účelem tohoto rozdělení je mít možnost přidávat do aplikace jen jednodušší slovníková slova, která se podle definovaných pravidel sama převedou na odpovídající kompletní slova. Na Obrázku 1 je příklad slovníkového slova (z Latin-English Dictionary [10]) a odpovídajícího kompletního slova (z `wiktionary.org`).

Definice 1 (Slovníkové slovo)
Typ slovníkového slova, označovaný T_p, je kartézský součin množin hodnot.
Slovníkové slovo, nebo word principal (podle latinského pars principalis), označované w_p, je uspořádaná n-tice hodnot.
Pokud $w_p \in T_p$, pak můžeme mluvit o tom, že w_p je typu T_p.

Příklad 2
Nejdříve zavedeme typ T_p, který bude v tomto případě odpovídat italským přídavným jménům, která se svou složitostí zdála vhodná pro tuto ukázku.
Nechť Σ je množina znaků využívaných v italských slovech a Σ^* množina možných řetězců nad touto abecedou.
$T_p = \Sigma^* \times \{invariable, one, two\} \times \{hard, soft, mixed\}$
T_p se tedy skládá ze tří částí - řetězce nad Σ, který představuje kmen slova, položky invariable, one nebo two, která představuje, jak se rozlišují koncovky u dvou rodů, a položky hard, soft nebo mixed, která určuje konkrétní podobu koncovek podle tvrdosti.
Nyní následuje několik příkladů slovníkových slov, a zda odpovídají typu T_p.
$w_{p1} = \langle bianc, two, hard\rangle \in T_p$
$w_{p2} = \langle verd, one, hard\rangle \in T_p$
$w_{p3} = \langle blu, invariable, hard\rangle \in T_p$
$w_{p4} = \langle grig, three, soft\rangle \notin T_p$ Druhá položka není invariable, one nebo two.

$w_{p5} = \langle servus, serv, 2, M, 0\rangle \notin T_p$ Zde neodpovídá ani velikost n-tice.

Poznámka 3

Slovníkové slovo odpovídá položce ve slovníku, která o slově uvádí jen některé základní informace (kořen, rod, vzor skloňování a podobně), ze kterých může čtenář znalý gramatiky odvodit všechny tvary slova.

Typ slovníkového slova zpravidla odpovídá slovnímu druhu, jelikož je ale v některých případech možné odvozovat více slovních druhů od jednoho typu (například přídavná jména a odpovídající příslovce, jako rychlý a rychle a podobně), nepoužívám v textu označení slovní druh, ale kategorie.

Množiny hodnot tvořící typ by mohly teoreticky být jakékoliv, v programu jsou však omezené jen na řetězce, celá čísla, pravdivostní hodnoty a hodnoty z libovolně definovaného výčtu.

Definice 4 (Kompletní slovo)

Kompletní slovo, word complete, označované jako w_c je uspořádaná n-tice řetězců.

Příklad 5

Kompletní slovo může vypadat například takto:

$w_{c1} = \langle bianco, bianca, bianchi, bianche\rangle$

Jeho velikost:

$|w_{c1}| = 4$

Řetězce se mohou i opakovat:

$w_{c2} = \langle verde, verde, verdi, verdi\rangle$

Definice 6 (Gramatické pravidlo)

Gramatické pravidlo g je funkce typu $g\colon T_p \to \Sigma^*$.

Příklad 7

Příklad gramatických pravidel pro T_p:

$w_p = \langle stem, termination, hardness\rangle, w_p \in T_p$

$$g_1(w_p) = \begin{cases} stem\text{, když platí } termination = invariable \\ \text{jinak } stem + e\text{, když platí } termination = one \\ \text{jinak } stem + io\text{, když platí } termination = two \text{ a } hardness = soft \\ \text{jinak } stem + o\text{, když platí } termination = two \text{ a } hardness \neq soft \end{cases}$$

$$g_2(w_p) = \begin{cases} stem\text{, když platí } termination = invariable \\ \text{jinak } stem + e\text{, když platí } termination = one \\ \text{jinak } stem + ia\text{, když platí } termination = two \text{ a } hardness = soft \\ \text{jinak } stem + a\text{, když platí } termination = two \text{ a } hardness \neq soft \end{cases}$$

Pro $w_{p1,2,3}$ z předchozího příkladu platí:

$g_1(w_{p1}) = bianco$
$g_1(w_{p2}) = verde$
$g_1(w_{p3}) = blu$
$g_2(w_{p1}) = bianca$
$g_2(w_{p2}) = verde$
$g_2(w_{p3}) = blu$

Definice 8 (Gramatika)

Pokud máme gramatická pravidla $g_1, g_2, ..., g_n$ pro stejný typ T_p, pak $\Gamma = \langle g_1, g_2, ..., g_n \rangle$ je gramatika pro typ T_p.

Dále, pro $w_p \in T_p$ a $w_c = \langle g_1(w_p), g_2(w_p), ..., g_n(w_p) \rangle$, pak můžeme říci, že w_c je odvozené z w_p pomocí gramatiky Γ, což by se dalo napsat jako $w_p \Rightarrow_\Gamma w_c$.

Příklad 9

$\Gamma = \langle g_1, g_2, g_3, g_4 \rangle$, kde g_1, g_2 jsou z předchozího příkladu a g_3, g_4 jsou další gramatická pravidla.
$w_{p1} \Rightarrow_\Gamma \langle bianco, bianca, bianchi, bianche \rangle$
$w_{p2} \Rightarrow_\Gamma \langle verde, verde, verdi, verdi \rangle$
$w_{p3} \Rightarrow_\Gamma \langle blu, blu, blu, blu \rangle$

Definice 10 (Pravidelná a nepravidelná slova)

Máme-li gramatiku Γ pro typ T_p, pak kompletní slovo w_c, pro které platí $|w_c| = |\Gamma|$, nazveme pravidelné, pokud existuje $w_p \in T_p$ takové, že $w_p \Rightarrow_\Gamma w_c$. Neexistuje-li takové w_p, nazveme w_c nepravidelné.

Příklad 11

Máme gramatiku Γ z předchozího příkladu.
$w_{c5} = \langle nero, nera, neri, nere \rangle$
$w_{c6} = \langle rosa, rosa, rosa, rosa \rangle$
$w_{c7} = \langle gran, grande, grandi, grandi \rangle$
$w_{c8} = \langle san, santa, santi, sante \rangle$

w_{c5} a w_{c6} jsou vůči gramatice Γ pravidelná, příslušná slovníková slova by byla:
$w_{p5} = \langle ner, two, hard \rangle$
$w_{p6} = \langle rosa, invariable, hard \rangle$
Slova w_{c7} a w_{c8} jsou nepravidelná. Pokud bychom chtěli, aby první tvar mohl končit na n, muselo by odpovídající w_p mít $termination = invariable$, pak by ale musely vypadat stejně i všechny ostatní tvary.

Poznámka 12

Jelikož slov je v každém jazyce jen konečný počet, je vždy možné navrhnout typ slovníkového slova a gramatiku tak, aby žádná skutečná slova daného jazyka nebyla vůči gramatice nepravidelná. Docílilo by se toho pomocí přidání položek pro různé výjimky do typu slovníkového slova a upravení gramatiky. V krajním

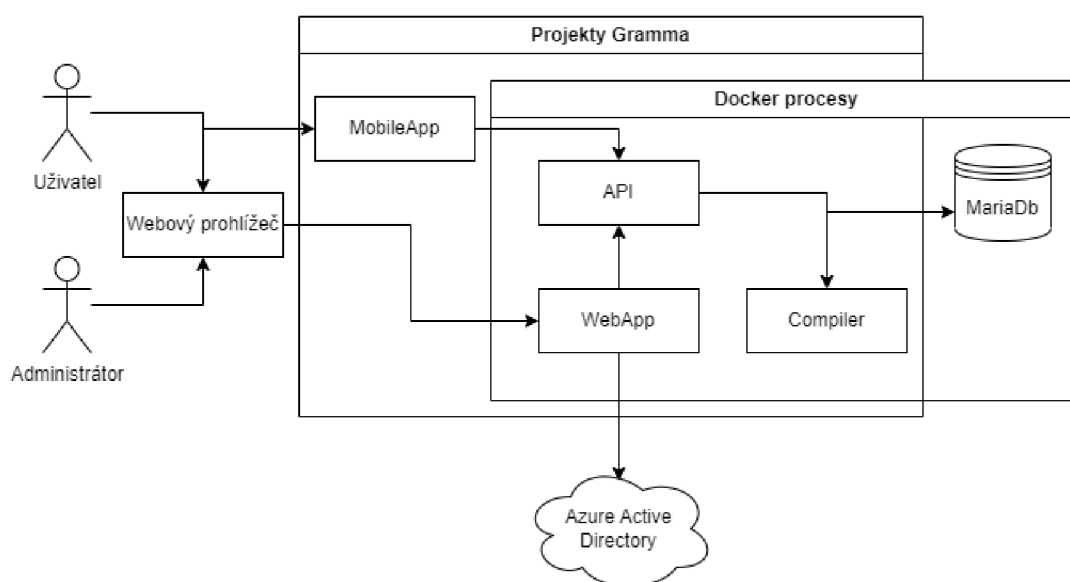
případě by slovníkové slovo mohlo být rovné kompletnímu slovu a gramatika funkcí identity, pak by ale přestala mít smysl pohnutka k tomuto rozdělení, totiž že slovníková slova obsahují jen některé základní informace a usnadňují tak zápis slov.

Poznámka 13

Italská přídavná jména jsou podle stejných pravidel jako zde implementována i v aplikaci. Dotyčné soubory jsou v data/languages/Italian/adjectives.

2 Implementace

Architektura systému je na Obrázku 2. Jádro aplikace tvoří čtyři procesy běžící v Dockeru. **MariaDb** je databází aplikace, ukládající jazyky, slovní zásobu, cvičení a související věci. **Compiler** slouží ke kompilování, ukládání a aplikování gramatických pravidel. **API** slouží k manipulaci s daty v databázi a compileru. **WebApp** slouží ke generování webových stránek a obsluhování požadavků na vyšší úrovni než API. Pro přihlášení administrátora se používá **Azure Active Directory**. Uživatelská i administrátorská část aplikace je dostupná skrze webový prohlížeč. Uživatelská část je dostupná i přes mobilní aplikaci **MobileApp**, která spolupracuje přímo s API.



Obrázek 2: Architektura systému

2.1 Databáze

Jako databáze byla použita MariaDB. Soubor na vytvoření počátečního stavu je Initialize.sql v projektu API. Při inicializaci získá databáze tabulky pro následující objekty: **language** - jazyk, **category** - slovní druh jazyka, **rewriter** - pravidla pro přepis znaků v jazyku, **exercise** - cvičení a **exercise_vocab** - vybraná slovní zásoba pro cvičení. Při přidávání nových slovních druhů pak vznikají další dva druhy tabulek, které v programu nazývám table-principal a table-complete.

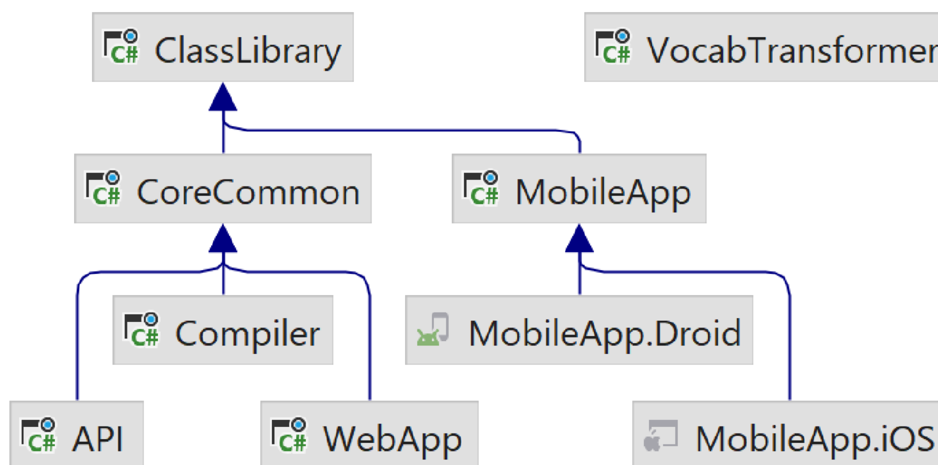
Table-principal, například Latin-nouns-principal pro latinská podstatná jména, slouží k ukládání slovníkových slov.

Table-complete, například Latin-nouns-complete, slouží k ukládání kompletních slov.

K převodu tvarů ze slovníkového na kompletní slouží pro každý slovní druh knihovna, které obsluhuje compiler.

2.2 Projektový diagram

Aplikace je naprogramována pomocí jazyka C# a využívá různé frameworky. Řešení se skládá z celkem devíti projektů, jejichž diagram se vzájemnými závislostmi je na Obrázku 3. **ClassLibrary** je knihovnou tříd, referencovanou z několika dalších projektů. Tato knihovna je vytvořená ve frameworku .NET Standard a mohou ji tedy referencovat všechny ostatní projekty. **CoreCommon** je také knihovnou tříd, ale obsahuje funkčnost sdílenou v projektech .NET Core 7. Tyto tři projekty v .NET Core jsou pak **Compiler**, **API** a **WebApp**. Všechny využívají technologie ASP.NET Core, pro tvorbu webových aplikací. První dvě využívají model controller-based APIs a WebApp využívá model Razor Pages. **MobileApp** je základním projektem mobilní aplikace, používající technologii Xamarin Pages. **MobileApp.Droid** a **MobileApp.iOS** jsou pak implementacemi mobilní aplikace pro Android a iOS. Tyto dva projekty byly vytvořeny převážně strojově, původní kód je hlavně v projektu MobileApp. Bokem stojí **VocabTransformer**. Jedná se o pomocný program na strojové vytvoření slovní zásoby pro latinu, který není nijak integrován do hlavní aplikace. Věnovat se mu budu v Příloze D.



Obrázek 3: Vytvořené projekty

2.3 ClassLibrary

Obsah knihovny tříd ClassLibrary bychom mohli rozdělit do tří částí. První jsou modely odpovídající položkám v databázi a pomocné třídy pro práci s nimi. To

Infinitive					
Active			Passive		
Present	Perfect	Future	Present	Perfect	Future
laudāre	laudāvisse	laudātūrum esse	laudārī	laudātum esse	laudātum īrī

Infinitive			
	Present	Perfect	Future
Active	laudāre	laudāvisse	laudātūrum esse
Passive	laudārī	laudātum esse	laudātum īrī

Obrázek 4: Tabulka při dvou různých šířkách

jsou Language, Category, RewriterRule, Exercise, WordPrincipal a WordComplete. Pomocné jsou Scope a Prefill, struktury, které používá Exercise, a výčtové typy ParamType a PrincipalInputType. Druhou částí je třída Utility, která obsahuje několik rozšiřujících metod, zejména pro práci s kolekcemi a textovými řetězci. Třetí skupinou je třída TableTree, který pomáhá při vytváření tabulek pro hierarchie určité Category, a několik jejich pomocných tříd - TableTreeNode, TableElement a ElementType.

2.3.1 Category a TableTree

Všechny druhy cvičení v aplikaci Gramma využívají tabulky s tvary slov. Administrátor však nikdy nemusí definovat rozložení konkrétní tabulky, definuje jen hierarchii tvarů v daném slovním druhu a převod na tabulky různých rozměrů už probíhá strojově. Výsledná tabulka ve dvou rozměrech je pro názornost na Obrázku 4.

Převod probíhá v následujících krocích, které následně popíšu podrobněji.

1. Administrátor definuje tvary a hierarchii pomocí odsazení řádků.
2. Zadaný text se převede na podobný, kde je ale hierarchie vyznačená závorkami. Tento text se i ukládá do databáze.
3. Hierarchie vyznačená závorkami je převedena na strom, kde listy odpovídají tvarům kompletního slova, ostatní uzly vyznačují hierarchii.
4. (volitelně) Část stromu může být odstraněna. Toho se využívá ve cvičeních, která pracují jen s částí tvarů.
5. Nastaví se obsah buněk. U HTML tabulek se používají šablony, takže se zde vloží jen značka pro pozdější přepis.
6. Strom se převede na seznam typu TableElement, kde jednotlivé prvky odpovídají částem tabulky, jako jsou například buňky s hlavičkami, s obsahem

nebo konce řádků, tak jak se vyskytují ve výsledné tabulce (po řádcích) za sebou.

7. Předchozí krok se opakuje vícekrát pro tabulky různé šířky.

8. Prvky z několika seznamů typu TableElement se spojí do jednoho tak, aby prvky z každého seznamu v něm byly zastoupeny ve svém původním pořadí, ale výsledná délka byla co nejkratší.

9. Prvky ve výsledném seznamu se převedou na HTML. Každý může být viditelný jen při určitých šířkách okna, čímž docílíme toho, že tabulka mění počet sloupců při změně šířky okna.

10. Výsledná HTML šablona se ukládá do databáze. Při posílání tabulky uživateli jsou nahrazené značky v šabloně z databáze skutečnými daty.

Takto probíhá vytváření tabulek pro webovou aplikaci. V mobilní aplikaci je proces trochu kratší: vytvoří se jen jeden seznam typu TableElement (stejný postup do kroku 6), který se pak převede na tabulku. Tabulky v mobilní aplikaci tak mají pevně danou šířku.

Nyní popíšu jednotlivé kroky podrobněji.

Každá kategorie má několik tvarů uspořádaných do jisté hierarchie. Pro demonstrační účely zde použiji jednoduchou kategorii o celkem čtyřech tvarech. Jedná se o přídavná jména v italském jazyce, která mohou být v jednotném nebo množném čísle a každé z toho pak v mužském nebo ženském rodě. Ve skutečnosti mohou kategorie obsahovat desítky nebo i stovky tvarů (například slovesa ve starověké řečtině by jich měla kolem tří set). Administrátor zadává strukturu s následující syntaxí (multi-line structure):

```
singular
 masculine
 feminine
plural
 masculine
 feminine
```

nebo zkráceně jen:

```
singular
 masculine
 feminine
plural*
```

kde * značí, že plural obsahuje tvary se stejnými názvy jako singular, tedy předchozí člen na stejné úrovni.

V databázi se kategorie ukládá ve tvaru structure, který používá uzávorkování k určení hierarchie. Kategorie z našeho příkladu by měla structure:

```
singular(masculine,feminine),plural*
```

Odsazení je zde nahrazeno závorkou, tvary na stejné úrovni jsou odděleny čárkou.

Převod mezi těmito dvěma tvary je poměrně jednoduchý a obstarává jej třída Category.

Třída TableTree následně převádí druhý tvar na strom, jak znázorňuje Algoritmus 1.

Algorithm 1 BuildNodes(structure, predecessor)

```
nodes ← new List of Nodes
parts ← divide(structure)
for part in parts do
    node ← new Node
    index ← index of '(' in part
    if index exists then
        node.name ← part up to index
        node.successors ← BuildNodes(substructure of part, node)
    else if part ends with '*' then
        node ← clone of last node in nodes
        node.name ← part without '*'
    else
        node.name ← part
    end if
    node.predecessor ← predecessor
    add node to nodes
end for
return nodes
```

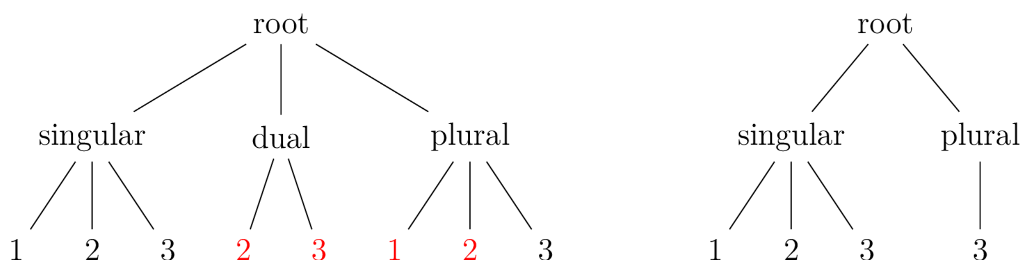
Takto vytvořený strom se pak převádí na seznam typu TableElement, předtím je ale možné použít filtraci, která ponechá jen určité listy a odebere všechny vnitřní uzly, které pod sebou nemají žádné ze zachovaných uzlů. Příklad stromu před a po filtraci je na Obrázku 5.

Funkci pro filtrování je popsaná v Algoritmu 2. Ta vybírá ponechané listy podle jejich indexů. V algoritmu je použito volání GenerateNodes(NextFromBottom). Funkce GenerateNodes vrací posloupnost uzlů vygenerovanou podle pravidla přechodu. Pravidlo NextFromBottom vrací první (nejlevější) list následujícího sourozence, nebo sourozence samotného, pokud je listem. Nemá-li další sourozence, vrací se předek. Začneme-li od prvního (nejlevějšího) listu stromu, pak tímto způsobem projdeme všechny uzly stromu s tím, že uzel je navštíven až poté, co byli navštíveni i všichni jeho potomci.

Před vytvořením tabulky, po případné filtraci, se ještě nastaví každému listu obsah buňky, která mu bude vytvořena, pomocí funkce SetContents. Nyní se přejde k vytvoření tabulky.

Algorithm 2 Filter(selector)

```
for leaf in leaves do
    leaf.CanRemove ← true
end for
for i in selector do
    leaves[i].CanRemove ← false
end for
for node in FirstLeaf().GenerateNodes(NextFromBottom) do
    if node.CanRemove and node.IsLeaf then
        pred ← node.Predecessor
        pred.Successors.Remove(node)
        pred.CanRemove ← true
    end if
end for
```



Obrázek 5: Strom před a po odfiltrování červeně označených listů.

Jak již bylo řečeno, jednou z vlastností tabulek ve webové aplikaci Gramma je měnit tvar podle šířky okna tak, aby byl počet sloupců menší při menší šířce, ale rozmístění dat v tabulce stále odpovídalo stejné hierarchii. Tato vlastnost komplikuje přímý převod ze stromu na HTML, a proto se strom převádí nejdříve do jiné struktury, kterou je seznam typu **TableElement**.

TableElement, reprezentující prvek tabulky, má následující vlastnosti:
Type - Typ prvku z výčtu ElementType. Může to být Content - buňka s obsahem, Empty - buňka bez obsahu, TopHeader - hlavička v záhlaví tabulky, SideHeader - hlavička na straně tabulky, TopLeft - prázdná buňka vlevo nahoře, v tabulce, která má hlavičky nahoře i po stranách, LineBreak - konec řádku a TableBreak - konec tabulky.
Name - Hlavičkové prvky mají jméno, tedy text který se v nich zobrazí.
Grow - Hlavičky nahoře mohou mít šířku jako více obyčejných buněk.

Zdrojový kód 1 ukazuje funkci GetTableElements, která převádí strom s kořenem rootNode na posloupnost objektů TableElement pro tabulku o šířce width. Šířkou je zde myšlen jen počet sloupců s obsahem. Tabulka může obsahovat ještě jeden sloupec postranních hlaviček.

(4-5) Pro šířku 1 je použit jednodušší algoritmus.

(7-9) V jiném případě se vytvoří prázdné seznamy pro TableElementy, hla-

vičky a uzly.

(10) Do proměnné node se přiřadí kořen.

(12-30) Následuje cyklus, který po každém průchodu nastaví jako aktuální uzel NextFromTop, což je první potomek, pokud uzel není list, jinak následující sourozenec. Pokud jej nemá, pak se vezme následující sourozenec prvního z předků, který jej má. Pokud začneme u kořene, projdeme tímto cyklem celý strom, s tím, že se vždy první projde celý podstrom aktuálního uzlu, než se posune dál. Cyklus skončí, když takový uzel už není, což znamená, že aktuální uzel byl posledním (nejpravějším) listem. Uvnitř cyklu:

(14-15) Pokud rootNode není předkem aktuálního uzlu, cyklus končí. Funkce se totiž může volat jen na část stromu, a v takových případech by jinak mohlo docházet ke vstupu do nechtěné části.

(16) Pokud má aktuální uzel více listů než je požadovaná šířka nebo jen jednoho potomka, přeskočí se na další uzel.

(18-19) Pokud je obsah všech listů tohoto uzlu prázdný, je celý podstrom vycházející z něj z tabulky vyřazen.

(20-24) Pokud je aktuální uzel list, je zavolána funkce ProcessLeaf, která přidá do elementList prvky podle aktuálního obsahu headers a nodes, tyto vyprázdní a také přidá prvky odpovídající aktuálnímu uzlu (listu). Přidá se také konec (pod)tabulky.

(25-29) Ve třetím případě, tedy že aktuální uzel není list a obsahy jeho listů nejsou prázdné je volána funkce ProcessNonLeaf, která zjistí, zda je aktuální uzel kompatibilní s uzly uloženými v nodes. Je-li to možné, uzel je přidán do nodes a do headers se dá sjednocení jejich hlaviček. V opačném případě jsou přidány TableElementy odpovídající uzlům v nodes a hlavičkám v headers, a následně jsou nahrazeny hodnotami odpovídajícími aktuálnímu uzlu. Průchod podstromu aktuálního uzlu je následně přeskočen.

(32-33) Až skončí cyklus, jsou přidány TableElementy pro uzly v nodes a hlavičky v headers, pokud nějaké zbývaly.

(35) Vrácen je seznam elementList, ze kterého byly odebrány postranní hlavičky, byly-li všechny prázdné.

Tímto způsobem získáme posloupnost typu TableElement. V mobilní aplikaci, kde je šířka neměnná, stačí vygenerovat jen jedinou posloupnost. Webová aplikace má, jak už bylo řečeno, schopnost měnit podobu tabulky podle šířky okna, takže se musí tato funkce zavolat pro každou podobu (zvolil jsem šířky 1, 4 a 8). Máme tedy několik posloupností, které chceme převést na HTML. Víme přitom, že pro TableElement typu Content se musí vytvořit jen jeden odpovídající HTML element pro všechny shodné výskyty v posloupnostech (tyto elementy může uživatel zpravidla upravovat a nechceme, aby se při změně velikosti okna zobrazil jiný). Ostatní prvky jsou sice neměnné, ale jelikož se opakují, bylo by dobré v možných případech ušetřit velikost celkově vygenerovaného HTML.

Na vyřešení tohoto problému jsem navrhl následující algoritmus. Každá ze vstupních posloupností se rozdělí na místech Content TableElementů. Těch je v každé posloupnosti stejně, takže při rozdělení získáme pro n Content TableElementů

n+1 částí posloupnosti. Poté vždy zpracujeme i-tou část posloupností, pak i-tý Content TableElement a úplně nakonec poslední část posloupností.

Při zpracování částí posloupností se řeší následující problém: Máme několik vstupních posloupností a chceme vytvořit novou posloupnost tak, že prvky z každé původní jsou v ní zastoupeny, a to v původním pořadí. Výsledná posloupnost by měla být co nejkratší.

Zpracování částí (větví) posloupností probíhá následovně. Dokud jsou alespoň v jedné větvi nezpracované prvky, opakuj následující: Vyber první vyhovující podmínku:

1. Existuje prvek, který je aktuálním ve všech větvích, kde se vyskytuje.
2. Existuje větev, u které se aktuální prvek později opakuje.
3. Vždy platí (vyber jakýkoliv prvek).

Podle vybrané podmínky vygeneruj příslušné HTML a posuň se v dotčených větvích.

Vytvořené HTML má u každého elementu class naznačující při kterých šířkách má být viditelný.

Tabulky se takto vytvářejí jen jednou pro každou kategorii a jednou pro každé cvičení (ty často používají filtrované tabulky). Vytvoří se jakási šablona, která se ukládá v databázi, následně se doplňují jen obsahy buněk.

Nevýhodou je zde, že když plníme šablony u některých defektivních slov (např. bez jednotného čísla nebo bez trpného rodu), nevyužité části se v tabulce zobrazí jako prázdná místa, zatímco v mobilní aplikaci, kde je tabulka vytvářena na straně klienta, tyto části vůbec nejsou. Nezdálo se mi vhodné tuto poměrně složitou operaci vytváření tabulky volat u každého dotazu na server, tak jsem zvolil vytvoření šablony, ale možná ještě lepším řešením by bylo generovat i zde tabulku na straně klienta, a to pomocí javascriptu.

```
1  IEnumerable<TableElement> GetTableElements
2      (int width, TableTreeNode rootNode)
3  {
4      if (width is 1)
5          return GetSingleColumnsTableElements(rootNode);
6
7      var elementList = new List<TableElement>();
8      var headers = new List<string>();
9      var nodes = new List<TableTreeNode>();
10     var node = rootNode;
11
12     do
13     {
14         if (!node.HasAncestor(rootNode))
15             break;
16         if (node.LeavesCount > width || node.Successors?.Count is 1)
17             continue;
18         if (node.Leaves().All(n => n.Content is ""))
19             node = node.LastLeaf();
20         else if (node.IsLeaf)
21         {
22             ProcessLeaf(elementList, nodes, headers, node);
23             elementList.Add(TableElement.TableBreak());
24         }
25         else
26         {
27             ProcessNonLeaf(width, nodes, node, elementList, headers);
28             node = node.LastLeaf();
29         }
30     } while ((node = node.NextFromTop()) is not null);
31
32     if (nodes.Any())
33         AddInnerElements(elementList, nodes, headers);
34
35     return TryRemoveSideHeaders(elementList);
36 }
```

Zdrojový kód 1: Funkce GetTableElements

2.4 CoreCommon

CoreCommon obsahuje funkcionalitu sdílenou projekty .NET Core, tedy Compiler, API a WebApp. Obsahuje middleware, route constraints a validaci.

Middleware jsou v ASP.NET Core metody, kterými prochází HTTP požadavek, než se dostane ke kontroleru. Využívají se k věcem jako je směrování požadavků, autentikace, autorizace a podobně. CoreCommon definuje jen jeden middleware na zachytávání výjimek, které loguje do konzole a v případě zachycení výjimky vrací HTTP status kód 500 - Internal server error.

RouteConstraints slouží k omezení platných hodnot v URL adrese. HTTP požadavky, jejichž adresy nevyhovují těmto požadavkům, jsou pak automaticky zamítány. CoreCommon definuje jedno takové omezení, které v dané části adresy dovoluje jen malá a velká písmena a-z a podtržítko a délku omezuje na 2 až 25 znaků, čehož se využívá u jmen jazyků a kategorií.

Validace slouží ke zkontrolování stavu objektů. Pro její implementaci byl použit balíček FluentValidation, který dovoluje definovat i poměrně složitá pravidla. U kontroleru je možné zapnout automatickou validaci, která automaticky zamítá HTTP požadavky, pokud jejich objekty validací neprošly.

2.5 Compiler

Účelem této servisy je kompilovat, ukládat a vykonávat kód, který odpovídá gramatickým pravidlům. Jelikož zde dochází k ukládání přímo na disk a servisa by se při reálném nasazení používala řídčeji proti běžnému API, rozhodl jsem se ji oddělit od rozsáhlejšího projektu API. Tím by také mělo být možné případně servisu API škálovat na více strojů a nenastal by problém s nesdíleným úložištěm.

Projekt Compiler je vytvořen pomocí ASP.NET Core controller-based API. Ta rozděluje jednotlivé HTTP endpointy do tříd nazývaných kontrolery, které mohou obsahovat několik endpointů a nějakou sdílenou funkcionalitu. Druhou nabízenou alternativou jsou minimal APIs, kde jsou všechny endpointy definovány přímo ve třídě Program. Je tak vhodná pro některé velice jednoduché servisy, ale myslím, že Compiler, ač mnohem jednodušší než následující dvě servisy (API a WebApp), je i tak na úrovni, že controller-based API je pro něj vhodnější.

Celkem Compiler obsahuje tři kontrolery: CodeController slouží ke stahování, načítání a mazání zdrojových kódů, FlexController slouží k ohýbání slov podle gramatických pravidel a ArchiveController k archivaci stavu aplikace. Seznam všech endpointů nabízených Compilerem je v Příloze C.

2.5.1 Fungování Compileru

Prostředí platformy .NET, Common Language Runtime, umožňuje za běhu načítat spustitelné programy a knihovny napsané v jemu kompatibilních jazycích. Jedná se především o trojici jazyků C#, F# a Visual Basic. Kompatibilní jsou ovšem i některé implementace C++, Pythonu a jiných. Možnost pracovat s více

různými jazyky jsem chtěl zavést i do této aplikace, a proto Compiler umožňuje práci s jazyky F# a C#.

Při obdržení zdrojového kódu Compiler vytvoří dočasný adresář, kde vygeneruje soubor projektu .fsproj nebo .csproj. Používat při kompilaci soubor projektu je v současné době standartní postup, ač existují i některé způsoby, jak program kompilovat jen ze zdrojového kódu bez projektu.

Následně Compiler spustí proces dotnet build, který má k dispozici jeho docker image. Pokud kompilace neskončila úspěšně, jsou objevené chyby navráceny zpět volajícímu a dočasný adresář se smaže.

V případě úspěšné kompilace je nový kód vyzkoušen. .NET nabízí třídu AssemblyLoadContext, do které můžeme za běhu programu načíst jinou knihovnu nebo program, a pak pomocí reflexe přistupovat k jeho třídám a metodám. Takto se tedy načte zkompilovaná knihovna a skrze reflexi se zjistí, zda obsahuje třídu, která jménem odpovídá kategorii daného jazyka, má konstruktor, jehož parametry přesně odpovídají typu slovníkového slova, a metodu getAllForms, která vrací Dictionary. Neprojde-li tento test, je dočasný adresář smazán, v opačném případě se přesune na trvalé místo pro danou kategorii jazyka (a případně nahradí již existující).

FlexController pak používá také reflexi, která umožňuje i vytvářet instance načtených tříd a volat načtené metody.

2.6 API

Projekt API je také vytvořen pomocí ASP.NET Core controller-based API. Slouží ke zprostředkovávání dotazů do databáze a Compileru. Celkem obsahuje 11 kontrolerů. Stejně jako u Compileru je seznam všech endpointů v Příloze C.

LanguagesController, RewriterRulesController, CategoriesController, ExercisesController a ExerciseVocabController slouží k manipulaci s jazyky, přepisovacími pravidly, kategoriemi, cvičeními a slovní zásobou cvičení.

TablesPrincipalController slouží k vytváření a popisování tabulek pro slovníková slova.

TablesCompleteController se používá k vytváření tabulek pro kompletní slova.

WordsPrincipalController se využívá na práci se slovníkovými slovy jakéhokoliv jazyka.

WordsCompleteController se používá na práci s kompletními slovy libovolného jazyka.

Dva z endpointů - **GET WordsComplete/{language}/search/{query}** a **GET WordsComplete/{language}/{category}/search/{query}** slouží k vyhledávání slov a využívají funkcionalitu nabízenou databází MariaDB, totiž matching in boolean mode, která dovoluje upravit hledaný řetězec tak, aby šlo vyhledávat například jen podle částí slov. Dotazem může být jeden tvar nebo více tvarů, oddělených mezerou. V tom případě proběhne sjednocení. Průnik se provede s tvary, které začínají na +, rozdíl s tvary, které začínají na -. Pokud má tvar na konci *, pak vyhledá všechny tvary, co takto začínají. Například

„gram*“ vyhledá všechna slova, která začínají na gram - gramen, grammatica, grammaticus. „gram* -grame*“ vyhledá slova, která začínají na gram, ale ne na grame - grammatica, grammaticus.

ArchiveController se užívá k archivaci stavu aplikace.

CodeController se používá k manipulaci se zdrojovými kódy gramatických pravidel. Slouží ke komunikaci s Compilerem, ale také umožňuje vytvořit šablonu pro konkrétní kategorii jazyka v podporovaných programovacích jazycích. Příklad šablony vygenerované pro jazyk F# je v kódu 2.

Nejdříve je zaveden název jmenného prostoru a závislosti. Poté jsou definovány výčtové typy, v tomto případě termination a hardness. Poté je definován samotný typ, adjectives: nejdříve konstruktor, následně hlavní funkce, která vytvoří slovník všech tvarů slova. Šablony typů, které používají příznaky by obsahovaly ještě další části, zde z důvodu místa neuváděné. Následují funkce pro každý tvar slova, které už musí doplnit administrátor. Právě zde se definují gramatická pravidla. Administrátor má k dispozici parametry slovníkového slova, v tomto případě řetězec stem a výčtové typy termination a hardness. Příklad doplnění jedné z funkcí je v kódu 3.

Porovnává se termination a hardness s několika hodnotami, přičemž _ znamená libovolnou hodnotu. Podle první shody se vrátí buď stem slova nebo jeho rozšíření o některý z řetězců.

Administrátor si pochopitelně může dopsat i pomocné funkce a proměnné a značně tím zkrátit délku kódu u složitějších kategorií. Například v mých latinských slovesech o přibližně sto padesáti tvarech je délka těla průměrné funkce jen kolem čtyř řádků. Zde uvedený způsob zápisu gramatických pravidel, podle jednotlivých tvarů, používám ve svých šablonách, ale není jediný možný.

Druhou možností by mohlo být určení jakéhosi vzoru slova podle zadaných parametrů, a poté podle něj jednoduše doplnit všechny tvary. Tento způsob je v jistých ohledech přirozenější lidskému způsobu učení gramatiky - například v češtině se učíme skloňovat celá jména podle vzorů a ne jednotlivé pády podle rodu, tvrdosti a podobně. Výhodou tohoto postupu by byl menší počet srovnání - bylo by nutné provést jen několik málo srovnání pro určení vzoru, pak už by se všechny tvary doplnily podle něj. Nevýhodou by však byla narůstající velikost zdrojového kódu v závislosti na počtu vzorů.

Třetí možností by bylo zkombinovat první dvě a mít jen několik základních vzorů, které mají mezi sebou značné rozdíly. Jednotlivé vzory by pak pro jednotlivé tvary mohly obsahovat doplňující srovnání.

```
1  namespace Italian
2
3  open System
4  open System.Collections.Generic
5
6  type termination =
7  | invariable = 0
8  | one = 1
9  | two = 2
10
11 type hardness =
12 | hard = 0
13 | soft = 1
14 | mixed = 2
15
16 type adjectives(stem:string, termination:string, hardness:string) =
17   member this.stem = stem
18   member this.termination = Enum.Parse<termination>(termination)
19   member this.hardness = Enum.Parse<hardness>(hardness)
20
21   member x.getAllForms() =
22     let dict = new Dictionary<string, string>()
23     dict.Add("singular_masculine", x.singular_masculine)
24     dict.Add("singular_feminine", x.singular_feminine)
25     dict.Add("plural_masculine", x.plural_masculine)
26     dict.Add("plural_feminine", x.plural_feminine)
27     dict
28
29   //Complete the following functions to correctly create the needed
        forms
30
31   member x.singular_masculine:string =
32     null
33
34   member x.singular_feminine:string =
35     null
36
37   member x.plural_masculine:string =
38     null
39
40   member x.plural_feminine:string =
41     null
```

Zdrojový kód 2: Příklad šablony pro jazyk F#

```
1  member x.singular_masculine:string =
2   match (x.termination, x.hardness) with
3    | (termination.invariable, _) -> x.stem
4    | (termination.one, _) -> x.stem + "e"
5    | (termination.two, hardness.soft) -> x.stem + "io"
6    | (termination.two, _) -> x.stem + "o"
```

Zdrojový kód 3: Dokončené gramatické pravidlo v jazyce F#

2.7 WebApp

V této části budou popsány hlavně technologie, které byly při tvorbě projektu WebApp použity, jednotlivé stránky budou popsány až v následující sekci Aplikace.

2.7.1 Razor

Projekt webové aplikace je vytvořen pomocí ASP.NET Core Razor pages. Webová aplikace je poskládána z jednotlivých stránek. Každá stránka je definována souborem .cshtml (page), který definuje její strukturu a odpovídajícím .csthml.cs (page model), který definuje model a obsluhu žádostí.

Druhým vzorem, který ASP.NET Core pro tvorbu stránek podporuje je MVC, tedy Model-View-Controller, kde jsou části více nezávislé. Model je obyčejná C# třída, která obsahuje proměnná data stránky. View je soubor .cshtml, který definuje strukturu stránek. Controller se stará o samotnou obsluhu žádostí, podobně jako kontrolery v API, ale často vrací stránky vytvořené podle View a Modelu. Výhodou tohoto přístupu je, že se tyto tři části dají různě kombinovat, například jeden model používat ve více views.

Ačkoliv se může zdát, že by toto mohlo být výhodou v mé aplikace, kde několik stránek existuje v podobě administorské a uživatelské, jediná, kde by se to reálně využilo by byla stránka výběru cvičení. Jiné stránky, ač se zabývají stejnými objekty, mají modely poněkud rozdílné, kvůli tomu, co s nimi provádějí, například administrátor cvičení upravuje, uživatel jich užívá.

Obecně se dá asi říci, že čím více je stránek, tím výhodnější by se začalo stávat používat vzor MVC. Moje aplikace se sedmi administrátorskými a šesti uživatelskými stránkami však spíše získává z jednoduššího přístupu, který nabízí razor pages.

Třetí nabízenou možností je Blazor, který má na rozdíl od prvních dvou jeden zásadní rozdíl. Zatímco v případě Razor pages a MVC jsou celé stránky generované na serveru, Blazor vytváří takzvané single page aplikace, u kterých si uživatel nejprve stáhne WebAssembly, které obsahuje funkčnost všech stránek, a následně si ze serveru stahuje jen data a stránky si skládá sám.

Největší výhoda, kterou by mohlo přinést použití Blazoru by bylo zjednodušení vyhodnocování cvičení, které by mohlo probíhat na straně klienta. V aplikaci jsem ale nakonec použil technologie AJAX, která dovoluje posílat jen části stránek a nenačítat tak při každé úrovni cvičení celou stránku znovu, takže celková výhoda z použití Blazoru by nebyla zas tak velká. Tímto jej ale nezavrhuji a myslím, že by mohl být také rozumným řešením pro tvorbu webového frontendu v tomto případě.

Nyní, když jsem srovnal tři hlavní možnosti tvorby webových stránek, popíšu blíže funkčnost té, kterou jsem pro projekt WebApp zvolil, tedy Razor pages. Stránka se většinou skládá ze dvou částí - page a page model (ten u některých jednoduchých stránek nemusí být).

Page je soubor s příponou .cshtml, napsaný v jazyce razor, který kombinuje HTML a C#.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome to Gramma</h1>
    <p class="text-primary">@Html.Raw(Model.Message)</p>
</div>
```

Zdrojový kód 4: Jednoduchá stránka v jazyce razor

V kódu 4 je příklad jednoduché stránky v razoru. Část kódu je běžné HTML, znakem @ jsou uváděny různé direktivy pro razor nebo vkládán C# kód. @page značí, že se jedná o razor page a obsluha se vykonává přímo v příslušném page modelu. U stránek, které to vyžadují, je na tomto místě uvedena i URL route příslušné stránky, například {Id}/{handler?}. @model nastaví odpovídající page model, v případě razor pages většinou odpovídá jeden page model přesně jedné stránce. Další dvě použití jsou už vložení C# kódu. V prvním z nich nastaví titulek stránky. Ve druhém vloží kus HTML kódu v řetězci Message, uloženém v modelu. Vkládaný kód se používá také k podmínění částí stránky nebo k jejich opakování s různými daty.

Odpovídající page model je ve zjednodušené podobě v kódu 5. Daná třída se vytvoří při každém požadavku na danou stránku. Poté je zavolána metoda k obsloužení konkrétního požadavku, v tomto případě OnGetAsync(), která nastaví řetězec Message.

Složitější stránky mohou obsahovat více metod pro obsluhu požadavků. Operace a URL část handler, na kterou se konkrétní metoda naváže, vychází pouze z jejího názvu, například u stránky Admin/Exercise/{Id}/{handler?} se metody naváží následovně:
OnGet > GET Admin/Exercise/{Id}
OnGetCopy > GET Admin/Exercise/{Id}/Copy
OnPost > POST Admin/Exercise/{Id}
OnPostRemoveComplete > POST Admin/Exercise/{Id}/RemoveComplete

Jinak je funkčnost page modelů poměrně přímočará. Vraťme se ještě k page a ukažme si některé další možnosti.

Další věcí, co jazyk razor nabízí jsou tag helpers. Vypadají jako značky HTML, ale při generování stránky se mohou změnit podle funkčnosti dané značky. Dají se využít například k jednoduššímu vytváření formulářů.

Jedním z tag helperů je <partial>, který slouží ke vložení takzvané partial page. To jsou kusy razor kódu, bez direktivy page, které mohou mít model.

```
1  using ...
2
3  namespace WebApp.Pages;
4
5  [AllowAnonymous]
6  public class IndexModel : PageModel
7  {
8     public string Message { get; set; } = "";
9
10    private readonly IApiClient _client;
11
12    public IndexModel(IApiClient client)
13    {
14       _client = client;
15    }
16
17    public async Task OnGetAsync()
18    {
19       var languages = (await _client.ApiGetRequest
20          <IEnumerable<Language>>("Languages"))?.ToArray();
21       if (languages.IsNullOrEmpty())
22            Message = "No languages found.";
23       Message = \$"Returned {languages.Count} languages.";
24    }
25 }
```

Zdrojový kód 5: Odpovídající page model

```
<partial name="_PrincipalInputsPartial"
         model="Model.PrincipalParts" />
```

vloží na dané místo kód, který generuje partial page _PrincipalInputsPartial s daným modelem.

Zvláštním souborem je _Layout.cshtml, který definuje sdílenou část stránek (navigaci, rozložení, patičku, ...) a v ní místa, kde se vkládá kód z konkrétních stránek.

2.7.2 Javascriptové knihovny

Využití javascriptu ve WebApp není příliš složité, používá se hlavně k zasílání požadavků bez nutnosti znovu načítat stránku a nějaké jednoduché funkcionalitě, jako je přepisování znaků, zobrazování zpráv, tabulek a podobně.

Byla použita knihovna jQuery, která je základní součástí stránkových aplikací v ASP.NET Core, a která umožňuje zjednodušení práce s DOM oproti čistému javascriptu.

Dále byla využita Unobtrusive AJAX, která umožňuje zařídit asynchronní fungování formulářů, bez nutnosti psát mnoho javascriptového kódu. Většinu

této konfigurace lze nastavit pomocí HTML atributů. Javascriptový kód se píše jen v některých případech, například pro zpracování získaných dat.

V menší míře byla použita také knihovna masonry, která umožňuje vzhledně zobrazovat dlaždice s obsahem o různých rozměrech a jednoduše je přeskládávat při změnách (tato knihovna je použita na stránce výběru cvičení).

2.7.3 Bootstrap

Pro vizualizaci byla použita CSS knihovna Bootstrap, která je také základní součástí stránkových aplikací v ASP.NET Core. Tato knihovna vytváří jednotný minimalistický vzhled. Je možné si ji vygenerovat pomocí CSS preprocesoru v upravené podobě. Tuto funkčnost jsem použil pouze pro změnu barev.

2.7.4 Microsoft Identity

Jelikož WebApp obsahuje jak uživatelskou, tak administrátorskou část, bylo žádoucí, aby se tyto části oddělily autorizací. Pro autorizaci jsem zvolil způsob, který by v reálných případech nebyl vždy dostačující, ale zde se mi zdál vhodný pro zjednodušení vyzkoušení administrátorské části aplikace - přístup do administrátorské části mají všichni přihlášení uživatelé.

Pro přihlášení jsem zvolil Microsoft Identity, jakožto jednu z možností, které ASP.NET Core aplikace nabízí mezi základními. Aby přihlášení správně fungovalo, musí být aplikace zaregistrovaná u některé organizace v Azure Active Directory. Univerzita Palackého jako organizace v Azure existuje, takže registrace aplikace byla snadná. Přístup k ní teď mají všichni s emailovým účtem univerzity. Kdyby se měla aplikace zaregistrovat pod jinou organizací, změna je snadná, jen se v appsettings.json vyplní nové údaje v sekci AzureAD.

Použití Azure pro přihlášení je zdarma a nepůsobí problémy použít jednou existující registraci aplikace pro přihlašování z různých lokálních instancí. Oproti tomu, kdybychom chtěli použít Azure i na jiné služby, například databázi nebo hostování kontejneru, musely by se pro každou instanci vytvářet samostatné, navíc jsou tyto služby zpoplatněné.

Samotné přihlášení funguje tak, že uživatel je přesměrován na server Microsoftu, který v případě úspěšného přihlášení vygeneruje cookie. Náš server pak pro autentizaci uživatele používá tyto cookies.

2.8 MobileApp

Mobilní aplikace nabízí přibližně stejnou funkčnost jako uživatelská část webové aplikace. Vytvořena byla pomocí Xamarin Forms. Tento framework umožňuje vytvářet mobilní aplikace pomocí kombinace XAML a C#. XAML je extensible application markup language, varianta XML, a zde slouží k definování struktury a vzhledu jednotlivých stránek, pomocí elementů jako jsou tlačítka, nápisy, poziční elementy a podobně. Příklad takového souboru je v kódu 6. Každá stránka má i odpovídající C# třídu, která řeší logiku.

```xml
<?xml version="1.0" encoding="utf-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
          xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
          x:Class="MobileApp.ConnectionConfigurationPage">
   <ContentPage.Content>
      <StackLayout>
         <Label Text="Please enter Url of API service."
               HorizontalOptions="CenterAndExpand" />
         <Entry x:Name="Entry"/>
         <Button Text="Set" Clicked="SetButton_OnClicked"/>
      </StackLayout>
   </ContentPage.Content>
</ContentPage>
```

Zdrojový kód 6: Příklad .xaml souboru stránky v Xamarin Forms

Ve skutečnosti je tedy způsob práce s Xamarin Forms podobný tomu s Razor pages, které mají stránky a modely. Důležitý rozdíl ale je, že vše běží na straně klienta, takže stránky se generují až u klienta a je možné je jednoduše upravovat pomocí C# kódu bez nutnosti komunikace se serverem.

Zvláštním souborem je App.xaml a jemu odpovídající App.xaml.cs, kde se nastavují sdílené věci pro aplikaci a spuštění včetně nastavení počáteční stránky.

Navigace v aplikacích Xamarin funguje formou zásobníku, otevřená stránka se vloží na vrchol zásobníku, zavřená se z vrcholu odstraní. Uživatel vidí vždy stránku na vrcholu, ale všechny zůstávají načtené v paměti.

2.8.1 MobileApp.Droid a MobileApp.iOS

Aplikace Xamarin se skládá z více projektů. Hlavní projekt, v tomto případě MobileApp obsahuje sdílený kód a je napsán v .NET Standardu. MobileApp.Droid a MobileApp.iOS jsou projekty, které využívají hlavní projekt a vytvářejí podle něj aplikaci pro konkrétní platformu, Android nebo iOS. V těchto projektech může vývojář nastavit věci, které jsou specifické u dané platformy, například barva lišty na Androidu, ikony a podobně.

Vývoj jsem dělal pro platformu Android. Aplikaci na iOS není možné bez příslušného hardware otestovat, nechávám ji v řešení jen pro úplnost, ale je možné, že v ní budou chyby, nebo nebude fungovat vůbec!

3 Aplikace

Následující sekce má tři části. V prvních dvou je přehled stránek v administrátorské a uživatelské části webové aplikace, třetí je o mobilní aplikaci.

3.1 Administrátorská část

V této podsekci budou obecněji popsány jednotlivé stránky administrátorské části. Obrázkový příklad vytvoření nového jazyka, kategorie, přidání slov a cvičení je v příloze B.

3.1.1 Přihlášení

Do administrátoské sekce se uživatel přihlásí kliknutím na Sign in vpravo na horní liště. Je přesměrován na stránky Microsoftu, kde se přihlásí účtem Univerzity Palackého.

3.1.2 Index

Tato stránka má dvojí funkčnost: Jednak obsahuje seznam jazyků dostupných v aplikaci. Administrátor může přidávat a odebírat jazyky, či vstoupit na stránku konkrétního jazyka.

Dále je možné si zde stáhnout archiv aplikace nebo načíst stav aplikace z archivu.

3.1.3 Language

Stránka language je pro konkrétní jazyk. Skládá se ze tří sekcí.

V sekci Categories může administrátor přidávat a odebírat kategorie, nebo vstoupit na stránku některé z nich.

V sekci Rewriter může administrátor nastavit pravidla pro přepis znaků v konkrétním jazyce. Pravidla se píší každé na samostatný řádek, levá a pravá strana jsou odděleny mezerou. Levá strana obsahuje řetězec, který má být nahrazen, pravá strana řetězec, který jej nahrazuje. Pokud bychom například chtěli usnadnit psaní německých znaků, mohli bychom zavést následující pravidla:

```
a: ä
A: Ä
o: ö
O: Ö
u: ü
U: Ü
s: ß
```

V některých administrátorských kontextech, ale především při plnění cvičení, jsou pak a: nahrazovány za ä a podobně. Přepisování znaků je důležité především pro webové aplikaci na počítači, kde by jinak mohlo psaní některých znaků

(například řeckých, ale často i diakritiky) působit značné problémy. V mobilních zařízeních není až takový problém si nainstalovat klávesnici pro daný jazyk, na počítači ale i v případě stažení dané klávesnice nemusí být jasné která klávesa píše který znak. Vytvořením vlastních pravidel můžeme docílit schopnosti psát cizí znaky více intuitivně.

Nakonec je zde ještě odkaz na administrátorskou stránku s cvičeními pro daný jazyk.

3.1.4 Category

Stránka Category je pro konkrétní kategorii. Skládá se ze čtyř sekcí.

V sekci Structure zadá administrátor strukturu kompletních slov dané kategorie, jak bylo popsáno v sekci 2.3.1. Například

```
singular
 masculine
 feminine
plural*
```

udává, že slova dané kategorie mají jednotné a množné číslo, z nichž každé má mužský a ženský rod.

Jakmile je nastavena struktura, vytvoří se pro danou kategorii defaultní cvičení, které však není spustitelné, dokud není přidáno alespoň jedno slovo.

V sekci Principal parts zadá administrátor, ze kterých částí se budou skládat slovníková slova dané kategorie. Celkem je na výběr pět typů:

Strings - řetězce. Na jednotlivé řádky názvy proměnných.

Integers - celá čísla. Na jednotlivé řádky názvy proměnných.

Enums - výčtové typy. Na jednotlivé řádky vždy název proměnné, mezera a možné hodnoty oddělené mezerami.

Booleans - pravdivostní hodnoty. Na jednotlivé řádky názvy proměnných.

Flags - pravdivostní hodnoty. Na jednotlivé řádky názvy příznaků. Tyto jsou ale uloženy společně v jednom 64bitovém čísle. Díky tomu je možné jednodušeji kontrolovat složitější podmínky. Tato proměnná by se měla použít hlavně pro vyznačení vzácných příznaků slova (například, že slovo je jen v množném čísle), aby bylo možné rychle zkontrolovat, že slovo žádné z těchto příznaků nemá a pokračovat při výpočtu běžnou cestou. Vhodným použitím by bylo zadávat zde jen takové příznaky, jejichž absence nemá zvláštní význam, zatímco v booleans by měly mít svůj význam obě možné hodnoty.

Až administrátor zadá tyto dvě sekce, stanou se neměnnými a zobrazí se další dvě sekce.

Sekce Grammatical rules slouží k nastavení gramatických pravidel. Uživatel si může stáhnout šablony pro dostupné programovací jazyky nebo současný soubor s gramatickými pravidly. Dokončený soubor s pravidly může nahrát.

Sekce Vocabulary slouží k práci se slovní zásobou. Na začátku je formulář vygenerovaný podle částí slovníkových slov, nastavených v sekci Principal parts.

Textová pole pro řetězce obsahují i ikonu tužky, kterou můžeme zapnout pro konkrétní pole přepisování znaků. To nemusí být vždy potřeba, například pokud chceme mít i překlad slova, nechceme, aby se znaky přepisovaly do jiné abecedy. Všechny části formuláře obsahují i ikonu zámku, kterým můžeme nastavit, aby si daná část ponechala svůj obsah po přidání slova. To může být vhodné, pokud přidáváme po sobě několik slov, která mají v dané části stejnou hodnotu. Zadané slovo se pak přidá tlačítkem Add Word. Po přidání se objeví tabulka s odpovídajícím kompletním slovem a tlačítko Edit, které odkazuje na odpovídající stránku Word.

Doposud popsaný způsob slouží k přidávání slov po jednom, což je dobré na vyzkoušení správnosti gramatických pravidel a přidání omezeného množství slovní zásoby. Pro větší množství slov by tento způsob ale byl velmi zdlouhavý. Další možností je načíst slova hromadně ze souboru typu .csv. Takový soubor se dá pro existující slova stáhnout tlačítkem Download Vocubulary. Pokud administrátor dokáže najít existující slovník a upravit jej do tohoto tvaru, může si tím usnadnit práci při přidávání slov. Ukázka získání slovní zásoby pro toto použití je popsána v Příloze D.

Sekce Vocabulary obsahuje také tlačítko Browse Vocabulary, které odkazuje na odpovídající stránku Words.

3.1.5 Words

Stránka na práci s existující slovní zásobou konkrétní kategorie jazyka. Slova jsou rozdělena do stránek po padesáti záznamech. U každého slova je několik prvních tvarů, případně je možné zobrazit všechny pomocí Show all forms. Dále je možné slovo smazat nebo upravit.

Další možností je slova přímo vyhledávat. Využívá se při tom stejný způsob vyhledávání, jaký bude popsán za chvíli u stránky Dictionary, hledá se však pouze v dané kategorii.

3.1.6 Word

Při úpravě je možné pracovat buď se slovníkovým slovem nebo s odpovídajícím kompletním slovem. Pokud se upraví slovníkové slovo, je odpovídající kompletní slovo upravené podle aktuálních gramatických pravidel a nového slovníkového slova. Při každé změně gramatických pravidel se přepočítá. Pokud se upraví přímo kompletní slovo, je označeno jako nepravidelné a budoucí změny gramatických pravidel na něj nemají vliv.

3.1.7 Exercises

Tato stránka obsahuje seznam cvičení v konkrétním jazyce. U cvičení každého je možnost jej upravit, vytvořit kopii, odstranit nebo spustit (přesměrování do uživatelské části). Dále je možné vytvořit nové cvičení pro konkrétní kategorii jazyka.

3.1.8 Exercise

Tato stránka slouží k vytvoření nebo úpravě cvičení. U každého cvičení se nastavují následující vlastnosti:

Název a **popis**.

Jazyk a **kategorie** (doplněné automaticky).

Počet úkolů v cvičení (opakování chybných úkolů na konci cvičení se do tohoto čísla nepočítá).

Zobrazený úkol - text, který se zobrazí při každém úkolu. Součástí textu mohou být výrazy tvaru {p.name} nebo {c.name}, které se nahradí hodnotou name ve slovníkovém slově (p) nebo v kompletním slově (c). Například „Conjugate {c.infinitive_active_present}:“ se u konkrétního slova může zobrazit jako „Conjugate laudāre:“.

Výběr slovní zásoby:
ALL - ve cvičení se povolí všechna slova daného slovního druhu.
RULE - pravidlo podle kterého SQL vybere slova například „‘conjugation‘ = 1“ nebo „‘rareBits‘ NOT LIKE "%deponent%"“.
SELECTION - výběr konkrétních slov podle id nebo některého z tvarů. Pokud je na začátku řádku číslo, vybere se slovo s tímto id. Jinak se vybere slovo, které má všechny z tvarů uvedených na řádku, oddělených mezerou.

Typ cvičení - Vyplnění, nalezení, výběr nebo mix. Jednotlivé typy cvičení budou více ukázány v uživatelské části.

Tvary k vyplnění - administrátor vybere, které z tvarů kompletního slova budou ve cvičení, zaškrtnutím v tabulce. Je možné rychleji vybrat několik tvarů stisknutím a tažením myši, případně vybrané tvary smazat tlačítkem Clear.

Předvyplněné tvary - ve cvičení může být několik tvarů předvyplněných. Administrátor může po vybrání několika tvarů dát Add All, což znamená, že vybrané tvary budou předvyplněné vždy všechny, nebo může dát Add n Random of Selected, což znamená, že z vybraných se v každém úkolu předvyplní jen n náhodně vybraných. Tyto výběry je možné libovolně kombinovat. Clear All smaže všechny výběry. Clear Selected smaže aktuálně vybrané, ale ještě nepřidané tvary.

3.2 Uživatelská část

3.2.1 Index

Na domovské stránce jsou uvedeny některé jazyky pro rychlou navigaci k příslušným cvičením.

3.2.2 Languages

Na této stránce je seznam jazyků s počty cvičení a odkazy.

3.2.3 Exercises

Na této stránce je seznam cvičení pro konkrétní jazyk. U Každého je uveden název, popis, počet úkolů a ikona podle typu cvičení. Uživatel si může seznam vyfiltrovat na konkrétní kategorii.

3.2.4 Exercise - typ Fill

Tento typ cvičení je označen klávesnicí. Uživatel musí napsat každý tvar kompletního slova, který chybí v tabulce. Využívá při tom přepisování znaků tak, jak je definované pro konkrétní jazyk.

Toto cvičení je časově nejnáročnější, ale nejlépe se na něm dokáží skutečné znalosti.

3.2.5 Exercise - typ Find

Tento typ cvičení je označen checkboxem. Uživateli je dán jeden tvar a musí v tabulce označit všechna místa, kde se může vyskytovat.

Toto cvičení je nejrychlejší, ale procvičí se na něm vždy jen malá část tvarů daného slova. Může však být dobré k naučení některých vzorů v opakování určitých tvarů.

3.2.6 Exercise - typ Select

Tento typ cvičení je označen šipkou. Uživateli jsou dány obsahy všech chybějících políček v tabulce, musí je umístit na správné místo. Stačí nahoře vybrat tvar a pak kliknout na správné místo. Pokud se splete, stačí kliknout na chybné místo v tabulce a to se nahradí novým tvarem, nebo se vymaže, pokud nebyl žádný vybrán. Starý tvar je možné umístit na jiné místo.

Toto cvičení je jakýmsi kompromisem mezi oběma předchozími. Uživatel si na něm vyzkouší všechny tvary, ale jeho splnění netrvá tak dlouhou dobu. Nevýhodou však je, že všechny tvary jsou uvedeny, takže je nemusí znát zpaměti.

Vzhled cvičení typu Select je na Obrázku 6. Vzhled prvních dvou je podobný.

Existuje také cvičení typu Mix, které náhodně střídá tyto tři výše představené.

3.2.7 Dictionary

Kromě cvičení má uživatel k dispozici také slovník, kde může vyhledat tvar v konkrétním jazyce. Při psaní dotazu se používá přepisovač znaků konkrétního jazyka. Dotazem může být jeden tvar nebo více tvarů, oddělených mezerou. V tom případě proběhne sjednocení. Průnik se provede s tvary, které začínají na +, rozdíl s tvary, které začínají na -. Pokud má tvar na konci *, pak vyhledá všechny tvary, co takto začínají. Například „gram*“ vyhledá všechna slova, která začínají na gram - gramen, grammatica, grammaticus. „gram* -grame*“ vyhledá slova, která začínají na gram, ale ne na grame, tedy grammatica, grammaticus.

Latin nouns

Fill in the missing forms.

ōvis ōvem ōvibus ōvēs ōve ōvēs ōvium ōvibus ōvī

	Nominative	Genitive	Dative	Accusative	Ablative
Singular	ōvis	ōvis		ōvem	ōve
Plural	ōvēs		ōvibus		

Send

Obrázek 6: Příklad cvičení typu Select ve webové aplikaci

3.3 Mobilní aplikace

Mobilní aplikace obsahuje přibližně stejnou funkčnost jako uživatelská část webové aplikace, ale v některých místech upravenou tak, aby více vyhovovala mobilním zařízením.

3.3.1 Připojení

Při reálném použití by existoval nějaký server s pevně danou adresou, který by mobilní aplikace používala jako zdroj svých dat. Jelikož však je potřeba, aby si každý mohl zprovoznit celou aplikaci lokálně, nemůžeme o takovém přístupu uvažovat. Proto jsem zavedl stránku ConnectionConfigurationPage, která se zobrazí vždy při spuštění aplikace, a kde uživatel zadá adresu serveru pro připojení. Je to řešení, kterým jsem se snažil co nejvíce zjednodušit lokální zprovoznění aplikace. Postup, jak si zpřístupnit lokálně běžící server pro jeho použití v mobilní aplikaci je v první příloze.

3.3.2 MainPage

Při úspěšném spárování se serverem se octneme na této stránce. Kombinuje v sobě stránky Index a Languages a navigační lištu uživatelské části webové aplikace. Můžeme se odsud přesunout na stránku slovníku nebo si vybrat jeden z jazyků.

3.3.3 ExerciseSelectionPage

Při výběru jazyka se dostaneme na tuto stránku, která odpovídá stránce exercises ve webové aplikaci, ale ve zjednodušené podobě. Máme zde přehled cvičení v daném jazyce, ale s méně podrobnostmi.

3.3.4 ExercisePage

Při výběru cvičení se přesuneme na tuto stránku. Podporuje všechny typy cvičení, které byly představené v uživatelské části webové aplikace. Při úspěšném splnění cvičení je uživatel navrácen zpět na předchozí stránku.

3.3.5 DictionaryPage

Stránka slovníku funguje stejně jako ve webové aplikaci. Uživatel si vybere jazyk a hledané slovo. Podle toho se vytvoří tabulky.

Obrázek 7: Stránka slovníku a cvičení v mobilní aplikaci

Závěr

Mým cílem bylo vytvořit aplikaci na učení lidských jazyků se zaměřením na gramatiku. Výsledná aplikace nabízí několik druhů cvičení. Dále měla být aplikace použitelná pro libovolné jazyky. V testovacích datech jsem zanechal implementace několika jazyků které ovládám, včetně velmi složité řečtiny. Správce aplikace má k dispozici rozhraní k vytváření pravidel a slov, ale myslím si, že některé části mohly být dokonalejší, zejména definování struktury slov a částí slovníkového slova. Dále byla vytvořena mobilní aplikace, která obsahuje přibližně stejnou funkčnost jako webová aplikace. Nejsem si však příliš jistý, zda je pro existující druhy cvičení zcela vhodná, protože často obsahují velké tabulky, hůře zobrazitelné na mobilních zařízeních. Co se týče počtů ukázkových cvičení, vytvořil jsem jich v latině kolem třiceti, dále jednotky v češtině, angličtině, italštině a řečtině, což je myslím dostačující na ukázání možností aplikace.

Celkově si myslím, že aplikace svůj účel splňuje. Pro skutečné nasazení by si však zasloužila ještě některá další vylepšení. Jednalo by se především o větší personalizaci jednotlivým uživatelům. Dobrým vylepšením by mohlo být vytvoření učebních plánů, které by se skládaly z cvičení a stránek názorně vysvětlujících gramatická pravidla, přičemž postup uživatelů by se ukládal. Tímto by se aplikace stala přívětivější pro uživatele a trochu více nezávislou na dalších učebních pomůckách. Některá vylepšení by si zasloužila i administrátorská část. Kromě výše zmíněných například možnost vytvoření jednoduchých kategorií bez nutnosti definovat slovníková slova a gramatická pravidla. I skryté implementační detaily by se mohly vylepšit, například zdokonalit způsob ukládání slov v databázi, více přizpůsobit projekt API skutečným scénářům nebo pokrýt řešení jednotkovými testy.

Conclusions

My aim was to create an application for learning natural languages, focused on grammar. The final app offers multiple kinds of exercises. Next, the app was to be usable for any language. In the test data, I have left implementations of many languages I know, including the difficult Greek. The administrator has access to an interface for creating grammatical rules and vocabulary, but I believe that some parts of it could be better, especially the defining of structure and principal parts of words. Next, a mobile app was created, which contains more or less the same functionality as the web application. However, I am not quite sure it is suitable for the existing kinds of exercises, as they often require large tables, difficult to display on mobile screens. As for the sample exercises, I created about thirty in Latin and a few in Czech, English, Italian and Greek, which I believe is enough to show the possibilities the app offers.

All in all, I believe the app fulfills its purpose. For a real deployment however, it would deserve some more improvements. This would especially mean better personalization for the users. A good improvement would be the creation of learning plans, which would consist of exercises and pages explaining rules of grammar, while the progress of a user would be saved. This way, the app would become more pleasant to use and a bit less dependent on other learning resources. The administration section would also deserve some improvements. Aside from those already mentioned, it would be for instance adding the option to create simple categories without the need to define principal parts of words or grammatical rules. There could be some improvements in the hidden implementation details as well, for example to improve the way of saving vocabulary in the database, to make the API project more suitable for real scenarios or to cover the solution with unit tests.

A Zprovoznění aplikace

A.1 Jádro aplikace v Dockeru (fyzický stroj Windows 10)

Ačkoliv se aplikace skládá z několika samostatných servis, snažil jsem se, aby bylo jejich zprovoznění co nejjednodušší, proto jsem celou aplikaci dělal tak, aby byla spustitelná z Dockeru.

Vývoj jsem dělal na počítači s Windows 10. Jelikož však není možné používat docker ve virtuálním stroji s Windows, je tento postup použitelný jen pro fyzické stroje. Přesto jej doporučuji jakožto rychlejší.

1. Vytvořte HTTPS certifikát následujícími příkazy, poslední schvalte:

   ```
   dotnet dev-certs https --clean
   dotnet dev-certs https `
     -ep $env:USERPROFILE\.aspnet\https\aspnetapp.pfx `
     -p HermesArgeiphontes
   dotnet dev-certs https --trust
   ```

   Zde uvedený formát je pro PowerShell, v případě jiného shellu nahraďte $env:USERPROFILE příslušným ekvivalentem.

   V případě, že nemáte k dispozici příkaz dotnet, je ke stažení zde: dotnet.microsoft.com/en-us/download.

2. Stáhněte a zprovozněte si docker docs.docker.com/get-docker/

3. Z příkazové řádky ve složce src spusťte následující:

   ```
   docker compose up --build
   ```

   Tato část může poprvé trvat několik minut.

4. Webová aplikace Gramma nyní běží na https://localhost:7171

Poprvé spuštěná aplikace neobsahuje žádná data. Po přihlášení je možné na stránce https://localhost:7171/admin nahrát archiv „data/archives/Custom Exercises.zip“, čímž se zpřístupní mnou vytvořené jazyky, cvičení a podobně.

A.2 Jádro aplikace v Dockeru (virtuální stroj Ubuntu Server 22.04 LTS)

Druhou možností je aplikaci spustit ve virtuálním stroji Ubuntu.

1. U virtuálního stroje nastavte Port Forwarding:

```
Devices > Network > Network Settings
Advanced > Port Forwarding
0.0.0.0:7171 0.0.0.0:7171
0.0.0.0:5210 0.0.0.0:5210
```

2. Rozšiřte disk na povolený rozsah (nedělo se to automaticky, což způsobilo zaplnění disku):

```
sudo lvm
lvextend -l +100%FREE /dev/ubuntu-vg/ubuntu-lv
exit
sudo resize2fs /dev/ubuntu-vg/ubuntu-lv
```

3. Nainstalujte Docker podle docs.docker.com/engine/install/ubuntu/:

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch="$(dpkg --print-architecture)" \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" \
  stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

4. Nainstalujte .NET SDK:

```
sudo apt-get install -y dotnet-sdk-6.0
```

5. Vytvořte HTTPS certifikát:

```
sudo dotnet dev-certs https --clean
sudo dotnet dev-certs https \
  -ep /usr/local/share/ca-certificates/aspnet/aspnetapp.pfx \
  -p HermesArgeiphontes
```

6. Stáhněte si software práce a přesuňte se do složky src

7. Sestavte a spusťte aplikaci příkazem:

```
sudo docker compose -f docker-compose.ubuntu.yml up --build
```

8. Webová aplikace Gramma nyní běží na `https://localhost:7171` (hostující i virtuální stroj). Certifikát je v tomto případě instalace označován za nebezpečný.

A.3 Přístup mobilní aplikací zvenčí pomocí ngrok

1. V mobilním telefonu si nainstalujte aplikaci ze souboru cz.upol.Gramma-signed.apk.

2. V počítači (fyzický) s běžící aplikací si nainstalujte ngrok. ngrok.com/download

3. Zaregistrujte si na ngrok účet, registrací získáte authtoken.

4. V příkazové řádce spusťte následující (se svým tokenem):

   ```
   ngrok config add-authtoken <token>
   ```

5. V příkazové řádce spusťte následující:

   ```
   ngrok http 5210
   ```

   Nyní začne ngrok přesměrovávat žádosti.

6. Spusťte mobilní aplikaci a zadejte adresu vygenerovanou ngrokem (forwarding).

7. Mobilní aplikace má nyní přístup k serveru.

B Vytvoření latinského cvičení v aplikaci, krok za krokem

1. Spusťte jádro aplikace Gramma v Dockeru podle postupu v předchozí sekci.

2. Kliknutím na Sign in se přihlaste s platným účtem Univerzity Palackého.

3. Přesuňte se do sekce Admin (na horní liště). V případě, že jste už dříve nahráli archiv s daty, vraťte aplikaci do výchozí podoby nahrátím archivu data/archives/Empty.zip.

4. Do kolonky name v části Languages napište název přidávaného jazyka (Latin) a klikněte na Add. Následně klikněte na novou buňku Latin, která se objevila. (Obrázek 8)

Obrázek 8: Přidání nového jazyka.

5. Přidání pravidel pro přepis znaků. Do sekce Rewriter vložte obsah souboru data/Languages/Latin/rewriter.txt a klikněte na Update.

6. Přidání nové kategorie. Do kolonky name v sekci Category napíšte její název (nouns) a klikněte na Add. Následně klikněte na novou buňku nouns. (Obrázek 9)

7. Nastavení struktury a částí slovníkových slov kategorie. Ze souboru data-/Languages/Latin/nouns/inputs.txt přesuňte vstupy do příslušných kolonek a stiskněte u obou částí Set. (Obrázky 10 a 11)

8. Po nastavení obou částí se objeví dvě nové sekce - Vocabulary a Grammatical Rules. Nejprve nastavíme gramatická pravidla. V sekci Grammatical Rules, Upload vyberte soubor data/Languages/Latin/nouns/Latin-nouns.fs a klikněte na Upload.

Latin

Back to Admin page

Categories

nouns Remove

name Add

Rewriter

a- ā
e- ē
i- ī
o- ō
u- ū
y- ȳ
á ā
é ē
í ī
ó ō

Update

Obrázek 9: Stránka Language po přidání nové kategorie.

9. Nyní přistoupíme k přidávání slovní zásoby, nejprve ručně po jednotlivých slovech. V části Add Single Word vyplňte pole například podle obrázku (využívá se přitom přepisování znaků, takže o- se přepíše na ō) a klikněte na Add Word. (Obrázek 12)

10. Níže se objevila tabulka, kliknutím na Edit můžeme slovo upravit. Při úpravě buď můžeme změnit slovníkové části slova, nebo přímo tvary kompletního slova. Takto můžeme nastavit tvary nepravidelným slovům, jako například na obrázku. Kliknutím na Update se slovo upraví. (Obrázek 13)

11. Vraťte se zpět na stránku Latin nouns. Přidávat slovní zásobu je možné i ze souboru. V Sekci Add multiple words from file vyberte soubor Latin-nouns.csv a klikněte na Upload. Zpracování všech slov může chvíli trvat, ale vše se provádí na pozadí, takže můžete pokračovat ihned.

12. Vraťte se zpět na stránku Latin (odkaz je nahoře) a odtud pokračujte na stránku Exercises (Go to Exercises).

13. Jako kategorii vyberte nouns a klikněte na Create New Exercise.

Structure

singular
 nominative
 genitive
 dative
 accusative
 ablative
 vocative
 locative
plural*

Split level | 0

Set

Obrázek 10: Nastavení struktury kategorie.

14. Nastavte požadované vlastnosti cvičení, například podle obrázku (u Prefilled Forms jsem zvolil stejných osm jako u Forms to Complete a klikl na Add n Random of Selected), poté klikněte na Add Exercise. (Obrázek 14)

15. Nyní jsme úspěšně přidali nový jazyk s kategorií, slovní zásobou a cvičením. Vše je dostupné v uživatelské sekci.

Principal parts

Strings	nom root
Integers	declension
Enums	gender M F N
Booleans	hasLocative
Flags	noSingular noPlural ium ablI longEI plurIA ubus

Set

Obrázek 11: Nastavení slovníkových částí slov kategorie.

Vocabulary

Add single word

nom	bōs
root	bōv
declension	3
gender	M
hasLocative	
flags	noSingular noPlural ium ablI

Add Word

Obrázek 12: Přidání slova ručně.

	Nominative	Genitive	Dative	Accusative	Ablative	Vocative	Locative
Singular	bōs	bōvis	bōvī	bōvem	bōve	bōs	
Plural	bōvēs	boum	bōbus	bōvēs	bōbus	bōvēs	

Update

Obrázek 13: Úprava slova ručně.

Creating new Exercise

Back to Exercises

Basic settings

Name	Latin Example exercise
Description	An example exercise of Latin nouns
Language	Latin
Category	nouns
Number of stages	2
Displayed task	Conjugate (p.nom).
Selection of vocabulary	Selection
Selected vocabulary	lacus genū
Type of exercise	Fill

Forms to complete

	Nominative	Genitive	Dative	Accusative	Ablative	Vocative	Locative
Singular	✓	✓	✓	✓			
Plural	✓	✓	✓	✓			

Clear

Prefilled forms

	Nominative	Genitive	Dative	Accusative	Ablative	Vocative	Locative
Singular	✓	✓	✓	✓			
Plural	✓	✓	✓	✓			

Clear Currently Selected | Clear All | Add All | Add n Random of Selected | 1 | R1(0,1,2,3,7,8,9,10)

Add Exercise

Obrázek 14: Konfigurace cvičení.

C Endpointy Compileru a API

V této příloze je seznam všech endpointů vystavovaných službami Compiler a API.

C.1 Compiler

Služba Compiler používá celkem tři kontrolery.

C.1.1 CodeController

První kontroler slouží ke stahování, načítání a mazání zdrojových kódů.
GET Code/{language}/{category}
Vrátí aktuální zdrojový kód pro danou kategorii jazyka.
PUT Code/{language}/{category}/{extension}
Pokusí se zkompilovat nahraný zdrojový kód, a pokud se to podaří a kód je kompatibilní s daným jazykem, uloží si jej.
DELETE Code/{language}/{category}
Smaže zdrojový kód pro danou kategorii jazyka.

C.1.2 FlexController

Tento kontroler slouží k ohýbání slov podle gramatických pravidel.
POST Flex/{language}/{category}
Všechna zaslaná slovníková slova převede na kompletní slova podle aktuálně uložených gramatických pravidel pro danou kategorii.

C.1.3 ArchiveController

Poslední kontroler slouží k archivaci stavu aplikace.
GET Archive
Stáhne všechny aktuálně uložené soubory jako .zip archiv.
PUT Archive
Přepíše své uložené soubory podle obsahu obdrženého .zip archivu.

C.2 API

Služba API používá celkem jedenáct kontrolerů.

C.2.1 LanguagesController

První kontroler slouží k manipulaci s jazyky. Podobně i další čtyři slouží k manipulaci vždy s jednou tabulkou v databázi.
GET Languages
Vrátí všechny jazyky.
GET Languages/{name}

Vrátí jazyk s označením name.
POST Languages
Přidá jazyk do databáze
DELETE Languages/{name}
Odebere z databáze jazyk s označením name a všechny související kategorie, slova, cvičení a zdrojové kódy.

C.2.2 RewriterRulesController

GET RewriterRules/{language?}
Vrátí pravidla pro přepis pro jazyk language nebo pro všechny jazyky.
PUT RewriterRules/{language}
Nahradí pravidla pro jazyk language novými pravidly.

C.2.3 CategoriesController

GET Categories/{language?}
Vrátí kategorie pro jazyk language nebo všechny kategorie.
GET Categories/{language}/{name}
Vrátí kategorii name pro jazyk language.
POST Categories
Přidá kategorii.
PUT Categories/{language}/{name}
Nahradí kategorii name pro jazyk language novou.
DELETE Categories/{language}/{name}
Odebere z databáze kategorii name pro jazyk language a všechna související slova, cvičení a zdrojové kódy.

C.2.4 ExercisesController

GET Exercises/{language?}
Vrátí cvičení pro jazyk language nebo všechna cvičení. Vracené objekty obsahují jen základní data, například šablony tabulek nejsou zasílány.
GET Exercises/{id}
Vrátí cvičení s daným id.
GET Exercises/counts/{language?}
Vrátí počet cvičení pro jazyk language nebo počty pro všechny jazyky.
POST Exercises
Přidá cvičení.
PUT Exercises
Nahradí cvičení.
DELETE Exercises/{id}
Odstraní cvičení s daným id.

C.2.5 ExerciseVocabController

GET ExerciseVocab/ids/{exercise}
Vrátí id všech slov v daném cvičení.
GET ExerciseVocab/complete/{exercise}
Vrátí všechna kompletní slova pro dané cvičení.
PUT ExerciseVocab/{exercise}
Nahradí slovní zásobu pro cvičení. Slova jsou zde zadána jako kolekce řetězců, kde každý může být buď id slova, nebo některý z jeho tvarů.

C.2.6 TablesPrincipalController

Tento kontroler slouží k vytváření a popisování tabulek pro slovníková slova.
GET TablesPrincipal/{language}/{category}
Vrátí pole specifikující tabulku pro slovníková slova dané kategorie jazyka. Pole obsahuje pět prvků - řetězce, celá čísla, výčtové typy, pravdivostní hodnoty a příznaky.
POST TablesPrincipal/{language}/{category}
Vytvoří tabulku pro slovníková slova dané kategorie jazyka.

C.2.7 TablesCompleteController

Tento kontroler se používá k vytváření tabulek pro kompletní slova.
POST TablesComplete/{language}/{category}
Vytvoří tabulku pro kompletní slova dané kategorie jazyka se zadanými názvy sloupců.

C.2.8 WordsPrincipalController

Tento kontroler se využívá na práci se slovníkovými slovy jakéhokoliv jazyka.
GET WordsPrincipal/{language}/{category}
Vrátí slovníková slova pro danou kategorii jazyka. Nepovinné parametry: irregular - vrací jen pravidelná/nepravidelná slova. idFrom, idTo - počáteční a koncové id.
GET WordsPrincipal/{language}/{category}/{id}
Vrátí slovníkové slovo dané kategorie jazyka se zadaným id.
GET WordsPrincipal/{language}/{category}/file
Vrátí vygenerovaný soubor .csv, obsahující informace o slovníkových slovech dané kategorie jazyka.
POST WordsPrincipal/{language}/{category}
Přidá slovníkové slovo dané kategorie jazyka. Pokud pro ni existuje i knihovna s gramatickými pravidly, vygeneruje příslušné kompletní slovo a rovněž jej přidá.
POST WordsPrincipal/{language}/{category}/file
Přidá ze souboru slovníková slova dané kategorie jazyka. Pokud pro ni existuje i knihovna s gramatickými pravidly, vygeneruje příslušná kompletní slova a rovněž je přidá.

PUT WordsPrincipal/{language}/{category}
Nahradí slovníkové slovo dané kategorie jazyka. Pokud pro ni existuje i knihovna s gramatickými pravidly, vygeneruje příslušné kompletní slovo a rovněž jej nahradí.
DELETE WordsPrincipal/{language}/{category}/{id}
Odstraní slovníkové slovo s daným id z dané kategorie jazyka. Pokud existuje i odpovídající kompletní slovo, odstraní jej rovněž.

C.2.9 WordsCompleteController

Tento kontroler se používá na práci s kompletními slovy libovolného jazyka.
GET WordsComplete/{language}/{category}
Vrátí všechna kompletní slova pro danou kategorii jazyka.
GET WordsComplete/{language}/{category}/{id}
Vrátí kompletní slovo s daným id pro danou kategorii jazyka.
GET WordsComplete/{language}/{category}/page/{page}
Vrátí kompletní slova pro danou kategorii jazyka na zadané padesátislovní straně.
GET WordsComplete/{language}/search/{query}
Vrátí všechna kompletní slova v daném jazyce, odpovídající hledanému výrazu.
GET WordsComplete/{language}/{category}/search/{query}
Vrátí všechna kompletní slova v dané kategorii jazyka, odpovídající hledanému výrazu.
POST WordsComplete/{language}/{category}
Přidá kompletní slovo do dané kategorie jazyka.
POST WordsComplete/{language}/{category}/multi
Přidá několik kompletních slov do dané kategorie jazyka.
PUT WordsComplete/{language}/{category}/{id}
Nahradí kompletní slovo s daným id v dané kategorii jazyka. Slovo je označeno jako nepravidelné.
DELETE WordsComplete/{language}/{category}/{id}
Odstraní kompletní slovo s daným id z dané kategorie jazyka.

C.2.10 ArchiveController

Tento kontroler se užívá k archivaci stavu aplikace.
GET Archive
Stáhne aktuální stav aplikace jako .zip archiv. Součástí archivu jsou adresáře se zdrojovými kódy z Compileru a soubor gramma.sql, který obsahuje stav databáze.
PUT Archive
Přepíše stav databáze a Compileru podle obdrženého .zip archivu.

C.2.11 CodeController

Poslední kontroler se používá k manipulaci se zdrojovými kódy gramatických pravidel.
GET Code/{language}/{category}/{type}
Stáhne aktuální kód, C# nebo F# šablonu pro gramatická pravidla daného jazyka.
PUT Code/{language}/{category}
Předá kód ke kompilaci Compileru a v případě úspěchu i znovu vygeneruje kompletní slova podle nových pravidel.

D VocabTransformer

Projekt VocabTransformer vznikl jako ukázka strojového vytvoření slovní zásoby do aplikace Gramma. Ze vstupního souboru dokáže vytvořit soubory .csv, vložitelné do Grammatu, obsahující řádově tisíce podstatných jmen, přídavných jmen a sloves.

D.1 Popis

Jako vstup používám soubor la_vocab.txt, který obsahuje na každém řádku čtyři řetězce, oddělené tabulátorem, například:

```
capax   a-s---nv-   capax   ca^pa_x
capedo  n-s---fn-   capedo  ca^pe_do_
cepi    v1sria---   capio   ce_pi_
clonius n-s---mn-   Clonius Clo^nius
```

První sloupec obsahuje tvar slova bez vyznačení délek, přízvuků a velkých písmen.

Druhý sloupec obsahuje v zakódované formě slovní druh slova a některé další informace k jeho zařazení (osoba, číslo, čas, způsob, slovesný rod, jmenný rod, pád, stupeň). Význam jednotlivých znaků je popsán v souboru „data/VocabTransformer/latin vocab explanation.txt“.

Třetí sloupec obsahuje lemma, které je stejné pro různé tvary téhož slova.

Čtvrtý sloupec obsahuje stejný tvar slova jako první sloupec, ale s vyznačením délek, přízvuku a velkými písmeny.

Tento zdrojový soubor jsem převzal z projektu Latin macronizer, https://github.com/Alatius/latin-macronizer, který umožňuje automaticky vyznačit délky v latinském textu.

Samotný program VocabTransformer pak spouští čtyři funkce.

První je spuštěna funkce SortCategories, která vstupní soubor rozdělí na několik menších souborů, každý pro jeden slovní druh. Určit slovní druh pro daný řádek je jednoduché, stačí se podívat na první symbol v druhém sloupci.

Další tři funkce jsou si podobné, každá zpracovává jeden slovní druh.

1. Postupně načítá řádky z příslušného souboru, dokud patří ke stejnému lemmatu (řádky ve vstupním souboru již jsou seřazené podle lemmat).

2. Pro každé lemma si pamatuje jeho jméno (třetí sloupec) a možné podoby (upravený čtvrtý sloupec) pro každý tvar slova (druhý sloupec).

3. Podle získaných informací se snaží zjistit hlavní části slovníkového slova: deklinaci a kmen u podstatných jmen, deklinaci, kmen a terminaci u přídavných jmen a konjugaci a tři kmeny u sloves. K zjištění deklinace (něco jako třída skloňování) či konjugace (třída časování) používá objekt typu Dictionary, ve kterém jsou uloženy možné koncovky pro daný tvar v dané deklinaci či konjugaci, například (singulár dativu podstatných jmen):

```
["sd"] = new[]
{
    new[] { "ae" },
    new[] { "ō" },
    new[] { "ī" },
    new[] { "uī", "ū" },
    new[] { "ēī", "eī" }
}
```

Každý z pětice řádků zde udává možné koncovky jedné z pěti deklinací.

4. Pokud nebylo možné určit přesně jednu deklinaci či konjugaci ze získaných informací o slově, je zamítnuto a jeho data zapsána do souboru zamítnutých slov (můžeme jej používat při dalším vylepšování programu).

5. V opačném případě se zjistí i ostatní části slovníkového slova a zapíší se do výstupního souboru ve tvaru, který uznává aplikace Gramma.

6. Po zpracování každého slovního druhu se do konzole vypíše počet přijatých a zamítnutých slov.

To je základní princip trojice funkcí pro podstatná jména, přídavná jména a slovesa. Každá se v některých detailech od tohoto postupu liší, ale na popis fungování to stačí.

D.2 Shrnutí výsledků

Podstatná jména dosáhla výsledku 17928 přijatých slov a 4829 zamítnutých, což je téměř 80% přijatých, nejlepší výsledek z trojice slovních druhů. Mezi zamítnutými slovy jsou buď ta, která mají ve vstupním souboru málo tvarů, takže je nebylo možné jednoznačně určit, nebo slova řeckého původu, jejichž skloňování je mírně odlišné, a s kterým jsem ve svých gramatických pravidlech neuvažoval.

Z přídavných jmen bylo 7234 přijatých a 2369 zamítnutých. Celkově je tedy zhruba 75% přijatých. Většina zamítnutí je způsobena malým počtem tvarů ve vstupním souboru. Často mají slova jen komparativ nebo superlativ, které jsou u obou deklinací stejné, a tudíž není možné ji určit.

Ze sloves bylo 3304 přijatých a 4468 zamítnutých. Je to nejslabší výsledek, s jen 42% přijatých. Je to způsobeno více vlivy. Jednotlivé konjugace v latině jsou v mnohých tvarech často stejné, takže v mnoha případech ve vstupním souboru chybí tvary, podle kterých by bylo možné jednoznačné určení. Moje gramatická pravidla pro slovesa jsou trochu zjednodušená, takže musím zamítat i některá slova, která by se při dokonalejších pravidlech mohla přijmout. Některá slova není možné určit, protože v Dictionary s koncovkami nemám uvedené všechny možné tvary, ale jen ty nejčastější. Tento vliv by však měl být minimální.

Celkově tedy máme několik tisíc slov z každého slovního druhu, což by mělo stačit k tomu, aby se jednotlivá slova příliš neopakovala a uživatel se mohl naučit podstatu gramatických pravidel místo toho, aby si je spojoval s konkrétními slovy.

Funkci na podstatná jména by bylo možné rozšířit, aby přijímala i slova řeckého původu. Funkce na přídavná jména se zdá být blízko svých možností. Funkce na slovesa by mohla být zdokonalena, znamenalo by to ale potřebu přepracovat i odpovídající gramatická pravidla.

E Obsah adresářové struktury

bin/

Soubor cz.upol.Gramma-Signed.apk pro instalaci mobilní aplikace na telefon s operačním systémem Android.

doc/

Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/

Kompletní zdrojové texty aplikace Gramma a soubor docker-compose.yml, sloužící ke spuštění jádra webové aplikace v Dockeru.

README.txt

Instrukce pro instalaci a spuštění aplikace Gramma, včetně všech požadavků pro její bezproblémový provoz.

data/

Ukázková a testovací data použitá v práci a pro potřeby testování práce při tvorbě posudků a obhajoby práce.

Bibliografie

[1] Andrew Troelsen, Philip Japikse. *Pro C# 7: With .NET and .NET Core.* 8th ed. edition. Apress, 2017. ISBN: 978-1-4842-3018-3

[2] Mark Reynolds. *Xamarin Mobile Application Development for Android.* Packt Publishing, 2014. ISBN: 978-1-7835-5083-8

[3] Robert Pickering, Kit Eason. *Beginning F# 4.0.* Second Edition. Apress, 2016. ISBN: 978-1-4842-1375-9

[4] *Microsoft Learn.* https://learn.microsoft.com/, Poslední přístup 15. 4. 2023

[5] *Learn Razor Pages.* https://www.learnrazorpages.com/, Poslední přístup 15. 4. 2023

[6] *Bootstrap Documentation.* https://getbootstrap.com/docs/5.3/, Poslední přístup 15. 4. 2023

[7] *Pluralsight.* https://app.pluralsight.com/, Poslední přístup 15. 4. 2023

[8] Meagan Ayer. *Allen and Greenough's New Latin Grammar for Schools and Colleges.* Carlisle, Pennsylvania: Dickinson College Commentaries, 2014. ISBN: 978-1-947822-04-7 https://dcc.dickinson.edu/grammar/latin/credits-and-reuse, Poslední přístup 15. 4. 2023

[9] Meagan Ayer, ed. *Goodell's School Grammar of Attic Greek.* Carlisle, Pennsylvania: Dickinson College Commentaries, 2018. ISBN: 978-1-947822-10-8. https://dcc.dickinson.edu/grammar/goodell/credits-and-reuse, Poslední přístup 15. 4. 2023

[10] Thomas Goodwin. *Latin-English Dictionary.* London Lockwood & Co., 7, Stationers' Hall Court Ludgate Hill, 1874. https://upload.wikimedia.org/wikipedia/commons/7/70/Latin-English_dictionary_%28IA_latinenglishdict00good%29.pdf, Poslední přístup 23. 4. 2023

[11] Hans H. Ørberg. *Lingua Latina per se Illustrata, Pars I: Familia Romana.* Domus Latina, 2003. ISBN: 978-87-997016-5-0