FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	dbs.fit.cvut.cz - tvorba otázek a štítků
Student:	Tomáš Douba
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 2023/2024

Pokyny pro vypracování

Cílem této práce je tvorba nových backendových mikro služeb se zaměřením na tvorbu/úpravu testových otázek a dále navazujících štítků pro obecnou kategorizaci. Výsledek práce bude součástí většího ekosystému a bude volně navazovat na magisterskou práci Ing. Andriiho Plyskache.

Postupujte v těchto krocích:
1. Analyzujte současný stav portálu se zaměřením na tvorbu testů a obecnou kategorizaci. Nezapomeňte na provázanost s ostatními částmi systému, na kterých souběžně pracují další studenti.
2. Na základě analýzy proveďte důkladný návrh s ohledem na další součásti backendu nově vznikající verze portálu dbsfit.cvut.cz.
3. Dle návrhu realizujte použitelný backend pro tvorbu/úpravu testů s důrazem na snadnou udržitelnost a rozšiřitelnost.
4. Při vývoji řádně testujte. Navrhněte a aplikujte vhodné testy s ohledem na budoucí vývoj této části backendu.
5. Navrhněte budoucí směr rozvoje, shrňte dosažené výsledky.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 6. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

dbs.fit.cvut.cz - Tvorba otázek a štítků

Tomáš Douba

Katedra Softwarového Inženýrství
Vedoucí práce: Ing. Jiří Hunka

11. května 2023

Poděkování

Chtěl bych poděkovat Ing. Jiřímu Hunkovi za vedení této práce, konzultace, cenné rady a hlavně trpělivost. Dále bych chtěl poděkovat Ing. Andriiji Plyskachovi za vhled a rady do DBS portálu. V neposlední řadě bych chtěl poděkovat své rodině a svým blízkým přátelům za podporu v průběhu studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu 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 předpisu. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel licenční smlouvu o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisu.

V Praze dne 11. května 2023 . . . . . .

Odkaz na tuto práci

Douba, Tomáš. *dbs.fit.cvut.cz - Tvorba otázek a štítků*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Práce se zabývá problémem rozčleňování existujícího systému do mikroslužeb. Systém je určen pro výuku databází a mezi relevantní části patří správa testového zadání, otázek a správa štítků. Štítky dovolují uživateli vyhledávání a filtrování objektů. S oddělením štítků do vlastní izolované služby je potřeba vytvořit syntaktický analyzátor, díky kterému se tato funkcionalita neztratí. Tento analyzátor dovolí zpracovat řetězec se štítky s logickými operátory do databázových dotazů.

Klíčová slova mikroslužba, syntaktický analyzátor, LL(k) parser, REST, PHP, Doctrine, Symfony, SQL, ORM, Docker

Abstract

The thesis deals with the problem splitting existing application into microservices. The application is used for teaching database systems. Relevant parts of the system contains test assignment and question management and tag management. Tags lets user search and filter objects. With the isolation of tags into its own service, a syntactic analyser is needed to keep such feature. The analyser will

be able to compile string input with tags and logical operations into database queries.

Keywords microservice, syntactic analyser, LL(k) parser, REST, PHP, Doctrine, Symfony, SQL, ORM, Docker

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době hledají organizace softwarová řešení, která jsou škálovatelná, flexibilní a snadno udržovatelná. Jedním z přístupů, který v posledních letech získal na popularitě, je architektura mikroslužeb. Jak název napovídá, architektura je založená na rozdělení monolitické aplikace do menších soběstačných služeb. Termín „mikroslužba“ existuje ve světě softwarého vývoje krátce přes 10 let. Mezi známé organizace, které tuto architekturu používají, patří Netflix, Amazon nebo Uber.

Portál DBS, dále jen portál, slouží jako doprovodný výukový nástroj pro výuku v předmětech BI-DBS, BI-SQL a jejich ekvivalentech pro anglické a kombinované studium. Postupem času se stával portál těžší na údržbu, úpravu nebo i škálování – důvody, kvůli kterým se vývojáři často obracejí k mikroslužbám. Práce čtenáře seznámí s hlavními pojmy a technologiemi nynějšího a navrhovaného portálu.

Tato práce navazuje na bakalářskou a diplomovou práci Ing. Andriije Plyskache, který detailně popisuje návrhy současného a budoucího portálu. Vývoj nad portálem je spojen s bakalářskou prací Jakuba Pavlička a diplomovou prací Radoslava Haška.

Cíl práce

Cílem práce je analyzovat současný stav portál. S použitím analýzy bude vytvořen návrh tříd, databázového schématu a rozhraní, pomocí kterého bude možné se systémem komunikovat. Výsledkem budou dvě mikroslužby zaměřené na správu zadání a otázek, a na správu štítků. V rámci štítků bude výstupem práce také návrh syntaktického analyzátoru pro výrazy obsahující štítky s logickými operátory. K samotné implementaci bude součástí výstupu také dokumentace mikroslužeb a syntaktického analyzátoru nejen na implementované funkcionality, ale i možnosti rozšíření.

Kapitola 1

Teorie

Tato kapitola se bude věnovat definicím pojmů, které budou použité v kapitolách analýzy a návrhu.

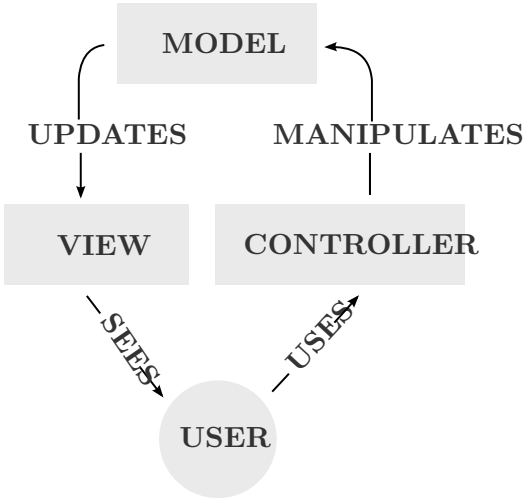
1.1 Architektura

1.1.1 Model-view-controller

Architektura model-view-controller rozděluje aplikaci do tří vrstev, každá s jiným účelem.

- Datová vrstva, model, zahrnuje definici datových entit a spravuje jejich perzistenci.
- Prezentační vrstva, view, zobrazuje data z datové vrstvy.
- Aplikační vrstva, controller, reaguje na uživatelský vstup a upravuje dle něho data datové vrstvy.

Cílem rozdělení aplikace je oddělení business logiky od uživatelského rozhraní. [4, 5]



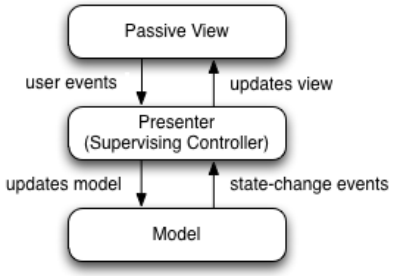
Obrázek 1.1: The model, view, and controller (MVC) pattern relative to the user [1]

1.1.2 Model-view-presenter

Obdobně jako model-view-controller, architektura model-view-presenter se skládá ze tři vrstev. Mezi rozdíly v rozložení vrstev patří

- Prezentační vrstva, view, má na starost zobrazování dat z datové vrstvy a získávání příkazů od uživatele pro doménovou vrstvu.
- Aplikační vrstva, presenter, obdrží vstup od prezentační vrstvy a s definovanou logikou manipuluje s datovou vrstvou.
- Nižší provázanost mezi modelem a view, kde presenter slouží jako středník.

Role a název komponenty datové vrstvy zůstává stejná. [5] [6]



Obrázek 1.2: Model-view-presenter diagram [2]

1.1.3 Mikroslužba

Architektura mikroslužeb je přístup k tvorbě aplikací. Spočívá v rozdělení do mnoha nezávislých, volně spojených menších modulů nebo služeb. Služby mezi sebou komunikují přes rozhraní příkladem REST a každá má vlastní prostředí včetně databází a správy dat a modelů. [7] Mezi výhody této architektury patří

- snadná škálovatelnost a snadnější vývoj
- moduly lze nasazovat na různé servery dle potřeby
- možné použití různých technologií a jazyků pro různé moduly
- jednodušší testování a integrace automatizovaných systémů
- lepší izolace chyb a snížení vlivu chyb na zbytek aplikace
- aktualizace, úpravy nebo nahrazení modulů bez vlivu na zbytek aplikace [8]

1.1.4 Slowly Changing Dimensions

Slowly Changing Dimensions (SCD) je rozměr dat, který obsahuje, jak nynější, tak i historická data v datovém skladu. Většinou se jedná o statická data, která se mohou pomalu a neplánovaně měnit. Příkladem jsou názvy zeměpisných lokací, zákazníci nebo produkty. Existuje více typů přístupů k SCD. [9, 10]

1.1.4.1 Typ 0

Atributy entity nemění a zachovává se původní data. [10]

1.1.4.2 Typ 1

Data jsou přepsána a původní data jsou zahozena. [9]

1.1.4.3 Typ 2

Tento typ ukládá kompletní historii hodnot. Při změně dojde k vytvoření nového záznamu a archivaci původního. K archivaci se mohou používat časové značky od-do použitelnosti nebo časová značka vytvoření záznamu a booleovskou hodnotou. [9]

1.1.4.4 Typ 3

Typ 3 ukládá omezený počet verzí hodnoty pro vybraný atribut záznamu. Je omezen tím, kolik je přiřazeno sloupců, například pro dvě verze se uchovává nynější hodnota a předchozí hodnota. [9]

1.2 Parser

Definice 1.2.1. (Abeceda [11]) *Abeceda* Σ je konečná množina prvků, symbolů abecedy.

Definice 1.2.2. (Řetězec [11]) *Řetězec* je konečná posloupnost symbolů abecedy. Prázdný řetězec značíme ε a množinu všech řetězců nad abecedou Σ^*.

Definice 1.2.3. (Bezkontextová gramatika [12]) *Gramatika* je uspořádaná čtveřice $G = (N, \Sigma, P, S)$, kde

- N je neprázdná množina symbolů, *neterminální symboly*,
- Σ je konečná množina symbolů, *terminální symboly*. Pro množiny N a Σ platí $N \cap \Sigma = \emptyset$.
- P je konečná množina přepisovacích pravidel ve tvaru $\alpha A \beta \rightarrow \gamma$, kde $\alpha, \beta, \gamma \in (N \cup \Sigma)^* \wedge A \in N$.
- S je počáteční neterminální symbol gramatiky. Platí, že $S \in N$.

Bezkontextová gramatika je gramatika, která má přepisovací pravidla ve tvaru $A \rightarrow \alpha$, kde $\alpha \in (N \cup \Sigma)^* \wedge A \in N$.

Definice 1.2.4. (Derivační strom [11, 12]) *Derivační strom* je zakořeněný uspořádaný strom, kde

- vrcholy stromu jsou tvořeny neterminálními, terminálními symboly nebo ε,
- list může být ε je-li jediným synem svého rodičovského vrcholu,
- vrchol s alespoň jedním synem je ohodnocen neterminálním symbolem,
- kořenem derivačního stromu je počáteční neterminální symbol.

Definice 1.2.5. (Zásobníkový automat [11, 12]) *Zásobníkový automat* (ZA) je uspořádaná sedmice $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ abeceda,
- G je abeceda zásobníku,
- δ je přechodová funkce ve tvaru $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times G^* \rightarrow 2^{Q \times G^*}$,
- q_0 počáteční stav,
- Z_0 je počáteční symbol zásobníku,

- F je množina koncových stavů a platí $F \subseteq Q$,

Definice 1.2.6. (Řetězec přijímaný zásobníkovým automatem [11, 12]) Zásobníkový automat *přijímá řetězec w* je-li přijímán přechodem do koncového stavu nebo prázdným zásobníkem. Řetězec je přijímán

- přechodem do koncového stavu, skončí-li automat v koncovém stavu a přečetl celý vstup,
- prázdným zásobníkem, přečetl-li celý vstup a zásobník je prázdný.

1.2.1 Syntaktická analýza

1.2.1.1 Syntaktická analýza shora dolů

Syntaktická analýza shora dolů je metoda analýzy řetězců bezkontextové gramatiky. Tato metoda začíná v kořeni derivačního stromu s počátečním symbolem a postupně se aplikují přepisovací pravidla, které rozšiřují strom o symboly pravých stran přepisovacích pravidel. [13, 14]

1.2.1.2 LL syntaktický analyzátor

Jedním z analyzátorů shora dolů je LL syntaktický analyzátor. LL je zkráceně pro **L**eft to right **L**eftmost derivation. Tento analyzátor zpracovává vstupní řetězec zleva doprava a tvoří nejlevější derivaci. Gramatiky, které se dají takto analyzovat, se nazývají LL(k) gramatiky, kde k je počet následujících symbolů, které je potřeba během analýzy znát. [13, 14]

1.2.1.3 LL(1) gramatika

LL(1) gramatika je bezkontextová gramatika, pro kterou platí omezení pro deterministickou analýzu:

- Je potřeba znát 1 následující symbol.
- Gramatika nesmí být zleva rekurzivní.
- Pro každou kombinaci neterminální symbolu a terminálního symbolu lze jednoznačně určit přepisovací pravidlo, tedy existuje nejvýše jedno přepisovací pravidlo.

[15, 16]

1.2.1.4 LL(1) parsovací tabulka

Parsovací tabulka pro LL(1) gramatiky je tabulka zobrazující všechny kombinace neterminálních a terminálních symbolů. K každé buňce se nachází nejvýše jedno přepisovací pravidlo. Pro sestrojení této tabulky je potřeba konstrukce dvou množin $FIRST$ a $FOLLOW$ pro každý neterminální symbol. Pro neterminální symbol X představuje množina $FIRST(X)$ všechny terminální symboly, kterými mohou řetězce vytvořené symbolem X začínat. Množina $FOLLOW(X)$ obsahuje všechny terminální symboly, které symbol X hned mohou následovat. [15, 16]

1.2.1.5 Metoda rekurzivního sestupu

Metoda rekurzivního sestupu je metoda analýzy shora dolů, kde pro každý neterminální symbol existuje funkce nebo procedura, která ho zpracuje. Má-li neterminální symbol přepisovací pravidlo s dalšími neterminální symboly na pravé straně, potom v těle funkce dochází k rekurzivnímu volání na daný symbol. [17]

1.2.1.6 Metoda pomocí parsovací tabulky

Metoda pomocí parsovací tabulky je metoda analýzy shora dolů. Oproti rekurzivnímu sestupu se používá parsovací tabulka pro předpovědění přepisovacího pravidla podle nynějšího symbolu a nadcházejícího vstupu. Metoda vyžaduje předem zkonstruovanou parsovací tabulku. Samotný algoritmus průchodu řetězcem používá jeden zásobník a odkazuje se na tabulku při rozhodování, které pravidlo použít. [17]

Fragment kódu 1 Pseudo-algoritmus pro analýzu LL(1) gramatiky

```
Initialize stack
Add Start symbol to stack
token := Read token from input

while Stack is not empty
    symbol := Pop from stack

    if symbol == token
        token := Read next symbol
    else if symbol is non-terminal
        rhs := Get right hand side symbols of production
        Push right hand side symbols to stack
    else
        Syntax error
```

Kapitola 2

Analýza

Tato kapitola popíše současné řešení portálu se zaměřením na testové zadání a otázky, a štítky. Zmíněné části budou dále analyzovány na funkční a nefunkční požadavky a případy užití.

2.1 Současný systém

Portál se dá rozdělit do více menších celků: správa semestrálních prací, správa uživatelů, správa databázových připojení, správa testů a další. Pod správou testů nalezneme zadání a otázky, na které se v této práci zaměříme. Mezi technologie, na kterých portál běží, patří PHP 7, Nette nebo PostgreSQL.

2.1.1 Technologie

2.1.1.1 PHP

PHP je open-source skriptovací jazyk určený pro vývoj webových stránek a může být zakomponován do HTML. Kód jazyka je vyhodnocován se straně serveru, ale lze jej spustit i v rámci příkazové řádky. [18] PHP obsahuje mnoho funkcí a knihoven, které usnadňují práci s webovými stránkami a databázemi. Může být také použit pro manipulaci s daty a generování obsahu v závislosti na uživatelském vstupu.

2.1.1.2 Nette

Nette je open-source framework napsaný v PHP, který se zaměřuje na tvorbu rychlých a bezpečných webových aplikací. Framework je modulární a sám o sobě obsahuje jen základní funkcionality. Další komponenty je možné přidávat dle požadovaných funkcí. [19] Jedná se o framework typu model-view-presenter, popsané v sekci 1.1.2.

2.1.1.3 PostgreSQL

PostgreSQL nebo Postgres je objektově relační databázový systém, zaměřený na bezpečnost, výkon a rozšiřitelnost. Nabízí vysokou úroveň přizpůsobení, příkladem tvorba vlastních datových typů, vlastní funkce nebo použití jiného programovacího jazyka. [20] založený na jazyku SQL a je open-source. PostgreSQL se stále vyvíjí, inovuje a pravidelně jsou vydávány nové verze.

2.1.1.4 Vagrant

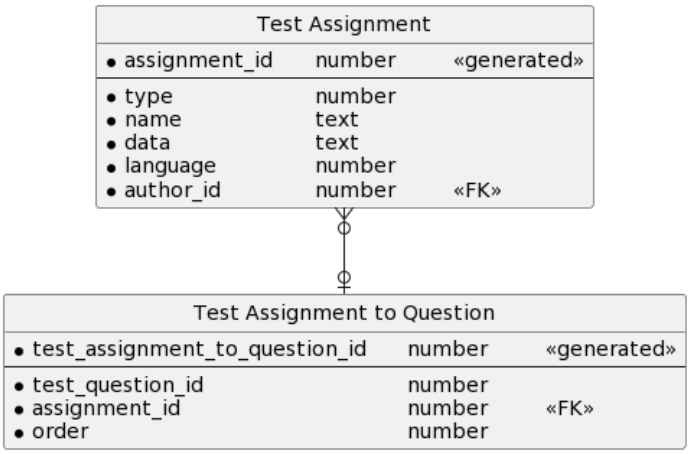
Vagrant je open-source nástroj pro automatizaci vytváření a správu virtuálních strojů. Umožňuje tvorbu konfigurovatelných, reprodukovatelných a přenositelných vývojových prostředí. Podporuje různé poskytovatele virtualizace jako VirtualBox, Hyper-V, VMWare nebo AWS. [21]

2.1.2 Testové zadání

Než se začne popisovat struktura testových zadání, je potřeba rozlišit, co si má čtenář pod slovem zadání představit. Tato práce bude používat slovní spojení „dílčí zadání“, angl. „assignment part“, pro základní prvek testové úlohy, příkladem odstavec textu, obrázek nebo diagram. V portálu se můžeme setkat s 5 typy dílčích zadání

- text
- obrázek
- normalizace relačního schématu
- transformace konceptuálního modelu
- databázové připojení

Slovo „zadání“, angl. „assignment“, bude použito pro soubor dílčích zadání. Zadání má představovat problém, nad kterým student řeší otázky.

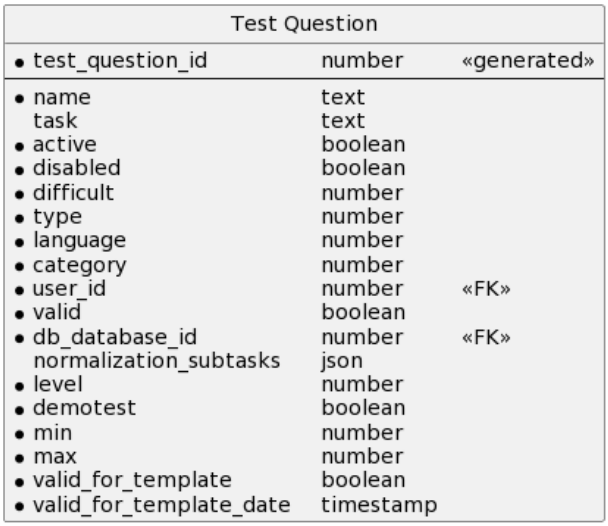


Obrázek 2.1: Struktura zadání

Na obrázku 2.1 vidíme strukturu dílčích zadání tvořenou ze dvou tabulek. Tabulka *test_assignment_to_question* slouží jako vazební tabulka mezi dílčím zadáním a otázkou. Můžeme si povšimnout atributu *type*, který představuje číselný ekvivalent výčtového typu dílčího zadání. Od hodnoty *type* se dále odvíjí obsah atributu *data* uvedené v následující tabulce.

Typ	Data
Text	String
Obrázek	Cesta k souboru
Normalizace relačního schématu	JSON
Transformace konceptuálního modelu	JSON
Databázové připojení	Celé číslo

Tabulka 2.1: Datový typ atributu v dílčím zadání



Obrázek 2.2: Struktura otázek

Na obrázku 2.2 vidíme strukturu otázky. Na první pohled obsahuje velké množství informací například atributy pro tvorbu testových šablon: *valid_for_template* a *valid_for_template_date*, hodnocení otázek: *min* a *max*.

Obdobně jako u dílčích zadání nalezneme zde atributy pro výčtové typy, kde je uložen jejich číselný ekvivalent

- *difficult* – obtížnost v rozmezí lehké až těžké,
- *type* – typ formuláře na odpověď, výčtem checkbox, radiolist, text, diagram, SQL, relační algebra, transformace a normalizace,
- *language* – čeština nebo angličtina,
- *category* – typ otázky, zda se jedná o otázku na SQL dotaz, dotaz v relační algebře, otázka nad modelem nebo teoretická otázka,
- *level* – vhodnost otázky pro zkoušku, test či demo test.

Otázky pro dílčí zadání typu normalizace relačního schématu dále obsahují atribut *normalization_subtasks*. Tento atribut je typu JSON a obsahuje pouze pole čísel 0 až 4. Jedná se o informaci, které úkoly jsou k normalizaci přiřazené.

2.1.3 Štítky

Pojem štítek v kontextu této práce nabývá významu nálepky nebo označení, které slouží k bližší identifikaci objektu nebo přidání informace k objektu. V širším kontextu může být štítek i vlastnost, schopnost nebo přívlastek –

cokoliv, co blíže popisuje daný objekt. Například „modrá“ barva u trička může být štítek. Lidé často vyhledávají a filtrují své preference k jídlu, filmům nebo jiným dennodenním záležitostem podle přívlastků. V programování se setkáme s dvěma charaktery štítku: předdefinované nebo přizpůsobitelné. Do předdefinovaných štítků patří výčtové typy, konstanty, hodnoty ve zdrojovém kódu. Přizpůsobitelné již vychází ze vstupu uživatele.

Funkce štítků, jako způsob shlukování podobných objektů do skupiny, není v současném systému implementované v obecné šíři. Nalezeneme pouze předdefinovaný typ štítků. Přínos štítků pro systém spočívá v možnosti filtrovat a vyhledávat podle nich. V limitovaném rozsahu najdeme ve strukturách zadání a otázek výše na obrázku 2.1 hodnoty výčtových typů v atributech k chování štítků například

- *obtížnost*,
- *typ otázky*,
- *jazyk*.

Výše zmíněné vlastnosti patří mezi prvky, podle kterých se momentálně dá filtrovat. Tuto možnost je potřeba reflektovat v novém systému a rozšířit na jakýkoli objekt.

V případě *obtížnosti* by také nemělo být povoleno, aby daný objekt mohl být, jak lehký, tak i těžký. Z toho vychází potřeba rozlišit skupiny štítků dle toho, zda jejich hodnoty jsou v disjunktním vztahu.

2.2 Požadavky

Tato sekce popíše některé požadavky vycházející z analýzy. Požadavky se dají rozdělit do funkčních a nefunkčních požadavků. Funkční požadavky vymezují hranicí mezi tím, co systém bude umět a co naopak nebude podporovat. Nefunkční požadavky popisují kvalitativní vlastnosti systému.

2.2.1 Funkční požadavky

2.2.1.1 FP1: Označení objektu štítkem

Uživatel bude schopný přiřadit vlastní štítek k objektům, které je budou podporovat, zejména testové otázky.

2.2.1.2 FP2: Vyhledávání objektů podle štítků

Uživatel bude schopný vyhledávat a filtrovat objekty podle výběru štítků.

2.2.1.3 FP3: Disjunktní štítky

Systém umožní uživateli definovat množinu štítků, ze kterých může uživatel přiřadit k objektu nejvýše jeden.

2.2.1.4 FP4: Zachování historie zadání a otázek

Systém nedovolí úplný výmaz zadání nebo otázek. Při úpravě musí být možné dohledat celou historii se všemi hodnotami.

2.2.2 Nefunkční požadavky

2.2.2.1 NFP1: Použití architektury mikroslužeb

Systém bude rozdělen do menších dílčích prostředí, každá spravující jinou doménu.

2.2.2.2 NFP2: Jazyk PHP

Systém bude implementován v jazyce PHP

2.2.2.3 NFP3: Rozhraní REST

Pro komunikaci mezi mikroslužbami a s frontendem bude použit vzor REST po protokolu HTTP.

2.2.2.4 NFP4: Výkon

Systém by měl zpracovat značnou část dotazů do 5 sekund a doba odezvy nesmí překročit 10 sekund. Uživatel by si neměl uvědomit, že čeká.

2.2.2.5 NFP5: Udržitelnost a rozšiřitelnost

Přidávání komponent nebo úprava nad systémem vyžaduje co nejmenší zásah na zbytek systému.

2.3 Případy užití

Případy užití zde budou popisovat možné scénáře, které vychází z požadavků na navrhované služby. Jelikož se práce zaměřuje na backendovou stranu systému, jsou interakce od uživatele jako „Uživatel si zvolí v menu tlačítko X“ nebo „Uživatel vyplní zobrazený formulář“ zkráceny a nahrazeny za „Uživatel si vyžádá X“ anebo „Uživatel pošle žádost o tvorbu/úpravu X“.

2.3.1 Aktéři

2.3.1.1 Student

Uživatel s omezenými právy. Jedná se o jedince, který studuje předmět na ČVUT předmět podporovaný portálem.

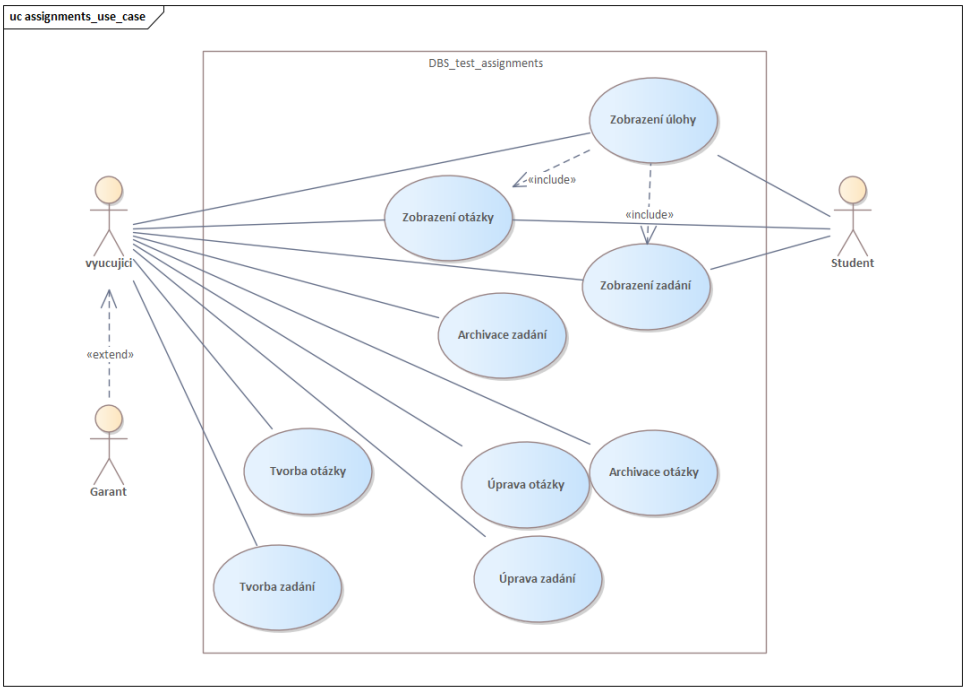
2.3.1.2 Vyučující

Vyučující se jedinec, který vzdělává studenty. Uživatel s právy k tvorbě materiálů na výuku, například zadání či otázky.

2.3.1.3 Garant

Uživatel s plnými právy, oproti vyučujícímu má přístup ke konfiguraci předmětu.

2.3.2 Testové zadání a otázky



Obrázek 2.3: Diagram případu užití pro zadání a otázky

2.3.2.1 UC1: Zobrazení úlohy

Umožňuje aktérovi si zobrazit detaily testové úlohy, příslušné zadání a její otázky.

- Aktéři: Student Vyučující, Garant

Scénář:

1. Uživatel si vyžádá testovou úlohu.
2. Systém nalezne otázku úlohy a připojí k ní její zadání.
3. Systém odpoví uživateli získaným objektem

2.3.2.2 UC2: Zobrazení zadání

Umožňuje aktérovi si zobrazit detaily zadání a části, ze kterých se skládá.

- Aktéři: Student Vyučující, Garant

Scénář:

1. Uživatel si vyžádá detaily testového zadání.
2. Systém nalezne zadání a její části.
3. Systém odpoví uživateli získaným zadáním.

2.3.2.3 UC3: Tvorba zadání

Umožňuje aktérovi vytvořit nové zadání.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o vytvoření zadání.
2. Systém validuje informace žádosti.
3. Systém vytvoří dle žádosti nové zadání.
4. Uživatel je informován o úspěšnosti žádosti.

2.3.2.4 UC4: Úprava zadání

Umožňuje aktérovi upravit informace existujícího zadání.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o úpravě zadání.
2. Systém validuje informace žádosti.
3. Systém nalezne příslušné zadání a archivuje jej.
4. Systém vytvoří dle žádosti nové zadání.
5. Uživatel je informován o úspěšnosti žádosti.

2.3.2.5 UC5: Archivace zadání

Umožňuje aktérovi vyřadit existující zadání ze seznamu použitelných zadání.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o archivaci zadání.
2. Systém nalezne příslušné zadání a archivuje jej.
3. Uživatel je informován o úspěšnosti žádosti.

2.3.2.6 UC6: Zobrazení otázky

Umožňuje aktérovi si zobrazit detaily otázky.

- Aktéři: Student Vyučující, Garant

Scénář:

1. Uživatel si vyžádá detaily otázky.
2. Systém nalezne otázku.
3. Systém odpoví uživateli získanou otázkou.

2.3.2.7 UC7: Tvorba otázky

Umožňuje aktérovi vytvořit novou otázku.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o vytvoření otázky.
2. Systém validuje informace žádosti.
3. Systém vytvoří dle žádosti novou otázku.
4. Uživatel je informován o úspěšnosti žádosti.

2.3.2.8 UC8: Úprava otázky

Umožňuje aktérovi upravit informace existující otázky.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o úpravě otázky.
2. Systém validuje informace žádosti.
3. Systém nalezne příslušnou otázku a archivuje jej.
4. Systém vytvoří dle žádosti novou otázku.
5. Uživatel je informován o úspěšnosti žádosti.

2.3.2.9 UC9: Archivace otázky

Umožňuje aktérovi vyřadit existující otázku.

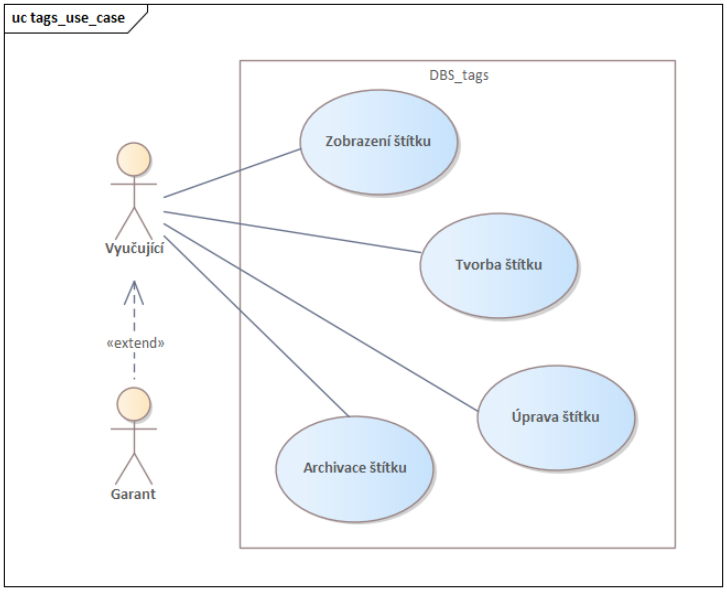
- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o archivaci otázky.
2. Systém nalezne příslušnou otázku a archivuje jej.
3. Uživatel je informován o úspěšnosti žádosti.

2.3.3 Štítky

Hodnota štítků, vlastnost nebo přívlastek, který představují, musí být schopný uživatel sám definovat. Do toho spadají tedy případy použití na správu.



Obrázek 2.4: Diagram případu užití pro štítky

2.3.3.1 UC1: Zobrazení štítku

Umožňuje aktérovi si zobrazit štítek.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel si vyžádá detaily štítku.
2. Systém nalezne štítek.
3. Systém odpoví uživateli získaným zadáním.

2.3.3.2 UC2: Tvorba štítku

Umožňuje aktérovi vytvořit štítek.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o vytvoření štítku.
2. Systém validuje informace žádosti.
3. Systém vytvoří dle žádosti nový štítek.
4. Uživatel je informován o úspěšnosti žádosti.

2.3.3.3 UC3: Úprava štítku

Umožňuje aktérovi upravit detaily štítku.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o úpravě štítku.
2. Systém validuje informace žádosti.
3. Systém nalezne příslušný štítek a upraví jej dle žádosti.
4. Uživatel je informován o úspěšnosti žádosti.

2.3.3.4 UC4: Archivace štítku

Umožňuje aktérovi deaktivovat štítek.

- Aktéři: Vyučující, Garant

Scénář:

1. Uživatel pošle žádost o archivaci štítku.
2. Systém nalezne příslušnou štítek a archivuje jej.
3. Uživatel je informován o úspěšnosti žádosti.

Kapitola 3

Návrh

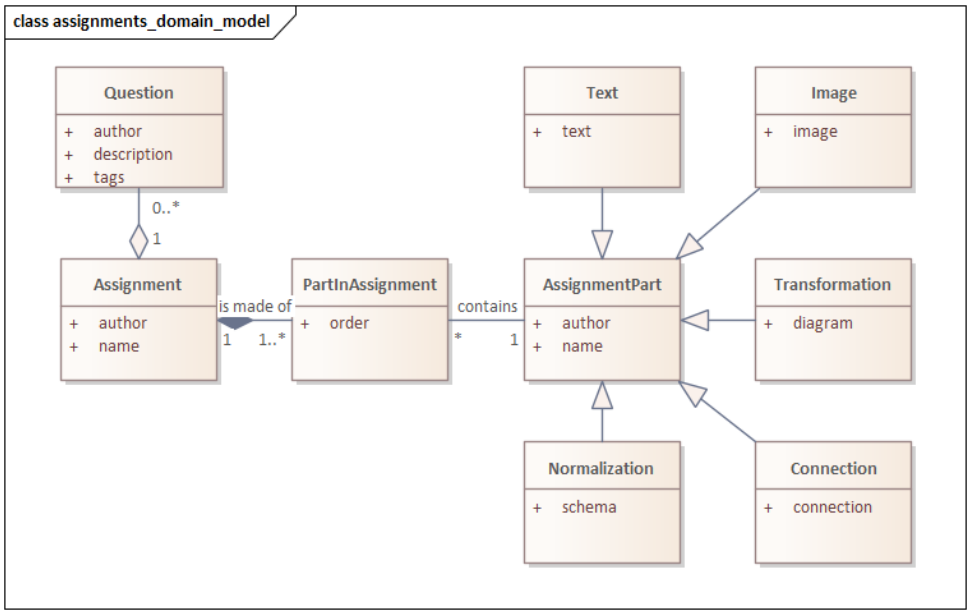
V této kapitole bude popsána architektura, která bude použita na backend aplikace. Také se kapitola bude věnovat návrhům mikroslužeb pro testové zadání a štítky z pohledu doménového modelu, databázového modelu a RESTového rozhraní.

3.1 Doménový model

Doménový model slouží k popsání a modelování entit a jejich vzájemných vztahů. Skládá se ze zjednodušených tříd, ve kterých chybí datové typy atributů a metody. Entity obsahují pouze nutné entity.

3.1.1 Testové zadání

Pro mikroslužbu na správu zadání bude použit název Assignments. Mezi klíčové entity, které je třeba zachovat, patří zadání a otázka. Zadání a otázky jsou v relaci M:N a je možné vytvářet otázky, které se nevztahují ke specifickému zadání. Jedním nedostatkem nynějšího řešení je znovupoužití již sestaveného zadání, například dvou textů, jednoho obrázku a jednoho diagramu, ve více otázkách. Cílem je vytvořit entitu, která by měla roli prostředníka pro dílčí zadání a otázky.



Obrázek 3.1: Doménový model DBS-Assignments

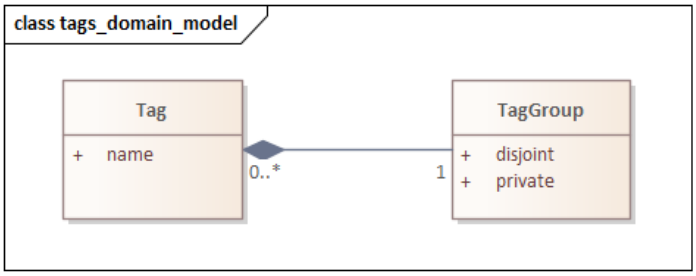
Model 3.1 zakomponovává obě entity, *Assignment* a *Question*, ze současné implementace na obrázcích 2.1 a 2.2, a přidává entitu *Assignment Part*.

Assignment Part nyní reflektuje původní chování objektu *Test Assignment* z 2.1 a obsahuje informaci jako text, obrázek, diagram a další. Nově entita *Assignment* má sloužit jako uspořádaný seznam „dílčích zadání“ – *Assignment Part*. O uspořádání se stará entita *PartInAssignment* s pořadníkem.

Z otázek byly odebrány atributy týkající se hodnocení testů a tvorbě testových šablon, jelikož jsou spravovány jinými mikroslužbami. Atributy připomínající výčtový typ jako obtížnost nebo jazyk jsou nahrazeny za „tags“. Uživatel si bude moci vytvořit vlastní štítky a bez omezení na výčet přiřadit k otázce.

3.1.2 Štítky

Pro mikroslužbu na správu štítků bude použit název Tags. Dostačující informace ke štítkům je název a případně popis. Přirovnáme-li výčtový typ *obtížnost* s hodnotami *lehké*, *průměrné* a *těžké*, štítky představují hodnotu výčtu a je potřeba entita na samotný výčtový typ.



Obrázek 3.2: Doménový model DBS-Tags

Model 3.2 obsahuje dvě entity: „Tag“ a „Tag Group“. Entita Tag Group hraje roli výčtového typu a zaobaluje štítky do skupin. Definuje atributy

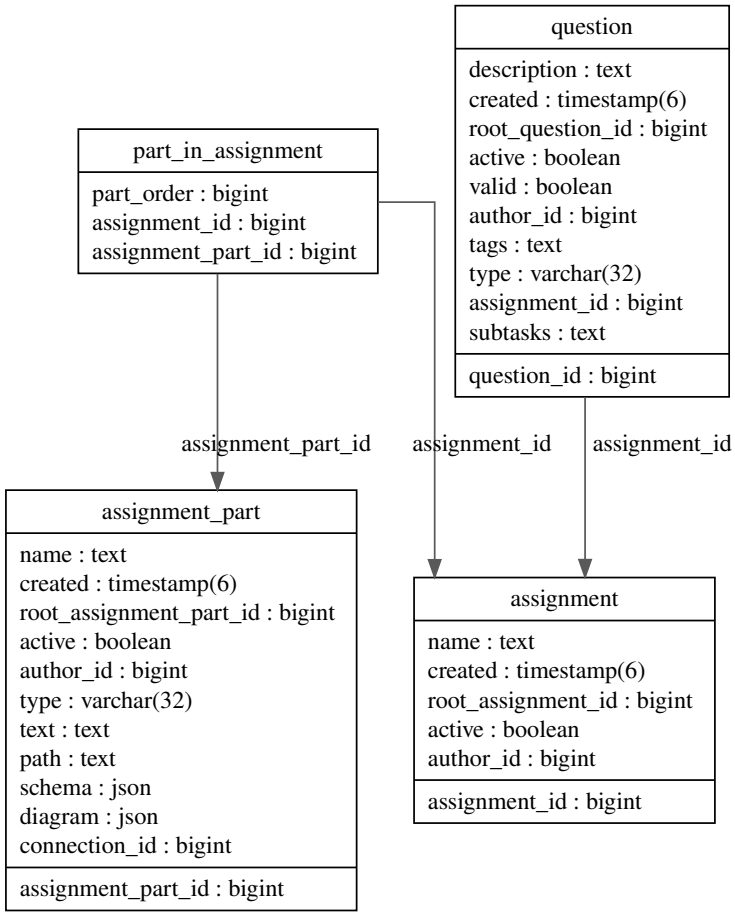
- *disjoint* – informace o disjunktnosti skupiny,
- *private* – úroveň práv přístupu ke skupině.

3.2 Databáze

Tato sekce popíše databázové diagramy vycházející z doménových modelů. Obě mikroslužby vyžadují nějakou formu zamezení smazání objektů. Ostatní mikroslužby zpravidla nedostanou žádnou informaci o změně, proto tyto databáze implementují nějaký z typů Slowly Changing Dimension.

3.2.1 Testové zadání

Převést entity Question, Assignment a PartInAssignment do tabulek nevyžaduje žádnou větší úpravu. Pro zachycení generalizaci entity *Assignment Part* z 3.1 v relační databázi je vybrána strategie Single Inheritance Table, kde všechny atributy všech synovských entit tvoří jednu tabulku. Knihovna Doctrine, která bude zprostředkovávat komunikaci s databází nabízí ještě Class Inheritance Table – každá entita má vlastní tabulku. Class Inheritance Table má negativní dopad na výkon, protože vyžaduje spojování tabulek při dotazování, a proto není použitý.



Obrázek 3.3: Databázový model DBS-Assignments

Tabulka *assignment_part* na obrázku 3.3 obsahuje tyto atributy synovských entit:

Typ	Atribut
Text	text
Image	path
Normalization	schema
Transformation	diagram
Connection	connection_id

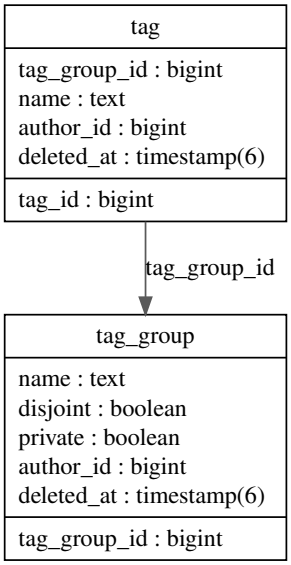
Tabulka 3.1: Název atributu k dílčímu zadání

Zadání, dílčí zadání i otázky jsou objekty, u kterých je důležité, aby si je uživatel mohl zpětně zobrazit. Proto jsou do tabulek *assignment*, *assignment_part* a *question* přidány podle typu 2 Slowly Changing Dimension atributy

- *created* – časová značka o vzniku záznamu,
- *active* – zda entita je historická nebo aktuální,
- *root_X_id* – entita, ze které je odvozena a je na začátku historie.

3.2.2 Štítky

Štítky v sobě oproti testovému zadání neobsahují informace, které je nutné historizovat. Přesto obě entity dostávají atribut s časovou značkou o času výmazu. Tento atribut slouží pouze k rozlišení archivovaných od aktivních záznamů.



Obrázek 3.4: Databázový model DBS-Tags

Zde je použit typ 1 Slowly Changing Dimension, úplný přepis a zahození informací předchozího záznamu.

3.3 Vyhledávání podle štítků

Jeden z požadavků pro štítky je najít objekty, které jsou například lehké, typu SQL a případně určené pro zkoušku. Vyhledané objekty jako množinu můžeme popsat pomocí štítků a hledáme způsob, jak ostatní mikroslužby mohou přes

rozhraní získat celou množinu aniž by předem znaly, které objekty do množiny patří nebo jak velká množina je. Nabízí se ukládat informaci ve formě řetězce s identifikátory štítků a logickými operátory. Tento příklad štítků můžeme přepsat do výrazu „lehké & SQL & zkouška“ a dále zaměníme za jejich identifikátory, například „1 & 6 & 10“.

3.3.1 Gramatika

Mezi požadované operátory patří „AND“ pro konjunkci a „OR“ pro disjunkci, kde AND má přednost. Pro tyto operátory budou použity znaky **&** a **|**. Gramatika také musí umět zpracovat kulaté závorky. Validní LL(1) gramatika by obsahovala:
Neterminální symboly:

- START – startovací symbol
- OR_EXPR
- OR_EXPAND
- AND_EXPR
- AND_EXPAND
- TERM

Terminální symboly:

- &
- |
- (
-)
- id – číselný identifikátor, celé číslo

Přepisovací pravidla:

1. START → OR_EXPR
2. OR_EXPR → AND_EXPR OR_EXPAND
3. OR_EXPAND → | AND_EXPR OR_EXPAND
4. OR_EXPAND → ε
5. AND_EXPR → TERM AND_EXPAND
6. AND_EXPAND → & TERM AND_EXPAND

7. AND_EXPAND → ε
8. TERM → (OR_EXPR)
9. TERM → id

Můžeme si povšimnout zbytečného přepisovacího pravidla START → OR_EXPR. Toto pravidlo slouží pouze k rozlišení startovacího symbolu pro přehlednost ve zdrojovém kódu. Může být odebráno a při nastavení startovacího symbolu na OR_EXPR jsou jazyky gramatik stejné.

Pro parsování této gramatiky je použita metoda analýzy pomocí parsovací tabulky. Motivace k této volbě je možná evaluace bez zpětného průchodu, která se vyskytuje v metodě rekurzivního sestupu. Také úprava gramatiky nemusí nutně vyžadovat přepsání funkcí a procedur analyzátoru, pouze hodnot parsovací tabulky.

3.3.1.1 Parsovací tabulka

Nejprve se musí zkonstruovat množiny $FIRST$ a $FOLLOW$ pro každý neterminální symbol.

FIRST:

- START = { (, id}
- OR_EXPR = { (, id}
- OR_EXPAND = { | , ε}
- AND_EXPR = { (, id}
- AND_EXPAND = { & , ε}
- TERM = { (, id}

FOLLOW:

- START = { $[1] ,) }
- OR_EXPR = { $,) }
- OR_EXPAND = { $,) }
- AND_EXPR = { | , $,) }
- AND_EXPAND = { | , $,) }
- TERM = { & , | , $,) }

[1] znak $ značí konec řetězce

Nyní pomocí vytvořených množin můžeme přejít na parsovací tabulku. Nechť řádky jsou označeny neterminálními symboly a sloupce terminálními symboly. Konstruování tabulky postupuje podle algoritmu:

Fragment kódu 2 Pseudo-algoritmus pro vytvoření parsovací tabulky

```
// Cell [N][t] contains rule for non-terminal symbol N and terminal symbol t

for each rule "A → B"
    for each symbol "x" in set FIRST(A)
        Add to cell [A][x] rule "A → B"
    if FIRST(A) contains "ε"
        for each symbol "y" in set FOLLOW(A)
            Add to cell [A][y] rule "A → ε"
    if FIRST(A) contains "ε" and FOLLOW(A) contains "$"
        Add to cell [A][$] rule "A → ε"
```

[22]

Aplikováním tohoto algoritmu zkonstruujeme tabulku. Buňky obsahují identifikátor přepisovacího pravidla z výčtu.

Parse Table	&	|	(	)	id	$
START			1		1	
OR_EXPR			2		2	
OR_EXPAND		3		4		4
AND_EXPR			5		5	
AND_EXPAND	6	7		7		7
TERM			8		9	

Tabulka 3.2: Vytvořená parsovací tabulka

3.3.2 Metoda syntaktické analýzy

Algoritmus analýzy pomocí parsovací tabulky zmíněné v sekci 1.2.1.6 je schopná pouze rozeznat, zda vstup patří do jazyka nebo nepatří. Aby bylo možné vyhledávat objekt podle vstupního výraz je potřeba vstup zpracovat a vyhodnotit. Jednou z možností, jak toho dosáhnout, je vytvořit derivační strom a převést derivační strom do abstraktního syntaktického stromu. Po každém dotazu na parsovací tabulku si musíme zaznamenat, jaký vrchol nebo symbol se momentálně zpracovává, a které symboly jsou produkcí derivace.

Hlavní změny spočívají v přidání zásobníku, který v sobě ukládá vrcholy stromu. Během průchodu řetězcem se v každé iteraci vytvoří nový vrchol a do zásobníku se vloží reference vrcholu tolikrát, kolik symbolů se nachází v produkci derivace. Výstupem této procedury je kořen derivačního stromu.[23]

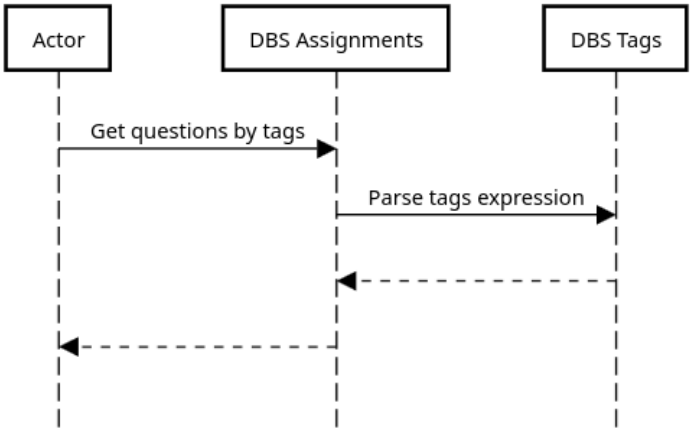
Fragment kódu 3 Pseudo-algoritmus pro vyhodnocení výrazu LL(1) gramatiky

```
Initialize symbol stack
Add Start symbol to symbol stack

Initialize node stack
Create Root node
Add Root node to node stack

token := Read token from input

while Symbol stack is not empty
    symbol := Pop from symbol stack
    parentNode := Pop from node stack
    currentNode := Create node for current symbol
    Add currentNode as child to parentNode

    if symbol == token
        token := Read next symbol

    else if symbol is non-terminal
        rhs := Get right hand side symbols of production
        Push right hand side symbols to symbol stack

        for each symbol in rhs
            Push reference of currentNode to node stack
    else
        Syntax error

Return Root node
```

[23]

3.3.3 Umístění analyzátoru

S přechodem do architektury mikroslužeb jsou štítky kompletně oddělené od objektů, které mají reprezentovat. S tím přichází otázka, kde se bude analýza provádět. Vezmeme příklad s otázkami, které se budou nacházet v mikroslužbě spravující zadání, „Assignments“. Uživatel pošle žádost o získání otázek do Assignments. Uvážíme-li analýzu rovnou v dané mikroslužbě, byla by potřeba přidat ten samý analyzátor do dalších mikroslužeb, které tuto funkcionalitu vyžadují. V tu chvíli duplikujeme kód a zvyšujeme obtížnost provádění úprav. Alternativou je provádět analýzu v mikroslužbě na štítky – Tags, což vyžaduje přeposlání dotazu z Assignments do Tags a poté použít odpověď na vytvoření dotazu v Assignments.



Obrázek 3.5: Sekvenční diagram dotazu na analýzu

Ve výše uvedeném algoritmu je výstupem procedury kořen derivačního stromu. Za předpokladu, že jej rovnou zpracujeme, vrací mikroslužba Tags abstraktní syntaktický strom. Aby mohla Assignments zpracovat odpověď, musí umět rozpoznat, že se jedná o AST. Bohužel se nedá vyhnout tvorbě společných objektů, které je nutné sdílet napříc mikroslužbami. Pro takové objekty můžeme vytvořit knihovnu a přidat jako závislost do cílových mikroslužeb.

Validním řešení je také přidat kompletní analyzátor do knihovny. Nevýhodou může být nárůst velikosti závislostí. Dále máme-li analyzátor v Tags, je možné provádět validaci existence daný štítků.

Kapitola 4

Realizace

V této kapitole bude popsána implementace mikroslužeb Assignments a Tags. Seznámíme se s technologiemi a prostředky, kterými vývoj bude probíhat. Podstatné části implementace zde budou zdůrazněny.

4.1 Prostředí

4.1.1 Technologie

Pro realizaci byl zachován programovací jazyk PHP s rozdílem ve verzi. Dále je také zachován databázový systém PostgreSQL, kde každá mikroslužba bude mít vlastní instanci. Namísto frameworku Nette byl zvolen Symfony. Celý systém v minulosti běžel v izolovaném virtuálním prostředí pomocí Vagrantu. S rostoucí popularitou Dockeru a přechodem na architekturu mikroslužeb se nabízí možnost použít Docker Compose na orchestraci izolovaných služeb v tzv. kontejnerech.

4.1.1.1 PHP

Verze 7.0 byla vydána v roce 2015 a oficiální konec podpory poslední iterace, verze 7.4, přišel v listopadu roku 2022. Absence nových aktualizací, ať již ve formě nových funkcionalit, nebo bezpečnostních záplat, bývá podnětem přechodu na novější verze či alternativy. První vydání PHP 8 byl v roce 2020 a pravidelně každý rok vychází nová iterace. Použitá bude verze 8.1 a mezi přidané změny patří výčtové typy, atributy, union typy, null safe operátor nebo match výraz. [24]

4.1.1.2 Symfony

Podobně jako Nette je Symfony framework napsaný v jazyce PHP, nabízí množinu oddělených komponent a také je open-source. Je zaměřené na rychlost a přepoužitelnost. Drží se standardům PHP a nabízí snadnou rozšiřitelnost. [25]

4.1.1.3 Git

Git je open-source verzovací systém. Mimo ukládání zdrojových souborů a jejich verzování, bude sloužit Git ke code-review mezi kolaboranty. V úvodu bylo zmíněno, že vývoj probíhá souběžně s vývojem nad správou testů, testových šablon a hodnocení testů, a uživatelským rozhraním. Správně použitý Git zaručí snadné sdílení souborů.

4.1.1.4 Docker

Docker je nástroj na vývoj a spouštění aplikací v kontejnerech. Oproti Vagrantu nevyžaduje nastavení virtualizačního nástroje a umožňuje zabalit aplikaci do snadno reprodukovatelných balíčků.

4.2 Implementace

Samotná implementace probíhala použitím nástrojů PHPStorm na zdrojové kódy PHP a Vistual Studio Code na ostatní textové soubory.

4.2.1 Rozhraní REST

Služby vyžadují rozhraní, přes které závislé mikroslužby a frontend mohou posílat dotazy. Pro volbu metody na danou žádost se držíme standardních konvencí. Na dotazy z případů užití

- zobrazení je použita REST metoda GET,
- vytvoření metoda POST,
- úpravu objektů metoda PATCH,
- archivaci metoda DELETE,
- použití parseru metoda POST.

4.2.1.1 Koncové body rozhraní

Jednotlivé mikroslužby mají prefix signalizující, o jakou mikroslužbu se jedná. Pro mikroslužbu „Tags“ je použit prefix „/tags“ a jsou poskytnuty tyto koncové body (bez prefixu):

- **GET /tag-groups** – Získání všech skupin štítků
- **POST /tag-groups** – Vytvoření skupiny štítků
- **GET /tag-groups/{tagGroupId}** – Získání detailů jedné skupiny štítků
- **PATCH /tag-groups/{tagGroupId}** – Úprava detailů skupiny štítků
- **DELETE /tag-groups/{tagGroupId}** – Archivace skupiny štítků
- **GET /tag-groups/{tagGroupId}/tags** – Získání všech štítků ve skupině
- **POST /tag-groups/{tagGroupId}/tags** – Vytvoření štítku
- **GET /tags/{tagId}** – Získání detailů jednoho štítku
- **PATCH /tags/{tagId}** – Úprava detailů štítku
- **DELETE /tags/{tagId}** – Archivace štítku
- **POST /tags/validity** – Validace množiny štítků na existenci a platnost disjunktních množin
- **POST /tags/expression/validity** – Validace a zpracování výrazu se štítky do abstraktního syntaktického stromu

Mikroslužba „Assignments“ má prefix „test-assignments“ a definice jejích koncových bodu (bez prefixu) vypadá takto:

- **GET /assignments** – Získání všech zadání
- **POST /assignments** – Vytvoření zadání
- **PATCH /assignments/{assignmentId}** – Úprava detailu zadání
- **DELETE /assignments/{assignmentId}** – Archivace zadání
- **GET /assignments-parts** – Získání všech dílčích zadání
- **POST /assignments-parts** – Vytvoření dílčího zadání
- **PATCH /assignments-parts/{assignmentPartId}** – Úprava detailu dílčího zadání
- **DELETE /assignments-parts/{assignmentPartId}** – Archivace dílčího zadání

- **GET /assignments/{assignmentId}/questions** – Získání všech otázek k zadání
- **POST /assignments/{assignmentId}/questions** – Vytvoření otázky k zadání
- **PATCH /questions/{questionId}** – Úprava detailu otázky
- **DELETE /questions/{questionId}** – Archivace otázky
- **POST /assignments/{assignmentId}/questions/by-tags** – Získání množinu otázek podle výběru štítků ve formě výrazu

4.2.1.2 Zdrojový kód

Třídy, které se starají o definici rozhraní a zpracování zpráv na daném rozhraní se nazývají řadiče, angl. controller. Symfony nabízí jednoduchou kostru ve formě abstraktní třídy AbstractFOSRestController na přidávání jednotlivých REST metod. Zde jsou ukázány pouze nespecifické zdrojové kódy, protože tato forma je použitá vícekrát v implementovaných mikroslužbách s minimálními rozdíly.

Fragment kódu 4 Vytvoření řadiče

```
class MyController extends AbstractFOSRestController
{
    public function __construct(
        private readonly MyResourceService $resourceService
    )
    {
    }
}
```

Od PHP 8.0 je možné přidat třídní parametry rovnou do konstruktoru třídy. Při konfiguraci Symfony je možné nastavit automatické vkládání závislostí. Které závislosti jsou předávány vychází z definice konstruktoru třídy. V příkladu výše je třída MyController závislá na třídě MyResourceService a v době instancování ji obdrží.

Fragment kódu 5 Přidání metod GET a DELETE rozhraní

```
// MyController

#[Get(
    path: /uri/path/to/resource,
    name: "getObject",
)]
public function getObject(): View
{
    /**
    * Handle resource fetching
    */
    return View::create(/* requested object */, Response::HTTP_OK);
}

#[Delete(
    path: /uri/path/to/resource/{objectId},
    name: "archiveObject",
)]
public function archiveObject(int $objectId): View
{
    if (!$this->resourceService->exists(objectId)) {
        return View::create($objectId, Response::HTTP_NOT_FOUND);
    }
    /**
    * Handle resource archiving
    */
    return View::create(null, Response::HTTP_NO_CONTENT);
}
```

Příklad výše ukazuje definování rozhraní pomocí PHP atributů. Obdobně jako Get, jsou k dispozici i atributy pro zbylé REST metody. Má-li rozhraní v URI cestě proměnnou například *objectId*, je do funkce automaticky také vložena.

Metody mohou požadovat ve formě formulářů a pro takové cesty rozhraní se musí definovat objekty pro přenos dat, Data Transfer Object (DTO), a třídu na serializaci daného objektu. Nejčastěji se s formuláři setkáme s metodami na tvorbu a úpravu.

Fragment kódu 6 Objekt pro přenos dat a serializer

```
// Data Transfer Object
public class CreateObjectRequest
{
    public string $name;

    public int $value;
}

// Serializer
public class CreateObjectType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder,
                              array $options): void
    {
        $builder->add("name", TextType::class)
            ->add("value", NumberType::class;
    }

    protected function getEntityType(): string
    {
        return CreateObjectRequest::class;
    }
}
```

DTO obsahují atributy dotazu jako třídní atributy. Pro serializaci objektu můžeme použít abstraktní třídu AbstractType. Funkce serializeru jsou zděděné z rodičovské třídy.

Fragment kódu 7 Metoda POST s tělem dotazu

```
// MyController

#[Post(
    path: /uri/path/to/resource
    name: "createObject",
)]
public function createObject(Request $request): View
{
    $formRequest = new CreateObjectRequest();
    $form = $this->createForm(CreateObjectType::class, $formRequest);
    $form->submit($request->request->all());

    /**
    * Check for form errors
    * Handle object creation
    */
    return View::create(/* response body */, Response::HTTP_CREATED);
}
```

Funkce createForm() v MyController je definována v rodičovské třídě. Voláním funkce submit() nám serializační třída zpracuje vstup do DTO. Ten dále můžeme použít pro vlastní logiku a máme přístup k jednotlivým parametrům dotazu. Tyto ukázky pokrývají značnou část požadovaného rozhraní a nevyžaduje závratné změny.

4.2.2 Parser

Než budeme moci číst vstup a analyzovat podle gramatiky, je potřeba provést lexikální analýzu nad vstupním řetězcem a získat sekvenci tokenů. Až tehdy můžeme provést syntaktickou analýzu.

Lexikální analýza spočívá ve čtení vstupního řetězce charakter po charakteru a porovnávat je s očekávanou množinou. Mezi přípustné charaktery patří:

- operátory & a |,
- závorky (a),
- číslice,
- mezery, protože se dají vypustit.

Z řetězce „1 & (22 | 3)“ se stane sekvence [1, &, (, 22, |, 3,)].

4.2.2.1 Syntaktická analýza

Nyní se zaměříme na syntaktickou analýzu. Terminální symboly a neterminální symboly jsou definovány pomocí výčtového typu, aby se ve zdrojovém kódu nevyskytovaly náhodné konstanty.

```
enum NonTerminalSymbol: string
{
    case NS_START = "ns_start";
    case NS_OR_EXPRESSION = "ns_or_expression";
    case NS_OR_EXPANSION = "ns_or_expansion";
    case NS_AND_EXPRESSION = "ns_and_expression";
    case NS_AND_EXPANSION = "ns_and_expansion";
    case NS_TERM = "ns_term";
}
```

Pro zvolenou metodu parsování 3.3.2 je potřeba parsovací tabulka. Jelikož se jedná o tabulku prvním instinktem je použít 2D pole nebo 2D asociační mapu. Pole v PHP je realizováno pomocí seřazené mapy, kde klíče mapy jsou omezené pouze na celá čísla a řetězce. Výčtové typy jsou novinka v PHP 8.0 a chovají se jako objekty, tudíž je nelze použít jako klíče. Čtenář může namítnout, proč nepoužít hodnotu výčtového typu, který je specifikován na řetězec. Určitě se tím dá tento problém vyřešit, ale tento způsob vyžaduje konverzi řetězce do výčtového typu před každým dotazováním do mapy.

Fragment kódu 8 Parsovací tabulka pomocí výrazu match

```
// ParseTable.php

public static function resolve(NonTerminalSymbol $nonTerminal,
                               TerminalSymbol $terminal): array
{
    return match ($nonTerminal) {
        NonTerminalSymbol::NS_START => match ($terminal) {
            TerminalSymbol::TS_LPARENS,
            TerminalSymbol::TS_NUMBER => [
                NonTerminalSymbol::NS_OR_EXPRESSION,
            ],
            default => throw new InvalidSymbolException(),
        },
        ... /** other cases **/
}
```

Novinkou v PHP 8.0 je konstrukce *match*. Je podobný konstrukci *switch*, kde se provádí rozvětvování podle hodnoty proměnné. Oproti switch porovnává match i typ. Vložením dvou konstrukcí match do sebe můžeme napodobit chování 2D mapy.

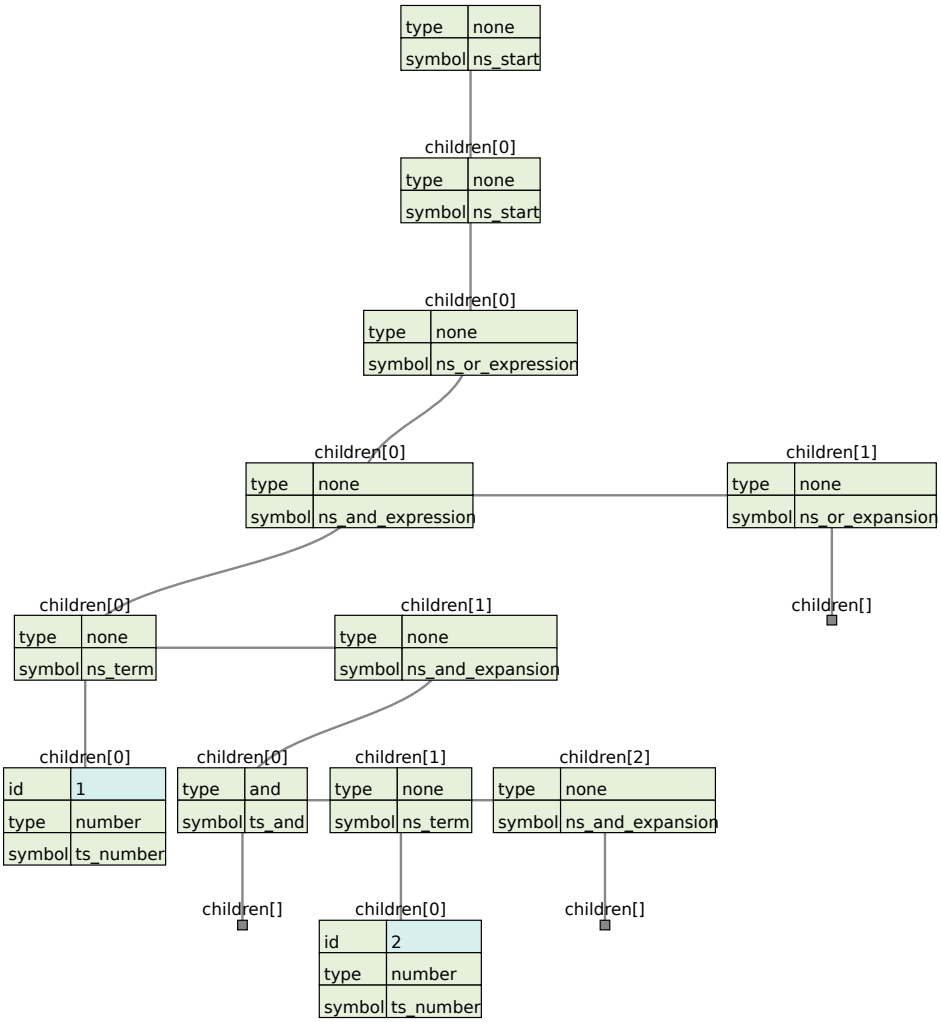
Implementace syntaktického analyzátoru se drží algoritmu uvedeného v 3.3.2. Jedna zvláštnost, která si zaslouží zmínit je volba zásobníku. Dispozici jsou standardní pole, třída SplStack z PHP knihovny na kolekce. V případě pole zvolíme-li vrchol zásobníku na začátku pole, potom jsou pro přidání a odebrání použity funkce *array_unshift()* a *array_shift()*. Kvůli implementaci pole jako mapy dochází v obou případech k lineárnímu průchodu polem na přeznačení klíčů. Pro náš případ to je nadbytečné. Co třeba vrchol zásobníku na konci? Funkce na přidání a odebrání v tom případě nemusí měnit klíče. Rozdíl ve výkonu mezi SPLStack a polem s vrcholem na konci je minimální. Pro implementaci byl vybráno pole s vrcholem na konci. [26]

Fragment kódu 9 Funkce pro zásobník pomocí PHP pole

```
// Array, top of stack at end start
$arr1 = [];
array_unshift($arr1, /* element */); //Add element
$element = array_shift($arr1);       //Remove element

// Array, top of stack at the end
$arr2 = [];
$arr2[] = /* element */        //Add element
$element = array_pop($arr1);   //Remove element
```

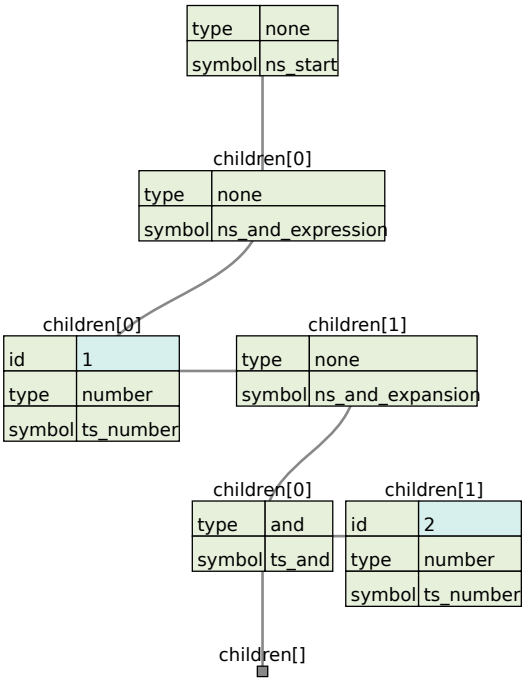
4.2.2.2 Úprava parsovacího stromu

Pro lepší čitelnost vezmeme jednodušší výraz „1 & 2“. Výsledkem syntaktické analýzy dostaneme tento derivační strom, vizualizace vygenerována s použitím JSON a nástroje JSON to Diagram Converter [3].



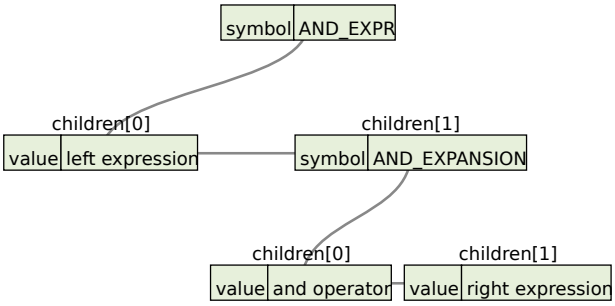
Obrázek 4.1: Derivační strom po analýze [3]

Povšimneme si, že strom obsahuje několik bezvýznamných listů a dlouhé lineární cesty. Bezvýznamné cesty vznikají z ε-pravidel a dlouhé lineární cesty z jednoduchých pravidel, jeden symbol na jeden symbol. Tyto konstrukty nám zvětšují strom bez přídatné informace a odstraněním se nic neztratí.



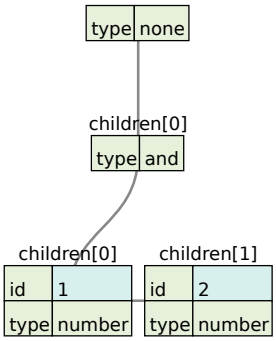
Obrázek 4.2: Ořezaný derivační strom [3]

Ořezaný strom vypadá mnohem lépe, ale ještě se neblíží požadovanému abstraktnímu syntaktickému stromu. Pro nalezení operátorů využijeme znovu přepisovací pravidla gramatiky. Způsob jak vznikne podstrom je přesně napsán podle pravidel. Hledáme podstrom vygenerovaný z pravidel a ten poté nahradíme za potřebnou reprezentaci v AST.



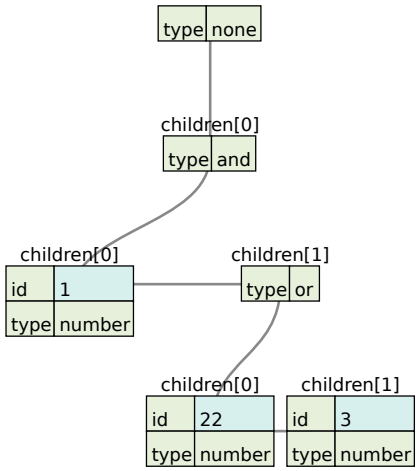
Obrázek 4.3: Podstrom operátoru AND [3]

To samé aplikujeme na operátor OR a upravíme již ořezaný derivační strom. Výsledkem je tato ukázka.



Obrázek 4.4: AST pro výraz „1 & 2“ [3]

Pro příklad „1 & (22 | 3)“ dostáváme tento strom.



Obrázek 4.5: AST pro výraz „1 & (22 | 3)“ [3]

Teď již máme abstraktní syntaktický strom a rekurzivním průchodem stromu jsme již schopni tvořit dotazy do ORM. Pro vrcholy, kde jsou identifikátory čísel vrací vrchol dotaz v duchu „WHERE tagId IN e.tags“, kde „e“ je dotazovaná entita a „tagId“ je identifikátor uložený ve vrcholu. Z vrcholů s operátory AND a OR dostáváme dotazy „WHERE left AND right“ a „WHERE left OR right“.

Způsob jaký dotaz je získán z vrcholu s identifikátorem závisí na způsobu uložení pole čísel v databázi. Doctrine ORM nabízí: *ARRAY*, *SIMPLE_ARRAY* a *JSON_ARRAY*. S typem *ARRAY* dochází k serializaci objektu do řetězce. *SIMPLE_ARRAY* předpokládá, že se jedná o primitivní typ, který se dá uložit pomocí řetězce. *JSON_ARRAY* zakóduje objekt do JSON objektu a uloží jej do databáze. Častokrát je JSON v databázi také ve formě řetězce. Ač se zdálo, že bylo na výběr, všechno je řetězec. V tu

chvíli musí daný vrchol vracet ekvivalent vyhledávání v poli pro řetězce. V případě otázek v mikroslužbě Assignments je použit *SIMPLE_ARRAY* na uložení štítků. Seznam „[1, 5, 6]“ je uložen v databázi ve formě „1,5,6“. Na vyhledávání v řetězci zbývá pouze operátor „LIKE“.

Fragment kódu 10 Vygenerovaný ORM dotaz pro identifikátor štítku

```
public function buildExpr(QueryBuilder $qb,
                          string $alias,
                          string $tagsField = "tags"): Orx
{
    $field = sprintf("%s.%s", $alias, $tagsField);
    $or = $qb->expr()->orX();
    $or->addMultiple([
        $qb->expr()->like($field, ":tag_middle_".$this->id),
        $qb->expr()->like($field, ":tag_start_".$this->id),
        $qb->expr()->like($field, ":tag_end_".$this->id),
        $qb->expr()->like($field, ":tag_single_".$this->id),
    ]);
    $qb->setParameter("tag_middle_".$this->id, "%,".$this->id.",%");
    $qb->setParameter("tag_start_".$this->id, $this->id.",%");
    $qb->setParameter("tag_end_".$this->id, "%,".$this->id);
    $qb->setParameter("tag_single_".$this->id, $this->id);
    return $or;
}
```

Třída *QueryBuilder* z knihovny Doctrine slouží ke konstrukci ORM dotazů. Výsledkem této funkce je dotaz ekvivalentní dotazu SQL:

Fragment kódu 11 SQL výraz ekvivalentní ORM dotazu

```
WHERE (
    e.tags LIKE '%1%'
    OR
    e.tags LIKE '1%'
    OR
    e.tags LIKE '%1'
    OR
    e.tags LIKE '1'
)
```

Toto řešení určitě není nejefektivnější, vytváří hodně vnořených podmínek v SQL dotazu. Přesto se nenabízí žádná lepší a snadnější alternativa.

4.3 Shrnutí

V této kapitole byla obecně popsána implementace REST rozhraní pro obě mikroslužby ve frameworku Symfony. Dále byly rozebrány kroky potřebné k syntaktické analýze pomocí parsovací tabulky. Výsledkem je parser skládající se ze tří hlavních komponent: lexikální analyzátor, syntaktický analyzátor a transformátor stromu. Parser je umístěn v mikroslužbě „Tags“ a je schopný vracet abstraktní syntaktický strom, který je možné konvertovat do výrazu do ORM nebo SQL.

Kapitola 5

Testování

Tato kapitola projde způsob a možnosti testování nad provedenou implementací. Cílem testování je hledání chyb, určení očekávaného chování. Správně napsané testy mohou zvýšit celkovou úroveň kvality vyvinutého produktu.

5.1 Nástroje

5.1.1 PHPUnit

PHPUnit je testovací nástroj pro jazyk PHP. Umožňuje testovat třídy, jednotlivé funkce a metody. Pomocí rozšíření pro spojení Symfony a PHPUnit můžeme také testovat chování tříd se závislostmi anebo imitovat HTTP volání.

5.2 Způsoby testování

5.2.1 Manuální testování

Manuální testy jsou prováděny uživatelem. Uživatel testuje na očekávané situace a hledá chyby a nedostatky. Tento způsob testování může zahrnovat průchod uživatelským rozhraním nebo posíláním vlastních dotazů na implementační rozhraní.

5.2.2 Automatické testování

Automatické testy jsou procedury, které jsou předem definovány při vývoji a jsou prováděny samy. Pro testování více hodnot pod jedním scénářem nabízí PHPUnit PHPDoc anotaci „@dataProvider [název funkce]“, jejíž parametrem je statická funkce, ve které se dají definovat testovací data.

5.2.2.1 Jednotkové testy

Jednotkové testy se zaměřují na testování individuálních tříd, funkcí nebo komponent. Jednotlivé testy by měly být od sebe nezávislé a pokrývat jinou situaci. V rámci projektu se hodí jednotkové testy na třídy syntaktického analyzátoru. PHPUnit nabízí třídu TestCase pro tvorbu jednotkových testů. Samotné testy jsou tvořeny z funkcí, jejichž název začíná na „test“.

Fragment kódu 12 Příklad jednotkového testu lexikálního analyzátoru

```
class LexerTest extends TestCase
{
    /**
     * @dataProvider provideValid
     */
    public function testLexerValid(string $expression,
                                   array $expectedTokens): void
    {
        $lexer = new Lexer($expression);
        $tokens = $lexer->tokenize();

        self::assertEquals($expectedTokens, $tokens);
    }

    public static function provideValid(): array
    {
        return [
            "simple and" => [
                " 1&2",
                [new NumberToken(1), new AndToken(), new NumberToken(2)],
            ],
            "simple or" => [
                "1 | 4  ",
                [new NumberToken(1), new OrToken(), new NumberToken(4)],
            ],
            "parenthesis" => [
                "2 & 30 | ( 2 | 26 & 2 )",
                [
                    new NumberToken(2), new AndToken(), new NumberToken(30),
                    new OrToken(), new LeftParensToken(), new NumberToken(2),
                    new OrToken(), new NumberToken(26), new AndToken(),
                    new NumberToken(2), new RightParensToken(),
                ],
            ],
        ];
    }
}
```

Fragment kódu 13 Jednotkový test syntaktického analyzátoru

```
class ParserTest extends TestCase
{
    /**
     * @dataProvider provideInvalid
     */
    public function testParserInvalid(string $expression,
                                      string $expectedException): void
    {
        $lexer = new Lexer($expression);
        $tokens = $lexer->tokenize();
        $parse = new Parser();
        self::expectException($expectedException);
        $parse->parse($tokens);
    }

    public static function provideInvalid(): array
    {
        return [
            "invalid token" => [
                " 1&2 2",
                UnexpectedTokenException::class,
            ],
            "unclosed parenthesis" => [
                " 1 | ( 4 & ( 2 ) | 2 ",
                UnexpectedEndOfStringException::class,
            ],
            "unopened parenthesis" => [
                "1 & 2 )",
                UnexpectedCharacterException::class,
            ],
            "operators next to each other" => [
                "6 & | 2",
                UnexpectedTokenException::class,
            ],
        ];
    }
}
```

5.2.2.2 Integrační testy

Integrační testy testují interakci mezi více třídami nebo komponentami. Má-li testovaná komponenta závislosti, je často používáno „mockování“ daných závislostí. Mockování tříd nám dovolí napodobit a definovat chování potřebné třídy. V případě testování služeb spravující objekty v databázi nabízí Doctrine definování entit do databáze – „data fixtures“ – a pak s nimi pracovat jako s PHP objekty. Integrační testy v Symfony vyžadují nastartování kernelu, ze

kterého poté můžeme tahat instance tříd. Namísto třídy TestCase se dědí ze třídy KernelTestCase.

Fragment kódu 14 Načtení kernelu a fixtures a úklid

```
class QuestionRepositoryTest extends KernelTestCase
{
    private QuestionRepository $repository;

    private AbstractDatabaseTool $databaseTool;

    private ReferenceRepository $referenceRepository;

    public function setUp(): void
    {
        parent::setUp();

        self::bootKernel();
        $container = static::getContainer();

        /** @var QuestionRepository $repository */
        $repository = $container->get(QuestionRepository::class);
        /** @var DatabaseToolCollection $databaseToolCollection */
        $databaseToolCollection = $container
                  ->get(DatabaseToolCollection::class);

        $this->repository = $repository;
        $this->databaseTool = $databaseToolCollection->get();

        $this->referenceRepository = $this->databaseTool->loadFixtures([
            AssignmentFixtures::class,
            QuestionFixtures::class,
        ])->getReferenceRepository();
    }

    public function tearDown(): void
    {
        parent::tearDown();

        unset($this->databaseTool);
    }
}
```

Fragment kódu 15 Integrační test získání otázek pomocí štítků

```
// QuestionRepositoryTest
/**
 * @dataProvider provideFindQuestionByTags
 */
public function testFindQuestionByTags(string $assignmentRef,
                                       TagNode $rootTagNode,
                                       array $expectedRefs): void
{
    /** @var Assignment $assignment */
    $assignment = $this->referenceRepository
           ->getReference($assignmentRef);
    $expected = $this->getReferences($expectedRefs);

    $questions = $this->repository
           ->findQuestionsByTags($assignment, $rootTagNode, []);

    self::assertEquals($questions, $expected);
}

private function getReferences(array $refs): array
{
    return array_map(function ($ref) {
           return $this->referenceRepository->getReference($ref);
    }, $refs);
}

public static function provideFindQuestionByTags(): array
{
    return [
        "1 | 5" => [
            AssignmentFixtures::ASSIGNMENT_5,
            TagNodeCreator::createNode([
                TagNodeCreator::createOrNode([
                    TagNodeCreator::createNumberNode(1),
                    TagNodeCreator::createNumberNode(5),
                ]),
            ]),
            [
                QuestionFixtures::QUESTION_6,
                QuestionFixtures::QUESTION_8,
                QuestionFixtures::QUESTION_11,
                QuestionFixtures::QUESTION_12,
            ],
        ],
    ];
}
```

5.2.2.3 Aplikační testy

Aplikační testy testují proces nad celou aplikací. V případě mikroslužby s rozhraním REST je dochází k poslání dotazu, jejího zpracování a validace výsledku. Symfony se třídou WebTestCase poskytuje namockovaný webový klient schopný posílat dotazy a i orientovat se v těle stránky pomocí HTML DOMu. Pro naše použití stačí posílání HTTP dotazů.

Fragment kódu 16 Aplikační test validace množiny štítků

```
class TagValidationControllerTest extends WebTestCase
{
    public function testTagsValidityNotFound(): void
    {
        $client = static::createClient();
        $client->jsonRequest('POST', '/tags/tags/validity', [
            "tags" => [1, 2, 200, 400, 3],
        ]);

        self::assertResponseStatusCodeSame(404);

        $response = $client->getResponse()->getContent();
        $content = json_decode($response);

        self::assertCount(2, $content);
        self::assertSame([200, 400], $content);
    }

    public function testTagsValidityDisjointConflict(): void
    {
        $client = static::createClient();
        $client->jsonRequest('POST', '/tags/tags/validity', [
            "tags" => [1, 3, 5, 6, 7, 8, 9],
        ]);

        self::assertResponseStatusCodeSame(409);

        $response = $client->getResponse()->getContent();
        $content = json_decode($response);
        $err = $content->error;

        self::assertStringContainsString("3 (tags: 5, 6)", $err);
        self::assertStringContainsString("4 (tags: 7, 8, 9)", $err);
    }
}
```

Tento test také využívá předdefinovanou sadu dat (fixtures).

5.3 Pokrytí

Pokrytí testů je míra měřící jak moc testy pokrývají na požadované vlastnosti, scénáře a chování. Základní operace na správu objektů – zobrazeni, tvorba, úprava a archivace – mají triviální logiku, kde automatizované testování nepřináší znatelnou výhodu oproti manuálnímu. Na parser jsou pro úpravu stromu, lexikální 12 a syntaktický 13 analyzátor použity jednotkové testy. Aplikační testy dotazů na rozhraní jsou použity na otestování dotazů syntaktické analýzy nebo nahrávání souborů.

Závěr

Výsledek

Cílem práce bylo zanalyzovat současný stav systému a vytvořit návrh na dvě části nového systému. Analytická část práce prošla nynější implementaci a požadavky, které z ní vychází. Z analýzy byl vytvořen návrh struktur a syntaktického analyzátoru pro vyhledávání. Výsledkem práce jsou syntaktický analyzátor a dvě implementované mikroslužby na správu zadání a správu štítků. Vytvořené systémy jsou otestovány na jednotkové a integrační testy.

Použití

Výstup práce je jeden z dílků skládačky. Momentálně uživatelské rozhraní vyvinuté mikroslužby nepokrývá, ale vývoj probíhá v plném proudu a v současné době slouží tyto dvě mikroslužby jako stavební kámen pro uživatelské rozhraní. Na mikroslužbě Assignments je závislá také správa testových šablon, která je pokryta v diplomové práci Radoslava Haška. Funkčnost štítků byla rozšířena a nyní může sloužit jako univerzální nástroj k filtrování v dalších systémech a uživatelském rozhraní.

Budoucnost

Obě mikroslužby byly vyvinuty s cílem zjednodušit úpravy a snížit celkovou komplexitu aplikace. Přijde-li nutnost rozšířit systém o nový typ dílčího zadání nebo otázek, měla by být změna bezbolestná. Rozšíření operátorů gramatiky analyzátoru vyžaduje pouze malý zásah do existujícího zdrojového kódu. Jak již bylo zmíněno, jedná se pouze o část plánovaného systému. V budoucnosti je možné očekávat další služby a související uživatelské rozhraní.

Literatura

[1] RegisFrey: The model, view, and controller (MVC) pattern relative to the user [online]. May 2010, [2023-03-07]. Dostupné z: https://commons.wikimedia.org/wiki/File:MVC-Process.svg

[2] Chandel, S.: Model-view-presenter Diagram [online]. 2009, [2023-03-07]. Dostupné z: https://www.gwtproject.org/images/testing_methodologies_mvp.png

[3] ivan111: Online JSON to tree diagram converter [online]. Jul 2014, [cit. 2023-05-04]. Dostupné z: https://vanya.jp.net/vtree/

[4] Codecademy: MVC: Model, view, Controller [online]. [cit. 2023-03-02]. Dostupné z: https://www.codecademy.com/article/mvc

[5] baeldung: Difference between MVC and MVP Patterns [online]. Nov 2022, [cit. 2023-03-02]. Dostupné z: https://www.baeldung.com/mvc-vs-mvp-pattern

[6] Model-view-presenter design pattern [online]. [cit. 2023-03-02]. Dostupné z: https://support.touchgfx.com/4.17/docs/development/ui-development/software-architecture/model-view-presenter-design-pattern

[7] Microservices [online]. [cit. 2023-03-19]. Dostupné z: https://www.ibm.com/topics/microservices

[8] Skryl, H.: 7 benefits of microservices architecture to know about [online]. Feb 2023, [cit. 2023-03-19]. Dostupné z: https://vilmate.com/blog/benefits-of-microservices-architecture/

[9] Slowly Changing Dimensions [online]. [cit. 2023-04-20]. Dostupné z: https://www.oracle.com/webfolder/technetwork/

tutorials/obe/db/10g/r2/owb/owb10gr2_gs/owb/lesson3/slowlychangingdimensions.htm

[10] Ross, M.: Design tip #152 slowly changing dimension types 0, 4, 5, 6 and 7 [online]. Jan 2016, [cit. 2023-04-20]. Dostupné z: https://www.kimballgroup.com/2013/02/design-tip-152-slowly-changing-dimension-types-0-4-5-6-7/

[11] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to automata theory, languages, and computation*. Upper Saddle River, NJ: Pearson, třetí vydání, Červen 2006, [cit. 2023-04-19].

[12] Šestáková, E.: *Automaty a gramatiky: sbírka vyřešených příkladů*. ČVUT, 2017, [cit. 2023-04-19].

[13] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, techniques, and Tools*, kapitola Parsing. Pearson India Education Services, 2007, [cit. 2023-05-02].

[14] Top-down parsing [online]. [cit. 2023-05-02]. Dostupné z: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803104932815

[15] Cockett, R.: LL(1) grammars and predictive top-down parsing [online]. Jan 2002, [cit. 2023-05-02]. Dostupné z: http://pages.cpsc.ucalgary.ca/~robin/class/411/LL1.2.html

[16] Maza, M. M.: LL(1) Grammars [online]. Apr 2004, [cit. 2023-05-02]. Dostupné z: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html

[17] Nelson, R. C.: Parsing. [cit. 2023-05-01]. Dostupné z: https://www.cs.rochester.edu/u/nelson/courses/csc_173/grammars/parsing.html

[18] What is php? [online]. [cit. 2023-02-22]. Dostupné z: https://www.php.net/manual/en/intro-whatis.php

[19] Grudl, D.: List of packages [online]. [cit. 2023-02-27]. Dostupné z: https://nette.org/en/packages

[20] PostgreSQL About [online]. [cit. 2023-02-23]. Dostupné z: https://www.postgresql.org/about/

[21] Introduction: Vagrant: HashiCorp developer [online]. [cit. 2023-02-25]. Dostupné z: https://developer.hashicorp.com/vagrant/intro

[22] anonymous007: Construction of LL(1) parsing table [online]. Mar 2023, [cit. 2023-05-01]. Dostupné z: https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/

[23] apgrgr: LL(1) parser generator [online]. Dec 2012, [cit. 2022-12-04]. Dostupné z: https://jsmachines.sourceforge.net/machines/ll1.html

[24] PHP: Releases [online]. [cit. 2023-02-22]. Dostupné z: https://www.php.net/releases/index.php

[25] Symfony: Symfony, high performance PHP framework for web development [online]. [cit. 2023-05-01]. Dostupné z: https://symfony.com/what-is-symfony

[26] DalPino, A.: PHP SplStack vs SplQueue vs Array [online]. Mar 2018, [cit. 2023-05-04]. Dostupné z: https://gist.github.com/andrewdalpino/492bbf4261d31dad5f847f9f4c42cbf9?permalink_comment_id=2368991#gistcomment-2368991

Příloha A

Seznam použitých zkratek

DBS Databázové Systémy

GUI Graphical User Interface

SQL Structured Query Language

PHP PHP: Hypertext Preprocessor

REST Representational State Transfer

JSON JavaScript Object Notation

ORM Object-relational Mapping

SCD Slowly Changing Dimension

HTML HyperText Markup Language

DOM Document Object Model

ČVUT České Vysoké Učení Technické

UC Use Case

AST Abstract Syntax Tree

DTO Data Transfer Object

Příloha **B**

Obsah přiloženého média

- `src`
  - `impl` ..... zdrojové kódy implementace
    - `Assignments` ..... mikroslužba Assignments
    - `Tags` ..... mikroslužba Tags
    - `DBS-Tags-Nodes` ..... balíček pro syntaktický analyzátor
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF