FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	dbs.fit.cvut.cz – Backend vyhodnocení testů
Student:	Jakub Pavličko
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 refaktoring části testového hodnocení portálu dbs.fit.cvut.cz, dále jen portál. Důvodem je nutná modernizace a pokrok k pohodlnějšímu a efektivnějšímu následnému vývoji. Při této úpravě se inspirujte prací Bc. Andriie Plyskache, který řešil testovou část.

Postupujte v těchto krocích:
1. Analyzujte současný stav portálu se zaměřením na část oprav a hodnocení testů. Nezapomeňte na provázanost s
ostatními částmi systému.
2. Na základě analýzy důkladně navrhněte strukturu nové části databáze věnované opravám a ohodnocením testů z portálu.
3. Nad databází postavte vhodný backend. Vzhledem k rozsahu se počítá minimálně s prototypovou implementací.
4. Na Vámi implementovaný kód realizujte vhodné testy – minimálně ukázkové jako šablonu pro další vývoj.
5. Navrhněte budoucí směr rozvoje.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 16. ledna 2023 v Praze.

Bakalárska práca

DBS.FIT.CVUT.CZ – BACKEND VYHODNOCENÍ TESTŮ

Jakub Pavličko

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Jiří Hunka
11. mája 2023

Odkaz na túto prácu: Pavličko Jakub. *dbs.fit.cvut.cz – Backend vyhodnocení testů.* Bakalárska práca. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov kódu

Chcel by som poďakovať predovšetkým vedúcemu práce Ing. Jiřímu Hunkovi za jeho odborné vedenie a usmernenie pri písaní tejto práce. Taktiež za rady a pripomienky, ktoré mi poskytol a v neposlednom rade za ochotu. Ďalej by som sa chcel poďakovať svojej rodine a priateľom, ktorí ma pri písaní tejto práce akýmkoľvek spôsobom podporovali.

Vyhlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom o dodržiavaní etických princípov pri príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, obzvlášť skutočnosť, že České vysoké učení technické v Prahe má právo na uzavretie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 citovaného zákona.

V Prahe dňa 11. mája 2023 .....

Abstrakt

Táto bakalárska práca sa zaoberá analýzou, návrhom a implementáciou mikroslužby pre hodnotenie testov v predmete BI-DBS na Fakulte informačných technológií ČVUT v Prahe. V predmete sa využíva DBS Portál, ktorý bol naimplementovaný študentmi FIT ČVUT. Portál je každoročne používaný stovkami študentov a od jeho vzniku nabral veľký technický dlh. Práca sa zaoberá modernizáciou časti portálu, ktorá slúži na správu a vyhodnocovanie odpovedí študentov vo forme mikroslužby. Práca bola písaná paralelne s bakalárskou prácou Tomáša Doubu a diplomovou prácou Bc. Radoslava Haška. Tieto tri práce majú za cieľ naimplementovať funkčnú verziu testového modulu. Výsledkom práce je mikroslužba s vystaveným rozhraním REST API, ktorá podporuje tvorbu a úpravu testových odpovedí, ako aj ich automatické a manuálne hodnotenie.

Klíčová slova DBS portál, mikroslužby, hodnotenie testov, automatické hodnotenie odpovedí, REST API, PHP, Symfony

Abstract

This bachelor thesis deals with the analysis, design and implementation of a microservice for test evaluation in the BI-DBS course at the Faculty of Information Technology of the Czech Technical University in Prague. The course uses the DBS Portal, which was implemented by students of the CTU FIT. The portal is used by hundreds of students every year and has gained a lot of technical debt since its launch. The thesis addresses the modernization of the part of the portal that is used to manage and evaluate student responses in the form of a microservice. The thesis was written in parallel with Tomáš Douba's bachelor thesis and Bc. Radoslav Hašek's master thesis. These three theses aim to implement a functional version of the test module. The result of the work is a microservice with an exposed REST API that supports the creation and editing of test answers, as well as their automatic and manual evaluation.

Keywords DBS portal, microservices, test evaluation, automatic answer evaluation, REST API, PHP, Symfony

Zoznam skratiek

API	Application Programming Interface
BI-DBS	Databázové systémy
BI-SP1	Softwarový týmový projekt 1
BI-SP2	Softwarový týmový projekt 2
CRUD	Create, Read, Update, Delete
ČVUT	České vysoké učení technické v Prahe
DBS portál	Systém pro podporu výuky předmětu BI-DBS
ER	Entity-relationship
FIFO	First In, First Out
FIT	Fakulta informačních technologií
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
OOP	Objektovo orientované programovanie
ORM	Object Relational Mapping
RA	Relačná algebra
REST	Representational State Transfer
SDLC	Software Development Life Cycle – životný cyklus softvérového projektu
SQL	Structured Query Language
UI	User Interface
YAML	Yet Another Markup Language

Úvod

Predmet BI-DBS – Databázové systémy – je povinný predmet bakalárskeho štúdia vyučovaný na Fakulte informačných technológií (FIT) ČVUT v Prahe. Každoročne ho absolvujú stovky študentov, ktorí si z predmetu odnášajú nielen základné znalosti teórie o databázových systémoch, ale aj praktické skúsenosti s ich používaním.

V praktickej časti predmetu zohráva veľkú úlohu práve DBS portál. Ten má vo výučbe 2 hlavné funkcie – podpora pre tvorbu semestrálnych prác a písanie zápočtových a skúškových testov. Ide o softvér, ktorého funkcionality boli vyvíjané študentmi FIT-u v rámci bakalárskych či diplomových prác, ale aj v rámci predmetov Softvérový projekt 1 a Softvérový projekt 2.

Dnešná podoba systému bola spustená približne pred 8 rokmi. To je vo svete IT doba, za ktorú mnoho technológií zostarne a vzniknú nové, ktoré sú na konkrétne potreby vhodnejšie. To sa stalo aj v prípade DBS portálu a vznikla preto potreba systém reimplementovať. K tejto potrebe prispel aj fakt, že sa na projekte vystriedalo veľké množstvo tímov, ktoré mali vždy čas niekoľko mesiacov počas semestra projekt pochopiť a doimplementovať doň plánované veci. Vzhľadom na veľkú komplexitu portálu bola táto úloha relatívne náročná, obzvlášť pre menej skúsených programátorov, čo sa odzrkadlilo na konzistencii a kvalite výsledného kódu.

Vzhľadom na potrebu vývoja tohto softvéru podobným štýlom aj naďalej, bolo navrhnuté použitie architektúry mikroslužieb. Systém sa rozdelí do viacerých menších služieb, ktoré budú mať omnoho menší záber a budú teda jednoduchšie na pochopenie, vývoj a následnu údržbu. Niektoré základné mikroslužby už boli naimplementované, na tie hlavné sa však ešte nedostalo.

Jednou z nich sú práve testy. Tie však zahŕňajú viacero samostatných častí. Niektoré z nich budú naimplementované v rámci bakalárskej práce Tomáša Doubu, iné v diplomovej práci Bc. Radoslava Haška.

Táto práca sa bude venovať mikroslužbe na vyhodnocovanie odpovedí študentov v testoch. Najskôr bude vypracovaná podrobná analýza časti systému, ktorý sa tejto problematike venuje v súčasnosti. Na základe týchto poznatkov bude vytvorený návrh novej mikroslužby. Práca bude zahŕňať aj samotnú implementáciu mikroslužby vrátane jej testovania.

Keďže ide o backend systému, pre jeho použitie v praxi bude potrebné naimplementovať aj frontend, ktorý bude využívať služby backendu. Vo výsledku z týchto mikroslužieb a frontendu vznikne systém, ktorý bude plne schopný nahradiť aktuálny testový modul. Výsledok práce bude prospešný pre študentov a vyučujúcich Databázových systémov na FIT ČVUT v Prahe.

Ciele práce

Hlavným cieľom tejto práce je modernizácia systému na vyhodnocovanie odpovedí študentov v testoch pre predmet Databázové systémy. Systém bude prerobený do formy mikroslužby s vystaveným API. Hodnotenie odpovedí budú môcť vyučujúci vykonať manuálne, avšak systém bude schopný opravovať odpovede aj automaticky. Okrem toho bude zabezpečovať aj ukladanie odpovedí študentov a referenčných odpovedí.

Najskôr sa podrobne zanalyzuje súčasná implementácia tejto funkčnosti v DBS portáli. Analýza bude zahŕňať najmä algoritmy a postupy použité na automatické vyhodnocovanie odpovedí. Tieto poznatky budú zhrnuté vo funkčných a nefunkčných požiadavkách, ktoré sa doplnia o potrebné vylepšenia. Na základe nich bude vytvorený návrh mikroslužby, podľa ktorého sa systém naimplementuje a otestuje, pokrývajúc všetky funkčnosti aktuálneho systému. Nakoniec bude navrhnutý budúci rozvoj systému s ohľadom na výsledky tejto práce.

Kapitola 1

Softvérový vývoj a architektúra

Napriek tomu, že vývoj softvéru prebieha už desiatky rokov, neustále sa prichádza s novými prístupmi či metódami, ako tento proces zjednodušiť a urýchliť. Ide o oblasť, v ktorej sa nedá jedno pravidlo aplikovať na všetko. Existuje veľké množstvo metodík, ktorými sa dá pri vývoji riadiť. Je však nutné vedieť identifikovať, ktorá celý proces zjednoduší a ktorá ho naopak skomplikuje.

V tejto kapitole budú porovnané 2 tradičné prístupy k vývoju softvéru podľa SDLC. Taktiež sa tu popisujú bežné praktiky, ktoré zvyšujú kvalitu dodávaného kódu. Ďalej budú porovnané mikroslužby s „tradičnou“ monolitickou architektúrou. V poslednej časti sa vysvetlí dôraz, ktorý sa v poslednej dobe kladie na paralelizáciu algoritmov a historizáciu dát.

1.1 Životný cyklus softvéru

Vývoj softvéru je komplexný proces. Pre tvorbu kvalitného softvéru je potrebné, aby sa k celému procesu tvorby pristupovalo systematicky. O to sa stará Software Development Life Cycle (SDLC). Ide o množinu aktivít, ktoré sú nevyhnutné, aby mohol vzniknúť softvér. SDLC jednotlivé aktivity popisuje a definuje ich následnosť.

Dané aktivity sú zobrazené na obrázku 1.1 a tu je popis, čo zahŕňajú.

- **Plánovanie** – zber požiadaviek, analýza nákladov a prínosov, tvorba časového plánu, odhad potrebných zdrojov
- **Návrh** – analýza požiadaviek a výber konkrétneho riešenia každej z nich
- **Implementácia** – rozdelenie požiadaviek na malé časti, ktoré sa postupne implementujú
- **Testovanie** – odhaľovanie chýb v kóde pomocou automatického a manuálneho testovania
- **Nasadenie** – nasadenie softvéru do produkcie, kde si ho vie zákazník sám vyskúšať
- **Údržba** – spracovanie spätnej väzby od zákazníka, oprava nájdených chýb, monitoring výkonu a zabezpečenia pre ďalšie zlepšovanie [1]

Existuje niekoľko modelov, ktoré využívajú fázy SDLC. Každý z nich si však tieto fázy chronologicky zoraďuje inak s cieľom optimalizovať vývoj. Medzi 2 najznámejšie patria *vodopádový* a *iteratívny model*.

Obr. 1.1 Software Development Life Cycle

1.1.1 Vodopádový model

Vodopádový model je prvým modelom, ktorý využíval aktivity definované v SDLC modeli. Aktivity nasledujú chronologicky počas celého procesu vývoja. Na začiatku sa vo fáze *plánovania* získajú a zdokumentujú všetky možné vstupy a požiadavky od zákazníka. Tieto požiadavky spravidla pokrývajú všetky funkčnosti budúceho systému. Následne sa podľa nich *navrhne* celá jeho architektúra. Na základe výstupov z návrhovej časti sa systém postupne *implementuje* a *testuje* bez ďalšieho vstupu od zákazníka. Na záver sa celý systém *nasadí* a je pripravený na akceptáciu zákazníkom. Po ukončení nasadenia začína *podpora*, v ktorej sa opravujú chyby, prípadne vylepšia existujúce funkčnosti.

Jednotlivé fázy vodopádového modelu idú zaradom a neprekrývajú sa. Ak sa určitá fáza ukončí, model už neumožňuje spätne sa k nej vrátiť a upraviť ju. V praxi to znamená, že ak sa na začiatku určia požiadavky, celý vývoj je na nich viazaný a zmeny v neskorších fázach sú veľmi zložité a nežiadúce.

Tento model má, samozrejme, svoje výhody aj nevýhody.

Výhody

- jednoduché odhadovanie zdrojov potrebných na vývoj
- každá fáza má jasné ciele
- dobrá dokumentácia

Nevýhody

- nutnosť znalosti všetkých požiadaviek pred začiatkom vývoja
- zložitá zmena požiadaviek po ukončení plánovacej fázy
- zákazník nedostáva čiastočné výsledky, nevie teda projekt smerovať počas vývoja
- nevhodný na dlhodobý vývoj – ak sa odkloní od očakávaní, príde sa na to neskoro
- nasadenie prebieha ako tzv. „big-bang“ na konci vývoja [2]

1.1.2 Iteratívny model

Hlavným princípom iteratívneho modelu je postupné vylepšovanie jednoduchých riešení. Na začiatku vývoja sa zozbierajú požiadavky, ktoré sa týkajú časti systému implementovanej v danej iterácii. Narozdiel od vodopádu je množstvo požiadaviek, ktoré je potrebné vedieť na začiatku projektu podstatne menšie. Pre konkrétnu časť systému sa spraví návrh, implementácia, testovanie a následne sa nasadí.

Vývoj sa uskutočňuje v niekoľkých iteráciách, v ktorých sa postupne upresňujú požiadavky a získava sa spätná väzba od zadávateľa. Tieto zmeny sa v iteráciach zapracovávajú a na konci každej je nasadená funkčná časť systému. Každá iterácia prechádza všetkými fázami SDLC. Po skončení iterácií je nasadená kompletná verzia softvéru.

Výhody

- priebežné nasadzovanie funkčného softvéru
- včasné odhalenie chýb v návrhu
- jednoduchšia zmena požiadaviek ako pri vodopáde
- kladie dôraz na spätnú väzbu od zákazníka

Nevýhody

- nevhodný na menšie projekty, ktoré sa nedajú rozdeliť na menšie celky
- nemožnosť presne ohraničiť trvanie vývoja [3]

1.2 Analýza požiadaviek

Proces zbierania a analýzy požiadaviek je kľúčovou súčasťou softvérového vývoja. Umožňuje viesť projekt správnym smerom a je jeho oporným bodom. Na základe požiadaviek sa zvyčajne na konci projektu vyhodnotí jeho úspešnosť. Keďže výstup býva tradične v jazyku, ktorému rozumie zadávateľ, je veľmi dôležitý pri vzájomnej komunikácii. V tejto sekcii budú bližšie priblížené najčastejšie prístupy k identifikácii, dokumentovaniu a prioritizácii softvérových požiadaviek.

Požiadavky sa všeobecne delia na funkčné a nefunkčné.

- **Funkčné** – popisujú funkcionalitu, ktorá musí byť naimplementovaná, aby systém umožnil plniť potrebné úlohy.
- **Nefunkčné** – nesúvisia s funkcionalitou systému, definujú vlastnosti systému. [4]

Analýza požiadaviek je nevyhnutná, či už vo väčšej alebo menšej miere, v každom projekte. Existuje preto niekoľko modelov, ktoré definujú štruktúru a rozdelenie požiadaviek pre dosiahnutie konkrétneho cieľa. Medzi najznámejšie patrí *FURPS* a *MoSCoW*.

1.2.1 FURPS

FURPS je akronymom pre kategórie požiadaviek – *Functionality*, *Usability*, *Reliability*, *Performance*, *Supportability* a *Security*. Bol vyvinutý spoločnosťou Hewlett-Packard a prvýkrát publikovaný v roku 1987. [5]

Požiadavky sa teda rozdelia do nasledujúcich kategórií:

- **Functionality** (Funkčnosť) – popisujú sa tu všeobecné funkcionality systému, interakcia s inými systémami, zabezpečenie...

- **Usability** (Použiteľnosť) – určujú, ako má systém vyzerať, ovládať sa alebo rýchlo reagovať
- **Reliability** (Spoľahlivosť) – popisuje napríklad kedy by mal byť systém k dispozícii alebo ako často môže zlyhať
- **Performance** (Výkon) – zahŕňa dobu odozvy systému, množstvo dátového toku či dobu spustenia
- **Supportability** (Podporovateľnosť) – popisujú sa tu vlastnosti systému ako testovateľnosť, jednoduchosť údržby alebo podpora

FURPS bolo neskôr rozšírené na FURPS+, ktoré zahŕňa okrem 5 základných kategórií aj určité nové, ktoré v pôvodnom modeli chýbali. Medzi ne patria:

- **Design constraints** (Obmedzenia návrhu) – obmedzenia na hardvérovú či softvérovú platformu, databázu či sieťové vlastnosti.
- **Implementation requirements** (Požiadavky na implementáciu) – obmedzenia v spôsoboch, akým je softvér vyvíjaný (napríklad držanie sa rôznych noriem či štandardov).
- **Interface requirements** (Požiadavky na rozhranie) – popisujú spôsoby potrebnej interakcie s daným systémom z pohľadu iných systémov (aké dáta má poskytovať, ich formát...)
- **Physical requirements** (Fyzické požiadavky) – požiadavky na hardvér a fyzické zariadenia, ktoré bude systém používať (minimálny výkon procesora, maximálny odber prúdu...) [6, s. 92 – 93]

1.2.2 MoSCoW

MoSCoW sa narozdiel od FURPS zameriava na prioritizáciu požiadaviek. Začleňuje ich do 4 kategórií, v ktorých postupne klesá priorita – *Must*, *Should*, *Could*, *Won't*.

- **Must** – definuje, čo systém musí spĺňať, aby bol považovaný za úspešný
- **Should** – požiadavky, ktoré by systém mal zahŕňať, ak to je možné (nemusia byť v prvej verzii systému)
- **Could** – funkcionality, ktoré nie sú kritické, ak sa nestihnú naimplementovať v prvej verzii, môžu byť zahrnuté neskôr alebo vynechané
- **Won't** – vymedzenie vecí, ktoré systém nebude obsahovať [6, s. 85]

1.3 Mikroslužby

Pri tradičnej monolitickej architektúre, kde je aplikácia písaná ako jeden celok, sú pri väčších systémoch niektoré aktivity, ako napríklad oprava chýb, škálovanie či nasadzovanie do produkcie, pomerne komplikované. Oveľa vhodnejšie je pre takéto systémy využiť architektúru mikroslužieb.

Architektúra mikroslužieb patrí do moderného spôsobu vývoja softvéru. Spočíva v rozdelení veľkého systému na súbor malých, samostatne fungujúcich, častí, ktoré spolu komunikujú na základe vystavených rozhraní.

Medzi ich hlavné benefity patria nasledovné.

- **Technologická rôznorodosť** – Mikroslužby umožňujú použitie rôznych technológií v rozličných častiach jedného systému, vďaka čomu je možné vybrať najlepší nástroj pre každú funkcionalitu a optimalizovať tak celkový výkon systému.

- **Škálovateľnosť** – V rámci aplikácie podliehajú jej rôzne časti aj rôznej záťaži. Pre lepšie zvládanie záťaže je výhodné vedieť systém škálovať tam, kde je to potrebné. Pri monolitických aplikáciách sú len 2 možnosti – škálovať alebo neškálovať. Ak sa aplikácia škáluje, musí sa škálovať celá, čím sa pridelí napríklad viac hardvérových prostriedkov aj častiam systému, ktoré ich vôbec nepotrebujú. Pri mikroslužbách je možné prideliť prostriedky len častiam, ktoré to potrebujú a fungovať tak efektívnejšie. Taktiež je možné systém decentralizovať jeho rozdelením na viacero zariadení.
- **Odolnosť** – Ak zlyhá nekritická služba v systéme postavenom na mikroslužbách, ostatné služby môžu naďalej fungovať bez nej. Pri monolitických systémoch však zlyhanie jedného, aj menej dôležitého, komponentu, môže znamenať pád celého systému.
- **Nasaditeľnosť** – Akákoľvek malá zmena kódu v monolite znamená nutnosť nasadiť celý systém naraz. To môže priniesť nemalé problémy, čo v praxi spôsobuje oddiaľovanie nasadenia systému do produkcie. To zase znamená nazbieranie veľkého množstva zmien, ktoré opäť zvyšujú riziko zlyhania systému. Pri mikroslužbách sa celý tento proces výrazne zjednodušuje, keďže je možné nasadiť jednotlivé služby samostatne. Ak v systéme aj nastane chyba, je jednoduché ju nájsť, opraviť a opätovne nasadiť v omnoho kratšom čase v porovnaní s monolitom.

Mikroslužby však nie sú všeobecným riešením. Okrem mnohých výhod majú aj svoje slabé stránky. Medzi ne patrí napríklad *bezpečnosť*. Keďže každá mikroslužba má vystavené rozhranie, pri vývoji sa musí dbať na dôkladné zabezpečenie každého z nich. Komunikácia medzi mikroslužbami prebieha prostredníctvom siete, čo znamená, že v porovnaní s monolitom budú mať mikroslužby vyššie *oneskorenie*. Pre využitie potenciálu mikroslužieb je dôležité mať dobre nastavené a automatizované *procesy*, ako je testovanie, nasadzovanie a monitorovanie, čo môže byť na začiatku vývoja časovo náročné a nákladné. [7, s. 19 – 27]

1.4 Škálovateľnosť

Škálovateľnosť a paralelizácia sú kľúčové koncepty, ktoré hrajú veľkú rolu pri stavaní výkonných aplikácií. Škálovateľnosť je schopnosť systému prispôsobiť sa zmenám v aplikácii a vo výpočtovej záťaži. V takýchto situáciách je systém schopný zvýšiť alebo znížiť výkon tak, aby optimalizoval dobu odozvy a cenu prevádzky. [8] Delí sa na vertikálnu a horizontálnu.

1. **Vertikálna (Scaling up)** – pridávanie hardvérových prostriedkov (zvýšenie výkonu procesorov, pridanie pamäťových jednotiek) do jedného zariadenia
2. **Horizontálna (Scaling out)** – spájanie viacerých zariadení (s vlastným procesorom a pamäťovými jednotkami) do jedného funkčného celku [9]

Vertikálna škálovateľnosť je menej výhodná, pretože pre zvýšenie výkonu je potrebné zabezpečiť výkonnejší hardvér. Takéto zvyšovanie je však obmedzené a nedá sa v ňom pokračovať donekonečna. Okrem toho je takéto škálovanie aj finančne náročné, keďže cena hardvéru od určitého bodu nie je priamoúmerná výkonu, ale je násobne vyššia. Preto je dôležité pri odhadovaní záťaže systému myslieť ďaleko dopredu, aby sa hardvér nemusel meniť príliš často.

Na druhej strane, horizontálne škálovanie vyžaduje omnoho nižšie počiatočné investície, keďže krátko po spustení systémy zväčša nepodliehajú vysokému náporu. V prípade potreby sa pridá hardvér, pričom jeho výkon nemusí byť násobne vyšší ako to je pri vertikálnom škálovaní. Pridanie rovnako výkonného servera vie v ideálnom prípade takmer zdvojnásobiť výkon aplikácie.

Vertikálna škálovateľnosť zväčša nevyžaduje žiadne zásahy do návrhu systému, avšak pre horizontálnu je správny návrh kľúčový. Systém pracuje paralelne na viacerých zariadeniach, ktoré majú oddelené úložiská aj procesory. Je preto dôležité, aby bola aplikácia bezstavová všade, kde to je možné. Čím viac sa toto podarí dosiahnuť, tým jednoduchšie je škálovanie, pretože systém môže bežať nezávisle na viacerých zariadeniach bez potreby akejkoľvek synchronizácie. [10]

answer_id	*value*	*is_demo*	*created_by*
1	abc	false	1247

Tabuľka 1.1 Príklad tabuľky answer

1.4.1 Asynchrónna komunikácia

Pre zabezpečenie horizontálnej škálovateľnosti je kľúčové minimializovať závislosti medzi jednotlivými komponentmi. Jedným z mechanizmov, ktoré tomu napomáhajú je komunikácia medzi službami prostredníctvom tzv. *Message queues*. Ide o spôsob asynchrónnej komunikácie, ktorej princípom je ukladanie správ do dátovej štruktúry FIFO (konkrétne queue) jednou službou a ich následné odoberanie a spracovanie druhou. Vďaka tomu je možné, aby každá služba pracovala svojou vlastnou rýchlosťou a nie je potrebné, aby na seba čakali. Správy sú zvyčajne malé a obsahujú informácie ako požiadavky, odpovede či chybové hlášky. Účastníkmi komunikácie sú vždy 2 strany – producent a konzument. Producent vkladá správy do queue a konzument ich odtiaľ odoberá. Napriek tomu, že jedna queue môže byť používaná viacerými producentmi aj konzumentmi, každá správa je spracovaná práve raz a práve jedným konzumentom. [11]

Okrem zjednodušenia škálovania je výhodou tohto spôsobu komunikácie aj vyššia odolnosť voči chybám. Ak by systém konzumenta zlyhal, producent môže nezávisle na ňom pridávať správy do queue. Tie sa môžu spracovať neskôr po obnovení fungovania konzumenta. Znižuje sa teda stratovosť informácií pri páde systému. [12]

1.5 Historizácia a audit dát

Hlavným cieľom uchovávania zmien v databáze, je spätné dohľadávanie, aké zmeny boli na dátach uskutočnené, kedy boli uskutočnené a kto je za ne zodpovedný. Toto je možné docieliť pomocou tzv. auditovania databázy. Ide o proces, v ktorom sa sledujú, a pre účely spätnej kontroly aj ukladajú, akcie, ktoré sú vykonávané s databázou. To má okrem iného velké využitie v udržiavaní jej zabezpečenia. [13, s. 29 – 30]

Keďže databázy majú veľké množstvo funkcionalít, takisto existuje aj veľké množstvo kategórií, v ktorých sa môže audit realizovať. Najbežnejšie z nich sú:

- **Audit prihlasovania do databázy** – kto a odkiaľ sa prihlásil
- **Audit zdrojov používania** – klienti, ktorí pristupujú k databáze
- **Audit databázových chýb** – toto je obzvlášť dôležité pre zabezpečenie databázy, keďže pri SQL injection útokoch útočníci často skúšajú opakovane posielať dotazy, až kým sa im nepodarí dosiahnuť ich cieľ
- **Audit DDL zmien** – zmeny štruktúry databázových objektov
- **Audit DML zmien** – zmeny v dátach uložených v databáze
- **Audit DQL dotazov** – prístup k dátam [13, s. 349 – 374]

Nižšie sú bližšie vysvetlené 3 z týchto spôsobov – *audit tabuľka, verzovanie (versioning)* a *snapshoty*[1]. Každý z nich bude vysvetlený pomocou príkladu, v ktorom sa bude používať tabuľka *answer* so 4 stĺpcami – *answer_id, value, is_demo, created_by*. Do nej vložíme jeden záznam s odpoveďou k demo testu. Následne záznam upravíme a zmeníme prepínač *is_demo* na false. Hlavná tabuľka bude teda vyzerať ako tabuľka 1.1.

[1] Stav záznamu v danom okamihu

id	*p_key*	*operation*	*column*	*old_val*	*new_val*	*created_by*	*created_at*
1	1	C	value	null	abc	1247	2023-02-18
2	1	C	is_demo	null	true	1247	2023-02-18
3	1	C	created_by	null	1247	1247	2023-02-18
4	1	U	is_demo	true	false	1247	2023-03-27

Tabuľka 1.2 Príklad audit tabuľky

1.5.0.1 Audit tabuľka

Ide o štruktúru, ktorej záznamy reprezentujú jednotlivé zmeny v dátach hlavnej tabuľky. Funguje ako sekundárna tabuľka k jednej alebo viacerým hlavným tabuľkám, ktoré obsahujú len aktuálne záznamy.

V databáze sa pri tomto spôsobe logovania nastaví trigger, ktorý pri každej zmene spustí skript. Ten pri uskutočnení zmeny v hlavnej tabuľke vytvorí záznam v audit tabuľke. Tento záznam typicky obsahuje informácie o *aktérovi*, ktorý danú zmenu vykonal (ID, IP adresa atď.), *čase zmeny*, *type akcie* (*CREATE/UPDATE/DELETE*), *stĺpci*, ktorý bol zmenený a obsahuje *starú a novú hodnotu*. V prípade, že sa audit tabuľka používa pre viacero hlavných tabuliek, je potrebné pridať informáciu o konkrétnej tabuľke, v ktorej sa zmena vykonala. Toto logovanie funguje separátne pre každý stĺpec. Ak sa teda v rámci jednej zmeny zmenia 2 stĺpce, v log tabuľke sa vytvoria 2 záznamy. Je možné, aj odporúčané, vybrať z každej tabuľky podmnožinu stĺpcov, ktoré sa budú sledovať a ich zmeny uchovávať.

Fungovanie tohto spôsobu je možné vidieť v tabuľke 1.2. V audit tabuľke sa budú nachádzať záznamy typu *CREATE* na všetky stĺpce (okrem primárneho kľúča, ktorý sa generuje automaticky) a následne jeden záznam typu *UPDATE* na stĺpec *is_demo*.

Výhodou tohto spôsobu ukladania zmien je *nízka pamäťová náročnosť*, keďže sa do databázy ukladá minimálne množstvo informácií, ktoré je potrebné na zrekonštruovanie dát v akomkoľvek čase. Ďalšou výhodou je, že hlavná tabuľka obsahuje vždy len najaktuálnejšiu verziu záznamu. Vo väčšine prípadov sa v aplikácii pracuje s aktuálnymi dátami, historické nás zaujímajú len v špecifických prípadoch. To znamená, že bežná interakcia s databázou bude rýchlejšia ako v prípade uchovávania všetkých zmien v hlavnej tabuľke.

Nevýhodou je naopak, že v prípade potreby vytvorenia histórie konkrétneho záznamu je potrebné prejsť všetky záznamy a postupne vyskladať obraz v jednotlivých časoch, čo je relatívne výpočtovo náročné.

1.5.0.2 Verzovanie

Verzovanie záznamov sa naopak uskutočňuje v rámci hlavnej tabuľky a nie sú teda potrebné žiadne dodatočné štruktúry. Ku každému záznamu sa do tabuľky pridá informácia o verzii záznamu (čím vyššia verzia, tým novší záznam) alebo rozmedzie časov platnosti záznamu. Príklad môžeme vidieť v Tabuľke 1.3

Výhodou tohto spôsobu ukladania je jednoduchý prístup k celému záznamu v ľubovoľnom čase v histórii, nie je potrebné ho nijako spracovávať.

Na druhej strane veľkou nevýhodou je, že pri každej zmene sa zmení ID daného záznamu, čo môže byť problém, keďže na toto ID sú často viazané iné záznamy, ktoré by bolo potrebné tiež aktualizovať. Ďalšou nevýhodou je, že pri vytváraní nového záznamu je celý záznam skopírovaný. Tým pádom je pamäťová náročnosť podstatne vyššia ako pri separátnej logovacej tabuľke. Keďže má hlavná tabuľka aj viac záznamov, práca s aktuálnou verziou záznamov môže byť pomalšia ako pri oddelenej logovacej tabuľke.

answer_id	*value*	*is_demo*	*created_by*	*effective_from*	*effective_to*
1	abc	true	1247	2022-02-18	2023-03-27
2	abc	false	1247	2023-03-27	null

Tabuľka 1.3 Príklad verzovanej tabuľky answer

1.5.0.3 Snapshoty

Ukladanie snapshotov je takmer identické s verzovaním v hlavnej tabuľke. Jediný rozdiel je, že sa tieto snapshoty ukadajú do separátnej tabuľky. Týmto sa stráca nevýhoda, že rovnaký záznam má rôzne ID pre rôzne historické verzie. Zároveň sa zachováva výhoda jednoduchej práce s predchádzajúcimi verziami záznamu. Príklad je takmer identický s tabuľkou 1.3 s jedným rozdielom, že namiesto *effective_from* a *effective_to* je iba jeden stĺpec s časovým razítkom zmeny. [14]

Kapitola 2
Analýza

Vývoj softvéru je komplexný proces. Pre tvorbu kvalitného softvéru je potrebné, aby sa k celému procesu tvorby pristupovalo systematicky. Neoddeliteľnou súčasťou toho je správne pochopenie požiadaviek prostredníctvom podrobnej analýzy.

V prípade DBS portálu sa pracuje s už existujúcim softvérom, ktorý v priebehu rokov zostarol ako technologicky, tak aj kvalitou kódu. Systém je na pokraji udržateľnosti a pre ďalší vývoj je nevyhnutné, aby bol naimplementovaný v modernejších technológiách a s dôrazom na kvalitný objektový dizajn, ktorý bude jednoducho rozšíriteľný aj do budúcna. Tento proces sa odborne nazýva *Software Reengineering*. Skladá sa z 2 krokov – reverse engineering a forward engineering. Reverse engineering zahŕňa analýzu kódu, všetkej dostupnej dokumentácie a získavanie informácií od používateľov a ľudí, ktorí systém udržiavajú. Po reverse engineeringu nasleduje fáza forward engineeringu, v ktorej sa berú do úvahy získané poznatky z analýzy a na základe nich sa tvorí nový softvér. [15]

V rámci tejto kapitoly sa čitateľ v krátkosti oboznámi s DBS portálom a jeho praktickým využitím v rámci výuky na FIT ČVUT, rozoberú sa jeho aktuálne funkčnosti a problémy, pričom sčasti bude využitý reverse engineering a v poslednej časti popíšem nové riešenie backendu testového modulu založeného na báze mikroslužieb, ktoré vychádza z bakalárskej [16] a diplomovej [17] práce Ing. Andriia Plyskacha.

2.1 DBS portál

DBS portál je informačný systém využívaný na výučbu predmetu Databázové systémy (BI-DBS) na Fakulte informačných technológií ČVUT v Prahe. Portál plne podporuje výučbu predmetu a poskytuje študentom nástroje na tvorbu konceptuálnych modelov, relačných schém či spúšťanie dotazov v SQL aj RA nad ľubovoľnou databázou, čím si poznatky nadobudnuté na prednáškach upevňujú a využívajú ich v praxi.

Portál zohráva kľúčovú úlohu najmä v dvoch oblastiach – semestrálne práce a testy.

2.1.1 Semestrálne práce

Nutnou podmienkou na splnenie podmienok predmetu a udelenie zápočtu je vypracovanie semestrálnej práce. V nej si každý študent vyberie tému, ktorej sa bude v priebehu semestra venovať. Cieľom je vytvoriť databázu, nad ktorou sa dajú vykonávať dotazy. V semestrálnej práci si študenti prakticky vyskúšajú veľkú časť učiva z prednášok a proseminárov. To zahŕňa najmä:

- Vytvorenie konceptuálnej schémy databázy

- Vytvorenie tabuliek a ich naplnenie dátami pomocou SQL
- Vymyslenie dotazov v zadaných kategóriach v prirodzenom jazyku
- Transformáciu dotazov z prirodzeného jazyka do RA a SQL

Semestrálna práca je rozdelená do troch iterácií, pričom každá z nich má zadané kritériá, ktoré musia študenti splniť. Systém tieto kritéria kontroluje z veľkej časti automaticky, pričom učiteľ vždy po odovzdaní semestrálnej práce študentom dané kritériá dodatočne zvaliduje.

Podrobnejšie informácie je možné nájsť v práci Jakuba Lukačína, ktorý sa venoval práve refaktorovaniu backendu semestrálnych prác a podrobne popisuje celé riešenie systému. [18]

2.1.2 Testy

Nemenej podstatnou časťou portálu je písanie zápočtových a skúškových testov. V minulosti sa testy písali na papier a boli manuálne opravované vyučujúcimi, čo bolo časovo náročné pre učiteľov a užívateľsky nepríjemné pre študentov. Neskôr vznikol systém pre podporu výuky, ktorý bol však používaný málo pre jeho veľké nedostatky. Na základe tohto systému neskôr vznikol nový, ktorý mal lepší návrh a položil základ pre aplikáciu, ktorá sa používa dodnes. Jadro testového modulu napísal Petr Pejša vo svojej bakalárskej práci. [19] V priebehu rokov sa tento systém vyvíjal v rámci bakalárskych a diplomových prác a predmetov BI-SP1 a BI-SP2. Aktuálne systém podporuje tvorbu, vypĺňanie a opravu testov, v ktorých sú študenti preverovaní z pochopenia kľúčových tém, ktoré sa preberajú počas semestra. Oprava je z veľkej časti automatizovaná, čo výrazne šetrí čas učiteľom a zároveň skracuje dobu odozvy študentom, ktorí majú test opravený rýchlejšie.

Pre lepšie pochopenie testového systému bude vysvetlené, ako jeho jednotlivé časti fungujú a spolupracujú. V rámci testov sa pracuje s nasledujúcimi entitami:

- Testový termín
- Testová šablóna
- Varianta testu
- Zadanie
- Otázka
- Odpoveď

2.1.2.1 Zadanie

Zadanie vytvára kontext, ktorý študenta uvedie do problematiky úlohy a poskytne mu všetky informácie, ktoré sú potrebné na jej vypracovanie. Pri vytváraní zadania je nutné vybrať jeden z typov: *text, diagram, obrázok* alebo *normalizácia relačnej schémy*. Systém taktiež umožňuje vytvoriť komplexnejšie zadanie, v ktorom sa skombinujú viaceré zadania – napríklad textový popis a databázový diagram, čím je možné špecifikovať takmer akýkoľvek kontext.

Príklad: *(Textové zadanie) Majme fragment relačnej schémy:* ***zviera****(id, meno, druh, datum_narodenia)*

2.1.2.2 Otázka

Otázka v teste upresňuje zadanie a dáva študentovi konkrétnu úlohu, ktorú má pri vypracovaní splniť. Pri jej vytváraní je potrebné okrem samotného znenia otázky, vyplniť aj kategóriu, na základe ktorej sa môžu otázky automaticky dosadiť do dynamickej testovej šablóny. Aby sa

otázka mohla v použiť v nejakom teste, musí mať priradené aspoň jedno zadanie a aspoň jednu referenčnú odpoveď, na základe ktorej sa bude vyhodnocovať správnosť odpovede študenta.

Príklad: *Pomocou SQL vyberte zviera (id, meno), ktoré je staršie ako 2 roky. [kategória „SQL dotaz“]*

2.1.2.3 Odpoveď

Odpoveď môže byť dvoch druhov – *referenčná* alebo *odpoveď študenta*.

Referenčná odpoveď sa nastavuje pri vytváraní otázky a využíva sa ako pri automatickom, tak aj pri manuálnom opravovaní na vyhodnotenie správnosti odpovede študenta.

Odpoveď študenta vzniká v priebehu testu, keď študent vyplní odpoveď na otázku a odošle ju. Z odpovede študenta sa po jej opravení stáva referenčná odpoveď, na základe ktorej sa opravujú odpovede na rovnakú otázku od iných študentov.

Spôsob uloženia odpovede v databáze a celková práca s odpoveďou pri opravovaní závisí od jej typu. Typy odpovedí sú:

- **Checkbox** – výber z možností, viacero správnych odpovedí
- **Radio button** – výber z možností, jedna správna odpoveď
- **Text** – slovná odpoveď
- **Diagram** – konceptuálny model databázy vytvorený pomocou kresliaceho nástroja
- **SQL** – databázový dotaz v jazyku SQL
- **RA** – databázový dotaz v relačnej algebre
- **Transformácia** – prevod z ER do relačnej schémy
- **Normalizácia** – prevod relácií do normálnych foriem

2.1.2.4 Testová šablóna

Testová šablóna je vzor testu, z ktorého sa po priradení do termínu stanú inštancie testu (pre každého študenta vznikne samostatná inštancia). Šablóna má niekoľko typov: *demo test, automatický demo test, test v semestri, skúška*. Podľa spôsobu vloženia otázok sa rozlišuje šablóna *statická* a *dynamická*. Pri *statickej* musí byť priradená aspoň jedna konkrétna otázka. Týmto spôsobom bude mať každá inštancia testu z tejto šablóny rovnaké otázky. *Dynamická* sa líši najmä v spôsobe priraďovania otázok. Namiesto konkrétnych otázok sa do šablóny priradia podmienky, ktoré musia automaticky vygenerované otázky spĺňať. Konkrétna podoba testu teda vzniká až pri vzniku jeho inštancie. Dynamická šablóna umožňuje, že každý študent môže mať vygenerovaný iný test.

2.1.2.5 Inštancia testu

Pre umožnenie písania testu je potrebné vytvoriť inštancie testu. Tie vznikajú z konkrétnej šablóny. Následne sa vytvorí inštancia testu, ktorá má priradených študentov. Ak bola šablóna dynamická, inštancia testu už obsahuje konkrétne vygenerované otázky.

2.2 Vyhodnocovanie odpovedí

Táto sekcia sa zaoberá aktuálnou implementáciou hodnotenia testov DBS portálu. Budú tu popísané algoritmy, ktoré sa používajú v jednotlivých situáciách, externé závislosti systému a problémy, ktoré súčasná implementácia prináša. Tie budú v návrhovej časti opravené.

Vyhodnocovanie testov je časovo aj organizačne veľmi náročnou časťou skúšania študentov. Pri ručnom opravovaní odpovedí sa naráža na problémy, ako sú opakujúce sa odpovede, ktoré musí opravovateľ dôkladne skontrolovať, aby sa uistil, že sú identické s už opravenými (a ohodnotil ich rovnako) alebo odpovede, ktorými sa nemá zmysel zaoberať (syntaktické chyby, prázdna odpoveď). DBS portál sa snaží maximalizovať efektivitu opravovania aj tým, že zjednodušuje uvedené situácie. Robí to prostredníctvom dvoch mechanizmov – *automatické opravovanie* a *asistované manuálne opravovanie*.

2.2.1 Automatická oprava

Vďaka tomu, že každá otázka v teste má priradenú referenčnú odpoveď, je systém schopný automaticky vyhodnotiť zhodu študentovej odpovede s referenčnou. V prvom kroku sa skontroluje, či odpoveď nie je prázdna. Ak áno, automaticky sa ohodnotí 0 bodmi. Ak je odpoveď identická s referenčnou, automaticky sa vyhodnotí ako stopercentne správna a systém je schopný priradiť jej plný počet bodov. V týchto prípadoch nie je nutný žiaden zásah od človeka. Ak však odpoveď nie je prázdna a zhoda nie je stopercentná, vo väčšine prípadov zásah od človeka potrebný je.

Keďže typy odpovedí sú zásadne odlišné, aj spôsob ich vyhodnocovania sa líši. Základný postup sa však opakuje pri takmer všetkých odpovediach s výnimkou odpovedí typu radio button. Ten zahŕňa nasledujúce kroky

1. Kontrola prázdnosti odpovede
2. Porovnanie 100 % zhody s referenčnou odpoveďou
3. Porovnanie 100 % zhody s už opravenými odpoveďami študentov v rámci testu

Vyhodnocovanie odpovedí sa spúšťa v momente odoslania testu prípadne vypršania časového limitu na test. V prípade, že je testov v jeden moment odoslaných príliš veľa (to môže nastať najmä pri vypršaní časového limitu), systém sa môže zahltiť a neexistuje žiadny asynchrónny mechanizmus, ktorý by dokázal záťaž rozložiť. To je jedna zo slabých stránok systému.

2.2.1.1 Checkbox

Odpoveď typu checkbox je reprezentovaná poľom možností, ktoré sú označené ako správne. Pri oprave sa kontroluje množinový rozdiel odpovede študenta a referenčnej odpovede. Ak je prázdny, znamená to, že sú odpovede identické a odpoveď sa ohodnotí plným počtom bodov. Ak je neprázdny, systém pokračuje a porovnáva odpoveď s inými odpoveďami študentov, ktoré už boli v rámci testu ohodnotené. Ak sa nájde zhoda (zvolené rovnaké možnosti), odpovedi sa priradí rovnaký počet bodov ako opravenej zhodnej odpovedi. V opačnom prípade systém nie je schopný odpoveď vyhodnotiť a posiela sa do manuálnej opravy.

2.2.1.2 Radio button

Oprava tohto typu odpovede je plne automatická, keďže systém len vyhodnotí, či sa odpoveď zhoduje s referenčnou (plný počet bodov) alebo nezhoduje (0 bodov).

2.2.1.3 Text

Textová odpoveď je na automatickú opravu asi najzložitejšia. Stačí, aby mal študent vo svojej odpovedi preklep a odpovede sa považujú za rôzne napriek tomu, že významovo sú identické. Systém ich teda nie je schopný automaticky opraviť. Je niekoľko úprav, ktoré sa vykonávajú na texte, aby sa podobné chyby minimalizovali:

- Prevod textu na malé písmená
- Transformácia znakov UTF-8 na ASCII znaky

Texty sa po týchto úpravách porovnajú a zistí sa ich editačná vzdialenosť[1]. Následne sa vypočíta hodnotenie pomocou vzorca

$$100 - \frac{L(x,\ y) \times 100}{\max(|x|, |y|)},$$

kde $|x|$ vyjadruje počet znakov reťazca x a $L(x,\ y)$ editačnú vzdialenosť x a y. Ak je toto hodnotenie rovné 100, odpoveď sa automaticky opraví, v opačnom prípade sa navrhne opravujúcemu, ktorý bude danú odpoveď manuálne hodnotiť.

Ďalej pokračuje a porovnáva úplnú zhodu s už opravenými odpoveďami ostatných študentov. V tomto kroku sa editačná vzdialenosť nevyužíva z dôvodu časovej zložitosti algoritmu, keďže je dĺžka odpovede teoreticky neobmedzená a pri použití upraveného algoritmu na výpočet editačnej vzdialenosti s použitím memoizácie je časová zložitosť $\mathcal{O}(mn)$. [20, s. 27]

Takéto porovnávanie by teda bolo príliš neefektívne a spomaľovalo by celý proces opravovania.

2.2.1.4 Diagram

Odpoveď typu diagram, je v systéme reprezentovaná ako JSON objekt, viď A.1. Má 2 reprezentácie – jednu pre frontend a jednu pre backend. Prvou, frontendovou, je objekt, ktorý obsahuje pozície jednotlivých entít a vzťahov medzi nimi na osi x a y, ktoré sa využívajú na správne zobrazenie v kresliacom nástroji. Druhou je objekt, ktorý sa využíva na backende na automatickú opravu. Ten narozdiel od prvého neobsahuje pozičné informácie.

Postup pri oprave je opäť rovnaký – porovnanie s referenčnými odpoveďami a následne s odpoveďami študentov. Pri nenájdení zhody je odpoveď poslaná na manuálnu opravu.

Porovnanie s referenčnými odpoveďami je vykonávané algoritmicky. Z databázy sa vezmú všetky referenčné odpovede a pre každú sa spustí algoritmus na porovnávanie. Ten berie do úvahy a porovnáva nasledujúce parametre diagramu.

- **Entity** – ich celkový počet, názvy, počty atribútov a rodičovské entity (pri použití ISA hierarchie)
- **Atribúty entít** – názov, typ, ďalej sa kontroluje, či sú unikátne, súčasťou primárneho kľúča a nulovateľné
- **Väzby medzi entitami** – celkový počet, zdrojové a cieľové entity (nezáleží na tom, ktorá je zdrojová a ktorá cieľová, musia však byť v správnej kombinácii), násobnosť väzby, jej voliteľnosť a či je určujúca

[1] Vzdialenosť reťazcov x a y je daná najmenším počtom editačných operácií (vymazanie znaku, vloženie znaku, nahradenie znaku iným znakom), ktoré sú potrebné na transformáciu reťazca x na reťazec y. [20, s. 27]

Obr. 2.1 Príklad diagramu v DBS portáli

2.2.1.5 SQL

Pri odpovedi typu SQL je v prvom kroku porovnaný SQL dotaz ako reťazec. Porovnanie sa uskutoční najskôr s referenčnou odpoveďou a potom s odpoveďami ostatných študentov. Ak dôjde k zhode, odpovedi sa automaticky priradí hodnotenie.

V opačnom prípade je potrebné SQL dotaz spustiť v databáze. Je možné, že dotaz je syntakticky nesprávny a následne je ohodnotený 0 bodmi. Ak je možné dotaz vykonať, vykoná sa aj referenčný dotaz a ich výsledky sa porovnajú. Pri zhode sa priradí plný počet bodov, inak smeruje odpoveď do manuálnej opravy.

Porovnávanie výsledkov s dotazmi ostatných študentov nie je prínosné. Výsledky dotazov, ktoré boli hodnotené plným počtom bodov musia byť nutne identické s výsledkami referenčného dotazu. Pri dotazoch, ktoré boli hodnotené menším počtom bodov je používanie porovnania výsledkov nesprávne, keďže by mohlo dôjsť k chybným opravám. Napríklad, ak dotaz študenta je takmer správny, ale má malú chybu v podmienke, výsledok môže byť prázdny. Ak by iný študent napísal dotaz, ktorý je kompletne zlý, ale taktiež má prázdny výsledok, systém by ich vyhodnotil ako identické a udelil im rovnaký počet bodov, čo je nesprávne.

2.2.1.6 Relačná algebra

Odpovede typu relačnej algebry sú takmer identické s odpoveďami typu SQL. Jediný rozdiel je, že pred samotným spustením dotazu je potrebné previesť relačnú algebru na jej reprezentáciu v SQL. Na to sa používa RAT (Relational algebra translator) – nástroj, ktorý naimplementoval Martin Kubiš vo svojej bakalárskej práci „Překladač z relační algebry do SQL“ [21] a následne zrefaktoroval a rozšíril v diplomovej práci „Rozšíření překladače relační algebry“ [22]. Nástroj má definovaný API endpoint, ktorý prijíma dotaz v RA, typ databázy a jej schému a vracia transformovaný dotaz v SQL.

2.2.1.7 Transformácia

Oprava transformácie sa vykonáva s pomocou aplikácie TralexNew, ktorú navrhol a naimplementoval vo svojej bakalárskej práci Martin Šach. [23] Aplikácia je napísaná v Kotline a kompiluje sa do JavaScriptu, čo umožňuje jej jednoduché použitie v rámci DBS portálu. Aplikácia slúži na transformáciu odpovede vo forme reťazca na štandardizovaný JSON objekt, s ktorým je možné ďalej jednoducho pracovať.

Pri oprave sa teda študentova aj referenčná odpoveď transformuje pomocou Tralexu a následne sa porovnávajú JSON objekty. Na základe rozdielov v rôznych častiach objektu sa automaticky udeľuje skóre. Na udelenie plného počtu bodov je možné spraviť v transformácii určitý počet chýb. Pre každý druh chyby je definovaná penalizácia. Ak je výsledná penalizácia menšia ako zadaný limit, odpovedi sa udelí plný počet bodov. V opačnom prípade odpoveď putuje do manuálnej opravy.

Na frontende sa pri danej odpovedi zobrazuje percentuálna zhoda odpovede s referenčnou odpoveďou, ktorá je počítaná na základe penalizácie vypočítanej na backende. Okrem toho je tam dopočítavané aj navrhované hodnotenie, ktoré tiež vyplýva z penalizácie.

2.2.1.8 Normalizácia

DBS portál je schopný opraviť odpovede typu normalizácia plne automaticky. Túto opravu zabezpečuje knižnica Data normalization vytvorená Marekom Erbenom v rámci bakalárskej práce. [24] Keďže normalizácia je komplexnejší typ otázky, ktorá obsahuje viacero podúloh a zložitosť týchto podúloh je rôzna, každý typ má priradenú váhu. Knižnica obsahuje metódy na opravu všetkých týchto typov a vracia percentuálne hodnotenie odpovede vyjadrené desatinným číslom. Každá podúloha sa tak ohodnotí podľa vzorca

$$\frac{\textit{váha podúlohy}}{\textit{súčet všetkých váh}} \cdot \textit{hodnotenie podúlohy}.$$

Po sčítaní týchto bodov pre podúlohy získame celkové percentuálne hodnotenie celej odpovede.

2.2.2 Manuálna oprava

Ak sa odpoveď nepodarí opraviť automaticky, je potrebné, aby odpoveď opravil učiteľ ručne. Portál však výrazne zjednodušuje prácu aj v tejto oblasti. Ako bolo spomenuté pri automatickej oprave, odpovede sa porovnávajú s odpoveďami ostatných študentov, ktoré už boli opravené. To znamená, že ak do manuálneho opravovania spadne 10 ekvivalentných odpovedí, ktoré systém nebol schopný opraviť, stačí, že opravujúci ohodnotí jednu z týchto odpovedí a automaticky sa ohodnotí aj ostatných 9. Tým sa podstatne šetrí čas strávený pri opravovaní.

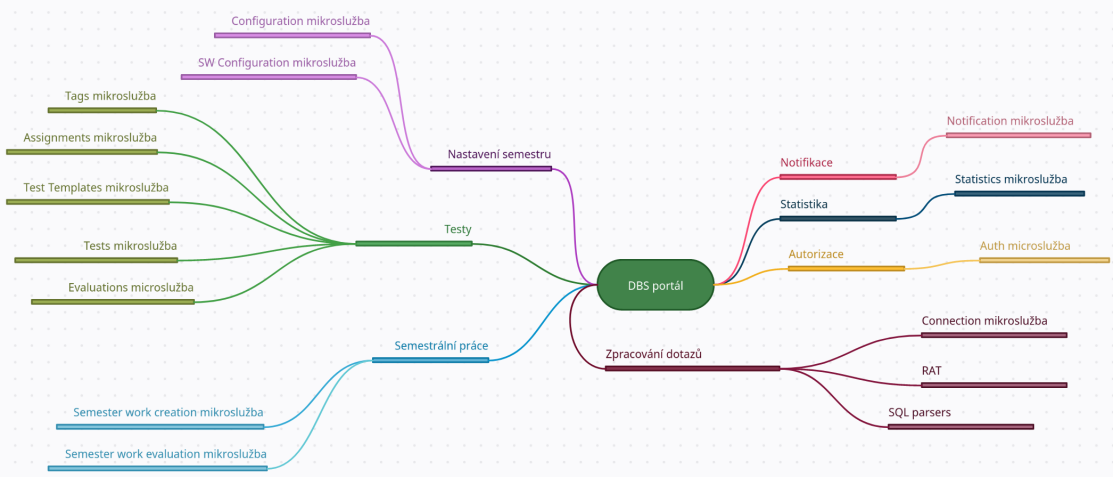
2.2.3 Mikroslužby a DBS portál

Backend DBS portálu sa v letnom semestri roku 2021 začal refaktorovať a implementovať s využitím architektúry mikroslužieb. Základy toho položil Ing. Andrii Plyskach vo svojej bakalárskej práci [16], v ktorej sa venoval návrhu refaktorovania portálu s cieľom zmodernizovať ho a zefektívniť ďalší vývoj. Od napísania tejto práce sa architektúra nového portálu mierne upravila a vyladili sa nedostatky, ktoré sa odhalili v priebehu implementácie. Tieto zmeny sú popísané podrobne v jeho diplomovej práci. [17] V nasledujúcich častiach budú v krátkosti zhrnuté hlavné princípy fungovania mikroslužieb v DBS portáli a zásady, ktoré sa dodržiavajú pri vývoji.

Portál bol teda rozdelený do mikroslužieb (viď obrázok 2.2), ktoré majú vystavené API a komunikujú spolu. Doposiaľ boli naimplementované nasledujúce 4 mikroslužby:

1. **Auth** – mikroslužba zabezpečujúca autentifikáciu a autorizáciu používateľov.
2. **Configuration** – mikroslužba slúžiaca na import dát zo systému KOS, konfiguráciu predmetu a získavaním informácií o používateľovi, akými sú jeho role, kurzy či predmety, v ktorých sa nachádza.
3. **SW Configuration** – mikroslužba zabezpečujúca nastavenie semestrálnej práce (rozdelenie do iterácií, body za jednotlivé iterácie, podmienky pre iterácie...) a semestra.
4. **Connection** – mikroslužba slúžiaca na spúšťanie dotazov nad rôznymi databázami a porovnávanie výsledkov viacerých dotazov.

Každá z mikroslužieb má vlastnú databázu, ku ktorej bude mať prístup iba ona sama. Dáta budú medzi mikroslužbami prúdiť prostredníctvom volaní jednotlivých API endpointov, čo mierne skomplikuje udržiavanie konzistencie dát, avšak na druhej strane výrazne zjednoduší prácu s databázou v rámci mikroslužby, keďže nebude potrebné rozlišovať tabuľky, s ktorými mikroslužba reálne pracuje od ostatných tabuliek.



Obr. 2.2 Rozdelenie mikroslužieb z diplomovej práce Ing. Andriia Plyskacha [17]

2.3 Požiadavky na nový systém

Proces získavania požiadaviek je kľúčový pre zabezpečenie úspešnosti projektu. Požiadavky poskytujú popis potrieb a očakávaní, ktoré sa kladú na systém a zároveň pomáhajú smerovať projekt počas celého vývoja. Na základe nich je možné projekt naplánovať, systém naimplementovať a v neposlednom rade ho pri odovzdaní otestovať. Znalosť toho, čo má systém robiť ešte pred začatím samotnej implementácie, má pozitívny vplyv na celkový dizajn softvéru, ako aj na kvalitu kódu a efektivitu pri vývoji. [25, s. 217 – 218]

Vzhľadom na to, že DBS portál už funguje dlhšiu dobu a bola v ňom vyladená väčšina problémov, majorita funkčných a nefunkčných požiadaviek bola získaná pomocou reverse engineeringu práve z existujúcej implementácie a validovaná počas konzultácií s ľuďmi, ktorí DBS portál používajú a udržiavajú. Do úvahy boli samozrejme brané aj zmeny, ktoré by systém zlepšili a v aktuálnej implementácii chýbali.

Pri analýze bude využitý model FURPS+ popísaný v 1.2. Jednotlivé požiadavky teda budú začlenené do 5 kategórií. Ďalej bude ku každej požiadavke z kategórie funkčnosti doplnená jej priorita podľa MoSCoW (must have, could have, should have, won't have) a zložitosť implementácie (jednoduchá, stredná, zložitá)

Functionality – funkčnosť

- **Rôzne formáty odpovedí**
  Systém musí podporovať spracovanie odpovedí v rôznych formátoch (SQL, RA, diagram, transformácia, checkbox, radio, textová odpoveď) s možnosťou ich jednoduchého rozšírenia o nové formáty.

  Priorita: *Must*, Zložitosť: *Stredná*

- **Automatické hodnotenie**
  Ak je odpoveď na 100 % zhodná s referenčnou odpoveďou, systém musí byť schopný takúto odpoveď vždy opraviť automaticky bez nutnosti zásahu učiteľa.

  V prípade, že odpoveď sa úplne nezhoduje, systém navrhne hodnotenie podľa odpovedí študentov z minulosti. Spôsob výpočtu navrhovaného hodnotenia musí byť jednoducho rozšíriteľný.

  Priorita: *Must*, Zložitosť: *Vysoká*

- **Manuálne hodnotenie**
  Každá odpoveď študenta môže byť ohodnotená manuálne. Ak bola odpoveď ohodnotená automaticky, toto hodnotenie je možné prepísať manuálnym hodnotením.

  Pri hodnotení odpovede môže učiteľ zadať buď počet bodov za odpoveď alebo percentuálnu správnosť odpovede, na základe ktorej sa vypočíta bodové hodnotenie.

  Systém musí byť schopný rozlíšiť automatickú opravu od manuálnej opravy.

  Priorita: *Must*, Zložitosť: *Nízka*

- **Verzovanie odpovedí**
  Systém bude podporovať tvorbu viacerých verzií jednej odpovede. Verzie odpovedí budú vytvárané automaticky pri každej zmene. Študent bude mať možnosť vrátiť sa k predošlej uloženej verzii.

  Priorita: *Should*, Zložitosť: *Stredná*

- **Identifikácia podvádzania**
  Systém musí byť schopný identifikovať a informovať učiteľa o prípadoch, kedy študent píše test z rôznych zariadení.

  Priorita: *Should*, Zložitosť: *Nízka*

- **Diskusia, poznámky**
  Učiteľ aj automat musí byť schopný pri odoslaní hodnotenia pridať poznámku, ktorá bude obsahovať komentár k danému hodnoteniu. Študent, ku ktorého odpovedi sa hodnotenie vzťahuje, musí mať možnosť reagovať na hodnotenie prostredníctvom diskusie. Tej sa môže zúčastniť študent a ľubovoľný učiteľ.

  Priorita: *Could*, Zložitosť: *Nízka*

Usability – použiteľnosť

- **API dokumentácia** – systém musí mať zrozumiteľné API, ktoré je dobre zdokumentované pre jeho jednoduché používanie. Dokumentácia bude prístupná a bude zahŕňať všetky endpointy, ktoré je možné volať.

Reliability – spoľahlivosť

- **Automatická oprava** – systém musí automaticky vyhodnotiť všetky odpovede, ktoré sa na 100 % zhodujú s referenčnou odpoveďou.

- **Ekvivalentné odpovede** – systém by mal udržiavať konzistenciu v ohodnotených odpovediach. Ak sa upraví hodnotenie na odpovedi, rovnaké hodnotenie sa nastaví aj na ostatných ekvivalentných bez potreby manuálneho nastavovania.

Performance – výkon

- **Škálovateľnosť** – Systém musí byť jednoducho horizontálne škálovateľný. Musí podporovať viacvláknové automatické hodnotenie odpovedí pre možnosť ich paralelného spracovania.

- **Vysoká záťaž** – systému musí zvládnuť vysokú záťaž počas písania testov. V jeden moment môže prísť niekoľko stoviek odpovedí.

Supportability – podporovateľnosť

- **Testovanie** – kľúčové oblasti systému musia byť pokryté jednotkovými alebo integračnými testami.
- **Rozšíriteľnosť** – kód musí byť písaný s využitím princípov OOP tak, aby bol jednoducho rozšíriteľný a udržiavateľný.
- **Integrácia** – API mikroslužby musí byť možné využívať v rámci ostatných mikroslužieb.

Kapitola 3
Návrh

V predchádzajúcej kapitole bolo popísané aktuálne fungovanie DBS portálu, ako aj architektúra nového portálu. Táto kapitola sa zameriava na konkrétny návrh mikroslužby na hodnotenie testov. Návrh sa bude skladať z troch častí – *databáza, softvérové štruktúry* a *automatická oprava*.

3.1 Databáza

Ako bolo spomenuté v analýze, nové mikroslužby v DBS portáli fungujú na princípe vlastnej databázy pre každú službu. Mikroslužba na hodnotenie testov teda bude mať taktiež svoju databázu. Návrh vznikol na základe podrobnej analýzy domény a bol sčasti inšpirovaný návrhom z bakalárskej práce Ing. Andriia Plyskacha [16].

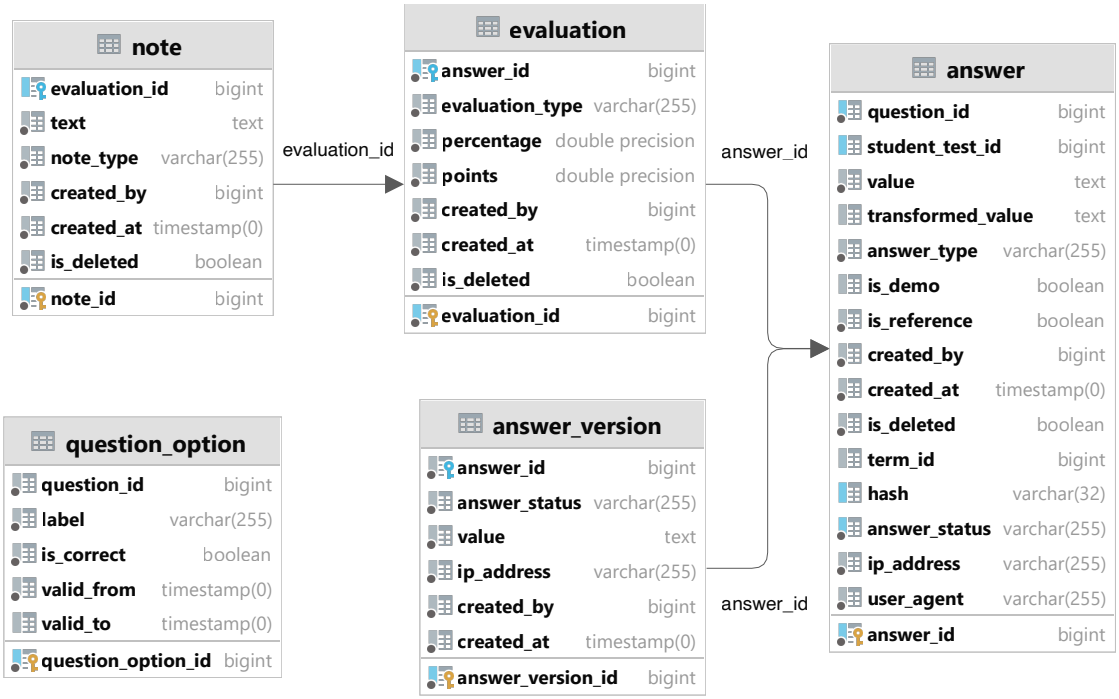
3.1.1 Štruktúry

Systém bude pozostávať z 5 tabuliek – *answer*, *answer_version*, *evaluation*, *note* a *question_option*. Databázový diagram je možné vidieť na obrázku 3.1

Answer

Tabuľka answer bude uchovávať informácie o všetkých odpovediach. Bude mať nasledujúce stĺpce.

- **answer_id** – Primárny kľúč tabuľky, automaticky generovaná sekvencia.
- **question_id** – ID otázky, ku ktorej odpoveď patrí.
- **student_test_id** – ID konkrétneho vygenerovaného testu pre konkrétneho študenta (smeruje do mikroslužby pre písanie testov).
- **term_id** – ID testového termínu pre konkrétneho študenta (smeruje do mikroslužby pre písanie testov).
- **answer_status** – Stav odpovede, slúži na rozlíšenie, či ide o automaticky uloženú odpoveď alebo bola uložená manuálne študentom. Možné hodnoty sú *auto_save* a *manual_save*.
- **value** – Hodnota odpovede.
- **transformed_value** – Spracovaná odpoveď.(napr. RA preložené na SQL, aby sa pri opravovaní eliminovala nutnosť prekladu).



Obr. 3.1 Databázový diagram mikroslužby

- **hash** – Zahashovaná normalizovaná hodnota odpovede – využíva sa na porovnanie 100 % zhody 2 odpovedí.
- **answer_type** – Typ odpovede.
- **is_demo** – Príznak, ktorý signalizuje, či je daná odpoveď na otázku v demo teste, v tom prípade sa odpoveď neposiela na manuálnu opravu.
- **is_reference_answer** – Príznak, ktorý signalizuje, či je odpoveď referenčná zadaná učiteľom alebo odpoveď pochádza od študenta, ktorý odpovedal na danú otázku v teste.
- **ip_address** – IP adresa zariadenia, z ktorého bola odoslaná daná odpoveď – spolu so stĺpcom *user_agent* sa využíva na kontrolu podvádzania pri písaní testu.
- **user_agent** – Informácie o prehliadači, z ktorého bola odoslaná odpoveď.
- **created_by** – ID používateľa, ktorý vytvoril odpoveď.
- **created_at** – Dátum a čas vytvorenia odpovede.

Answer_version

Tabuľku *answer* dopĺňa tabuľka *answer_version*, ktorá slúži na ukladanie predchádzajúcich odpovedí študenta. Spôsob ich ukladania je popísaný v sekcii 1.5.0.3 s jednou úpravou – nebudú sa verzovať všetky stĺpce hlavnej tabuľky. Ukladať sa bude iba minimum informácií, ktoré je potrebné na umožnenie študentovi vrátiť sa k staršej odpovedi. Tieto záznamy sa budú taktiež používať na kontrolu podvádzania (stĺpce *ip_address* a *user_agent*).

Tabuľka teda bude mať vlastný primárny kľúč **answer_version_id** a cudzí kľúč do tabuľky answer – **answer_id**. Ďalej bude obsahovať nasledujúce stĺpce taktiež z tabuľky answer, ktorých význam je nezmenený – **answer_status**, **value**, **ip_address**, **user_agent**, **created_by** a **created_at**.

Evaluation

Tabuľka *evaluation* bude obsahovať všetky hodnotenia odpovedí študentov. Bude mať nasledujúce stĺpce.

- **evaluation_id** – Primárny kľúč tabuľky, automaticky generovaná sekvencia.
- **answer_id** – ID odpovede, ktorá je hodnotená.
- **evaluation_type** – Typ hodnotenia, možné hodnoty podľa spôsobu ohodnotenia sú *manual* (manuálne hodnotenie učiteľom), *reference* (ohodnotené na základe referenčnej odpovede) a *match* (ohodnotené na základe identických odpovedí študentov).
- **percentage** – Percentuálna správnosť odpovede.
- **points** – Bodové hodnotenie odpovede.
- **created_by** – ID používateľa, ktorý vytvoril hodnotenie.
- **created_at** – Dátum a čas vytvorenia hodnotenia.
- **is_deleted** – Príznak, ktorý signalizuje, či je hodnotenie vymazané.

Evaluation_log

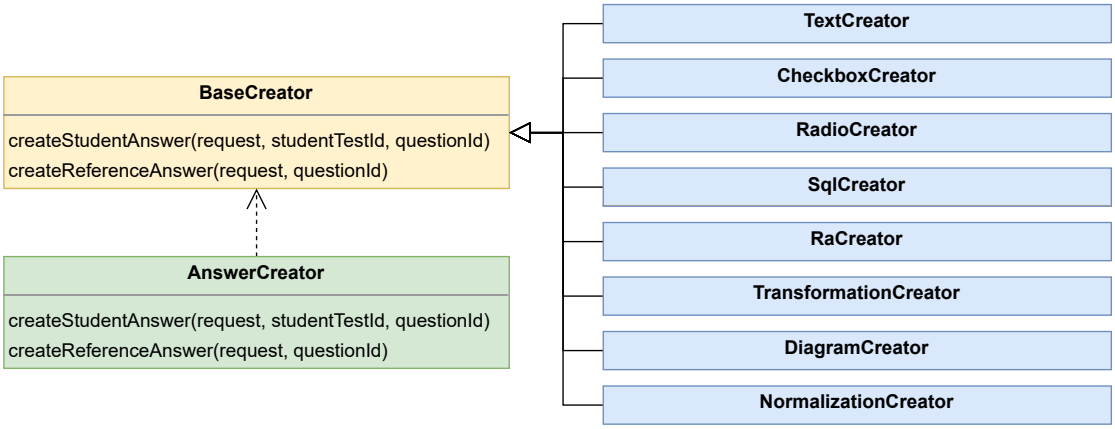
Vzhľadom na to, že hodnotenie sa bude meniť automaticky pri synchronizovaní s ostatnými ekvivalentnými odpoveďami, je dôležité, aby sa zachovala história jeho zmien. Z tabuľky evaluation sa tu budú ukladať nasledujúce hodnoty.

- **evaluation_log_id** – Primárny kľúč tabuľky, automaticky generovaná sekvencia.
- **evaluation_id** – Cudzí kľúč do tabuľky evaluation.
- **evaluation_type** – Typ hodnotenia.
- **percentage** – Percentuálna správnosť odpovede.
- **points** – Bodové hodnotenie odpovede.
- **created_by** – ID používateľa, ktorý je autorom zmeny.
- **created_at** – Dátum a čas vytvorenia logu.
- **is_deleted** – Príznak, ktorý signalizuje, či je hodnotenie vymazané.

Note

Tabuľka *note* bude obsahovať poznámky k hodnoteniu odpovedí. Každé hodnotenie môže mať viacero poznámok. Táto štruktúra bude slúžiť aj na prípadnú diskusiu k hodnoteniu. Tabuľka bude obsahovať nasledujúce údaje.

- **note_id** – Primárny kľúč tabuľky, automaticky generovaná sekvencia.
- **evaluation_id** – ID hodnotenia, ktorému poznámka patrí.
- **note_type** – Typ poznámky, možné hodnoty sú *evaluation_comment* a *discussion*.
- **created_by** – ID používateľa, ktorý vytvoril poznámku.
- **created_at** – Dátum a čas vytvorenia poznámky.
- **is_deleted** – Príznak, ktorý signalizuje, či je poznámka vymazaná.



Obr. 3.2 Diagram tried pre ukladanie odpovedí

Question_option

Mikroslužba bude okrem samotných odpovedí študentov spravovať tvorbu a prácu s možnosťami pri otázkach, v ktorých sa vyberá z viacerých možností. Na to slúži tabuľka *question_option*.

- **question_option_id** – Primárny kľúč tabuľky, automaticky generovaná sekvencia.
- **question_id** – ID otázky, ku ktorej daná možnosť patrí.
- **label** – Text možnosti.
- **is_correct** – Príznak, ktorý signalizuje, či je daná možnosť správna.
- **valid_from** – Dátum a čas, od ktorého je možnosť platná.
- **valid_to** – Dátum a čas, do ktorého je možnosť platná.

Keďže sa možnosti v priebehu času môžu upravovať, bude potrebné ich historizovať. Ako vyplýva z popisu stĺpcov, záznamy sa budú verzovať priamo v tabuľke, ako je popísané v 1.5.0.2.

3.2 Ukladanie odpovedí

Rôzne typy odpovedí majú aj rôzne formáty, ktoré je potrebné podporovať a pracovať s nimi. Je preto potrebné vytvoriť vhodnú abstrakciu, aby bolo v budúcnosti jednoduché pridať nový typ odpovede bez potreby veľkého zásahu do kódu.

Na to sa hodí *Strategy pattern*. Ide o behaviorálny návrhový vzor, ktorý sa využíva, keď je potrebné použiť rôzne algoritmy na spracovanie objektu v závislosti od jeho typu. [26] Návrh tried je možné vidieť na obrázku 3.2.

Trieda `AnswerCreator` sa bude používať na ukladanie odpovedí v rámci mikroslužby. Interne bude využívať stratégie, ktoré dedia z triedy `BaseCreator`. Tá obsahuje základnú implementáciu metód `createStudentAnswer()` a `createReferenceAnswer()`. Táto implementácia vloží hodnoty, ktoré dostane v parametri *request* do novej entity a tú uloží do databázy bez akéhokoľvek ďalšieho spracovania. Všetky triedy s výnimkou `RadioCreator` a `NormalizationCreator` však používajú vlastné stratégie na spracovanie odpovedí. Hneď po prijatí odpovede sa jej hodnota spracuje v závisosti od jej typu a táto spracovaná hodnota sa uloží do stĺpca *transformed_value* tabuľky *answer*. Hlavným cieľom tejto transformovanej hodnoty je normalizácia. Tým sa zvýši pravdepodobnosť zhody s inými odpoveďami, ktoré sú významovo identické, avšak majú napríklad iné formátovanie.

Tu je krátky popis transformácie hodnoty pre jednotlivé typy odpovedí, ktoré nevyužívajú základnú implementáciu.

- **Text** – Textová odpoveď sa očistí od nadbytočných medzier znakov na začiatku, na konci aj uprostred reťazca, znaky UTF-8 sa nahradia za znaky ASCII a všetky veľké písmená sa zmenia na malé.
- **Checkbox** – Odpoveď typu checkbox je reprezentovaná poľom *answer_option_id*. Toto pole sa pri transformácií zoradí vzostupne.
- **Radio button** – Hodnota je ID, ktoré zodpovedá *answer_option_id*. Nie je potrebné vykonávať žiadnu normalizáciu, využíva sa teda základná implementácia.
- **Diagram** – Diagram pochádza z kreslítka, ktoré daný JSON objekt normalizuje a všetky entity, atribúty aj vzťahy sú zoradené podľa preddefinovaných kritérií. Preto nie je potrebné vykonávať žiadne ďalšie úkony pri transformácii.
- **SQL** – Pre transformáciu SQL odpovede sa používa SQL formatter, ktorý je schopný SQL dotaz normalizovať. Následne sa upraví reťazec rovnako ako textová odpoveď.
- **Relačná algebra** – Dotaz v relačnej algebre sa najskôr pošle do transformátora relačnej algebry (RAT), ktorý ho transformuje na dotaz v SQL. Ten sa ďalej upraví rovnako ako SQL.
- **Transformácia** – Odpoveď typu transformácia je reťazec vo formáte HTML. Transformuje sa pomocou nástroja Tralex, ktorý vráti normalizovaný objekt.

Po tomto kroku má teda každá uložená odpoveď svoju transformovanú hodnotu, ktorá maximalizuje zhodu s ostatnými odpoveďami bez použitia zložitejších algoritmov na porovnávanie. Problém však je, že tento stĺpec je databázového typu `text`. Je však veľmi neefektívne dotazovať sa s použitím porovnania hodnôt, ktoré majú teoreticky neobmedzenú dĺžku. Z toho dôvodu sa v tabuľke *answer* pridal stĺpec *hash*.

Po transformácii odpovede sa daná hodnota zahashuje pomocou algoritmu xxHash. Ide o hashovací algoritmus, ktorý nemá kryptografické vlastnosti. Disponuje však vysokou mierou disperzie, nízkou kolíznosťou a je veľmi rýchly v porovnaní s inými hashovacími algoritmami. [27] Stĺpec *hash* má teda pri použití varianty XXH128 fixnú dĺžku 32 znakov a databáza s ním vie pracovať oveľa optimálnejšie. Pre lepšiu efektivitu vyhľadávania bude nad týmto stĺpcom vytvorený index, ktorý dotazovanie pri veľkom množstve dát ešte zrýchli.

3.3 Automatická oprava

Algoritmy na opravu odpovedí sú z veľkej časti prevzaté z predchádzajúcej implementácie DBS portálu. Tie sú podrobnejšie popísane v analýze v sekcii 2.2.1. V tomto ohľade je teda potrebné kód zrefaktorovať, aby využíval princípy OOP a bol jednoducho rozšíriteľný a upraviteľný.

Na to je, podobne ako pri tvorbe odpovedí, vhodný návrhový vzor *Strategy*. Diagram tried bude takmer identický s diagramom na obrázku 3.2. Pre každý typ odpovede bude vytvorená trieda, ktorá bude dediť zo zákadnej stratégie. To umožní jednoduché pridanie nového typu odpovede. Jediné, čo bude potrebné spraviť je vytvoriť novú triedu, ktorá bude tiež dediť zo základnej stratégie a naimplementovať potrebný algoritmus.

3.3.1 Možnosti vylepšenia algoritmov

Vyhodnocovanie odpovedí typu radio button alebo normalizácia dosahuje maximálnu efektivitu. Pri oboch z nich je systém schopný vyhodnotiť ich správnosť plne automaticky. To však pri

ostatných typoch neplatí a pri niektorých z nich je priestor na zlepšenie. Táto sekcia obsahuje návrhy, ktoré by mohli opravovanie jednotlivých typov odpovedí zefektívniť.

3.3.1.1 Checkbox

Vyhodnocovanie odpovedí typu checkbox by bolo možné plne zautomatizovať zavedením určitých pravidiel. Existuje viacero možností, ktoré sa dajú aplikovať, tu je niekoľko z nich.

- **Úplná zhoda** – V tejto metóde študent môže získať buď plný počet bodov pri vybratí všetkých odpovedí zhodne s referenčnou odpoveďou alebo 0 bodov, ak je v odpovedi chyba.

- **Čiastkové body** – Do úvahy pri hodnotení by sa brali len zvolené správne možnosti. Počet bodov za odpoved by sa vypočítal ako

$$\frac{počet\ zvolených\ správnych}{počet\ všetkých\ správnych} \cdot maximálny\ počet\ bodov.$$

- **Správne vs. nesprávne možnosti** – Do úvahy by sa brali ako zvolené správne, tak aj nesprávne možnosti. Výpočet počtu bodov by bol nasledovný:

$$\frac{počet\ zvolených\ správnych - počet\ zvolených\ nesprávnych}{počet\ všetkých\ možností} \cdot maximálny\ počet\ bodov.$$

- **Penalizácia nesprávnych možností** – Z celkového počtu bodov za otázku by sa za každú nesprávne zvolenú možnosť odpočítalo určité percento z celkového hodnotenia. Ak by to bolo napríklad 50 %, znamenalo by to, že študent môže spraviť jednu chybu za polovičný počet bodov a pri 2 chybách by bol počet bodov 0. Toto percento by sa dalo dynamicky nastaviť podľa počtu možností.

3.3.1.2 Transformácia

Oprava transformácie aktuálne prebieha spracovaním HTML reťazca do formy štandardizovaného JSON objektu pomocou nástroja Tralex. Tento objekt sa následne pri oprave porovnáva s referenčnou odpoveďou podľa zadaných kritérií.

Pri porovnávaní sa môže naďalej využívať algoritmus prevzatý zo starého portálu, avšak je ho potrebné zrefaktorovať. Ten prejde dvojicu objektov a na základe výsledkov porovnania jednotlivých vlastností objektu vráti počty chýb v rôznych kategóriách. Každá kategória má nastavenú váhu a pomocou nich sa vypočíta výsledná penalizácia odpovede. Táto penalizácia sa ďalej používa na vytvorenie hodnotenia typu *suggestion*, ktoré sa zobrazí učiteľovi pri manuálnej oprave. Keďže v mikroslužbe sa hodnotenie vytvára na backende aj keď ide len o jeho návrh pre učiteľa, z frontendu je možné odstrániť jeho dopočítavanie a zjednodušiť tak jeho logiku.

Ak zhoda nie je 100 %, odpoveď sa ďalej bude porovnávať s odpoveďami iných študentov pomocou hashu. Problémom však je, že Tralex, pri transformácii z HTML do JSON objektu nezoraďuje entity, atribúty ani relácie podľa žiadnych kritérií. To znamená, že ak študent vo svojej odpovedi zadá najskôr entitu B a potom entitu A, v objekte bude toto poradie zachované. Tým pádom sa nebude táto odpoveď považovať za identickú s odpoveďou iného študenta, ktorý zadal poradie týchto entít inak. Aby bolo možné využívať plný potenciál hashu, je potrebné, aby boli všetky dané veci zoradené podľa pevných kritérií. Vďaka tomu by poradie v objekte viac nebolo závislé od poradia, ktoré zadal študent v odpovedi. To by zasa zvýšilo počet zhôd a znížilo počet odpovedí, ktoré je potrebné manuálne opraviť.

3.3.1.3 SQL, RA

Oba tieto typy odpovedí majú identický spôsob vyhodnocovania, keďže ide o dotazy do relačnej databázy. V mikroslužbe *Connections* bola pridaná databázová funkcia, ktorú napísal Ing. Michal Valenta, Ph.D. a slúži na porovnávanie výsledkov 2 dotazov. Porovnávajú sa v nej nasledujúce vlastnosti výsledkov:

- Chyby v syntaxi
- Rovnaký počet stĺpcov
- Rovnaké názvy stĺpcov
- Rovnaký počet riadkov
- Rovnaké poradie riadkov
- Rovnaké dáta (porovnáva hodnoty v jednotlivých riadkoch)

Daná funkcia však zatiaľ nepodporuje porovnávanie názvov stĺpcov, ak sú v inom poradí, čo je potrebné doimplementovať.

Ďalším miestom, kde sa funkcia môže vylepšiť, je schopnosť porovnávať dáta nezávisle od názvov stĺpcov. Fungovať by to mohlo porovnávaním dátových typov hodnôt v jednotlivých stĺpcoch. Na základe tohto porovnania by sa vytvorilo mapovanie stĺpcov z výsledkov jedného dotazu do výsledkov druhého dotazu. S využitím tohto mapovania by sa ďalej porovnal zvyšok dát. Toto by umožnilo vyhodnotiť dotazy ako správne napriek nesprávnemu názvu stĺpca, čo vo väčšine prípadov nie je zásadná chyba, ktorá by mala byť penalizovaná.

3.3.2 Asynchrónne vyhodnocovanie

Automatická oprava sa spúšťa ihneď po odoslaní testu študentom alebo po uplynutí času na test. O spustenie opravy sa bude starať mikroslužba na priebeh testov, ktorá pošle požiadavku na opravu študentovho testu. V tom momente sa začne oprava jednotlivých odpovedí študenta z daného testu.

Keďže aktuálny systém pri opravovaní podliehal veľmi veľkej záťaži, je potrebné, aby sa opravovanie zoptimalizovalo a nebolo závislé na počte odoslaných odpovedí. Na to je potrebné nasadiť mechanizmus, ktorý bude schopný túto záťaž regulovať. To bude fungovať pomocou *Message queue*. Základný princíp fungovania je popísaný v 1.4.1.

Po prijatí požiadavky sa teda vytvorí jedna správa na každú odpoveď v teste a pošle sa do queue. Táto správa bude obsahovať minimálne informácie, ktoré sú potrebné a tým je ID danej odpovede. Následne si konzument vyzdvihne správu a spustí sa algoritmus potrebný na automatickú opravu odpovede. Po dokončení opravy môžu nastať 3 prípady.

1. Odpoveď bola ohodnotená automaticky
2. Odpověd nebolo možné ohodnotiť automaticky
3. Odpoveď nebolo možné ohodnotiť automaticky, ale bolo navrhnuté hodnotenie

Ak nastal prvý prípad, hodnotenie danej odpovede je ukončené a môže sa prejsť na spracovanie ďalšej. Ak však nastal druhý alebo tretí prípad, odpoveď je nutné poslať na manuálnu opravu.

3.3.3 Synchronizácia hodnotení

Pre zabezpečenie spravodlivého hodnotenia je dôležité, aby odpovede, ktoré sú navzájom ekvivalentné mali aj rovnaké hodnotenie. To znamená, že vždy, keď sa upraví hodnotenie jednej z odpovedí, musia byť upravené aj všetky ostatné z rovnakej skupiny ekvivalencie. Na to je potrebné použiť mechanizmus, ktorý bude výpočetne čo najmenej náročný, ale zároveň bude synchronizovať spoľahlivo.

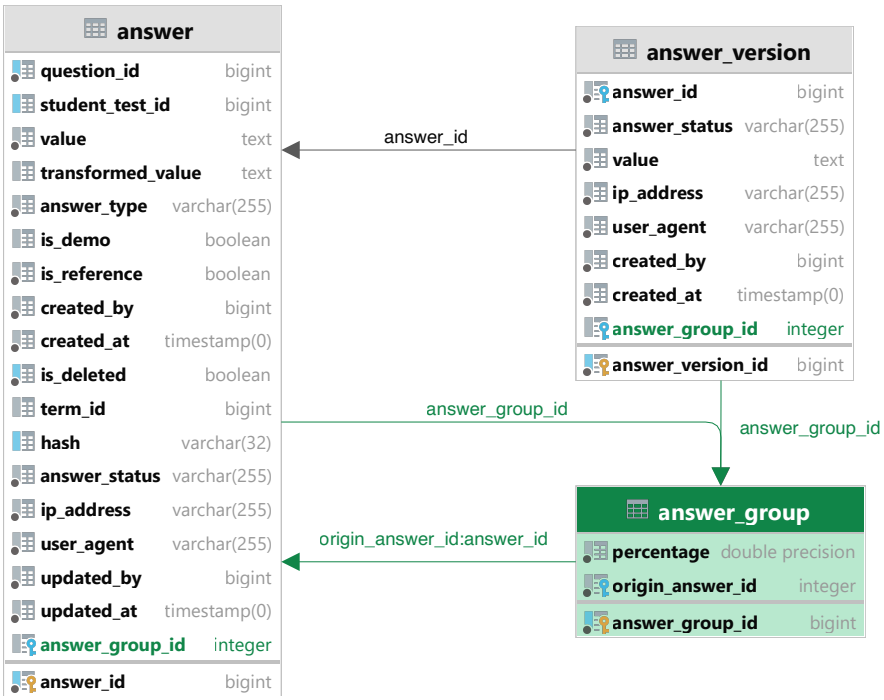
Najjednoduchší mechanizmus je použitie porovnania odpovedí na základe hashu. Pri vytvorení odpovede sa odpoveď normalizuje podľa postupu popísaného v 3.2. Tento spôsob zoskupovania odpovedí je jednoznačne nejrýchlejší, keďže ide o reťazec fixnej dĺžky, na ktorý môže byť aplikovaný index. Množstvo a spoľahlivosť zhôd závisí od kvality normalizácie odpovede.

Spôsob, ktorý by bol síce menej efektívny, avšak dosahoval by vyššieho množstva zhôd je používanie skupín ekvivalencie. V jednej skupine by boli všetky odpovede, ktoré sú vzájomne ekvivalentné. Všetky tieto odpovede by mali mať pre maximálnu spravodlivosť vždy rovnaké hodnotenie. Keďže sa maximálny počet bodov, ktoré je možné získať na jednotlivých otázkach môže v rôznych testových termínoch meniť, hodnotenie by bolo založené na percentách.

Vzhľadom na rozsah celého systému bude v rámci tejto bakalárskej práce naimplementovaný spôsob porovnávania hashov s možnosťou neskoršieho vylepšenia normalizácie odpovedí alebo nahradenia iným mechanizmom. Napriek tomu je pre budúce použitie nižšie podrobnejšie popísané využitie skupín ekvivalencie.

3.3.3.1 Databázová štruktúra

Pre zabezpečenie tvorby skupín ekvivalencie je potrebné do databázy pridať jednu novú tabuľku – *answer_group*. Tabuľka musí obsahovať atribúty *answer_group_id*, *percentage*, *origin_answer_id* a *created_at*. Do tabuľky *answer* sa pridá stĺpec s ID skupiny (*answer_group_id*), do ktorej daná odpoveď patrí. Tento stĺpec sa pridá aj do tabuľky *answer_version* pre zaznamenávanie prípadných zmien. Zmeny je možné vidieť na diagrame na obrázku 3.3.



Obr. 3.3 Zmeny v databázovom diagrame potrebné pre pridanie skupín ekvivalencie

3.3.3.2 Algoritmus

Keďže proces porovnávania odpovedí potrebný pre zaraďovanie odpovedí do skupín ekvivalencie je výpočetne náročnejší ako porovnávanie hashov, je dobré ho vykonávať asynchrónne. Vyhodnocovať ekvivalenciu je možné buď okamžite po vytvorení alebo upravení odpovede alebo na konci po odovzdaní testu. Obe riešenia majú svoje pre aj proti. Ak sa bude vyhodnocovať po ukončení testu, oprava sa nebude môcť vykonať okamžite, ale až po zaradení všetkých odpovedí do príslušných skupín. V prípade vyhodnocovania priebežne pri každom vytvorení alebo zmene budú v dobe ukončenia testu všetky odpovede už zaradené. Nevýhodou však je, že ak sa odpoveď mení veľakrát, bude to neefektívne.

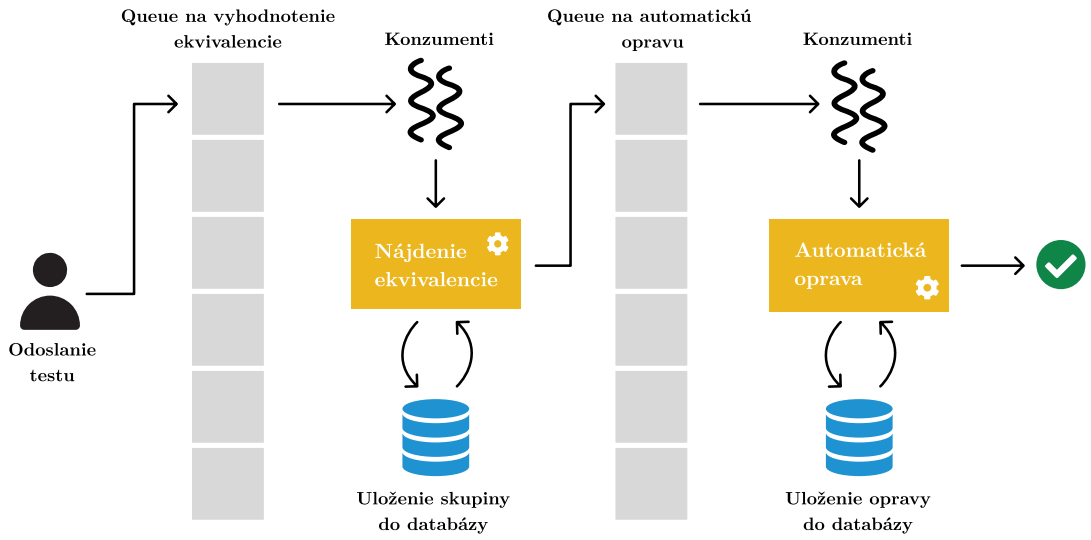
Pre názornosť sa bude v ďalšej časti pracovať s možnosťou vyhodnotenia ekvivalencie až po ukončení testu.

Odpovede budú vkladané do 2 FIFO štruktúr, ktoré budú zreťazené za sebou. Prvá bude slúžiť na odpovede čakajúce na hľadanie skupín ekvivalencie a druhá na samotné automatické vyhodnocovanie s ohľadom na skupinu, do ktorej odpoveď patrí.

- **Nájdenie ekvivalencie** – Proces hľadania skupiny ekvivalencie bude využívať rôzne algoritmy v závislosti od typu odpovede. V prvom rade sa bude odpoveď porovnávať s odpoveďami, ktoré už sú zaradené v skupinách. Z každej skupiny sa porovná iba s jednou odpoveďou, ktorá do nej patrí. V prípade, že žiadna z už vytvorených ekvivalencií nie je vhodná, pre danú odpoveď sa vytvorí nová skupina a odpoveď sa posunie na vyhodnotenie.
- **Automatické hodnotenie** – Proces automatického hodnotenia už bol popísaný v sekcii 3.3. Tento postup však bude aplikovaný iba v prípade, že odpovede v jej skupine ekvivalencie ešte nemajú nastavené hodnotenie. Ak ho nastavené majú, odpovedi sa nastaví identické a proces hodnotenia je ukončený.

Hlavnou výhodou tohto riešenia je následné synchronizovanie hodnotenia v prípade jeho zmeny. Ak učiteľ manuálne ohodnotí odpoveď iným počtom bodov, nájdenie odpovedí, na ktorých je potrebné hodnotenie opraviť je omnoho jednoduchšie a spoľahlivejšie. Rovnako to je, ak odpoveď nie je možné vyhodnotiť automaticky. V momente, kedy ju učiteľ ohodnotí sa rovnaké hodnotenie nastaví všetkým ekvivalentným odpovediam. Okrem toho poskytuje lepší prehľad o počte odpovedí, ktoré sa vyhodnotia automaticky.

Celý proces po ukončení testu je možné vidieť na obrázku 3.4.



Obr. 3.4 Proces automatického spracovania odpovede v ukončenom teste

Kapitola 4
Realizácia

V tejto kapitole bude popísaný postup implementácie mikroslužby, metodiky vývoja a budú tu podrobnejšie rozobraté zaujímavé problémy, na ktoré sa narazilo počas vývoja.

4.1 Príprava na vývoj

4.1.1 Vývojové prostredie

Všetky doposiaľ naimplementované mikroslužby sú kontajnerizované v dockeri. Každá mikroslužba má svoj vlastný kontajner, ktorý vychádza z oficiálneho imagu pre nginx. Celý kód mikroslužieb je implementovaný v jazyku PHP (verzia 8.1) a frameworku Symfony (verzia 6.0). Čo sa týka databázy, využíva sa *PostgreSQL* (verzia 14) a objektovo-relačné mapovanie je spravené pomocou knižnice Doctrine.

4.1.2 Tvorba novej mikroslužby

Proces vytvorenia mikroslužby bol vcelku jednoduchý. V *docker-compose.yaml* bolo potrebné vytvoriť nový záznam v sekcii `services` a kontajner následne spustiť pomocou príkazu `docker-compose up -d`.

```
test_evaluations:
  image: "gitlab.fit.cvut.cz:5000/dbs/dbs-microservices
          /nginx-unit-test-evaluations:php8.1-1.1.0"
  hostname: "test_evaluations"
  environment:
    APP_TEMPLATE: "symfony-dev"
  depends_on:
    - postgres-db
    - redis
    - auth
  volumes:
    - ./TestEvaluations:/var/www/public
  networks:
    - internal
  extra_hosts: # just for dev
    - host.docker.internal:host-gateway
  labels:
    cz.cvut.fit.dbs.microservice: "Test evaluations"
```

```
    traefik.enable: true

    traefik.http.routers.test_evaluations.rule:
        "PathPrefix(‘/test-evaluations‘)"

    traefik.http.routers.test_evaluations.entrypoints:
        "web"

    traefik.http.services.test_evaluations.loadbalancer.server.port:
        "9000"
```

Výpis kódu 4.1 Konfigurácia mikroslužby v docker-compose.yaml

Po spustení kontajnera sa spustila migrácia, ktorá vytvorila v databáze potrebné tabuľky, ktoré mikroslužba využíva a príprava na vývoj bola hotová.

4.1.3 Metodika vývoja

Vymyslieť dobrý softvérový návrh a rovno ho naimplementovať na prvý pokus je takmer nemožné. Systém a jeho procesy sú často komplikovanejšie ako sa na prvý pohľad, ale aj po podrobnejšej analýze, zdá. Mnohokrát je kontraproduktívne snažiť sa napísať všetko správne na prvý pokus. Z toho dôvodu bol zvolený iteratívny prístup k vývoju. Jeho špecifiká sú popísané v 1.1.2.

Vývoj prebiehal v niekoľkých iteráciach, pričom v každej z nich sa buď pridala nová funkcionalita alebo sa vylepšila už existujúca. Tu je popis hlavných iterácií.

1. **Kostra mikroslužby** – Kľúčovým tu bolo vytvorenie mapovania entít do tabuliek pomocou ORM, v rámci čoho boli vytvorené entity a repozitár pre každú tabuľku. Následne boli otestované vytvorením skúšobného kontroléra a služby. Tým sa vyskúšalo, že funguje správne smerovanie požiadaviek do kontroléra, prepojenie medzi vrstvami aplikácie, ako aj mapovanie do databázy.

2. **CRUD operácie** – Boli tu vytvorené kontroléry a služby, ktoré podporovali CRUD operácie na všetky entity bez akejkoľvek ďalšej logiky. To umožnilo, aby mohla byť mikroslužba volaná z frontendu.

3. **Tvorba odpovedí** – Rôzne typy odpovedí vyžadovali rôzny prístup k ich tvorbe. V rámci tejto iterácie bola entita `Answer` rozdelená na podentity pre každý typ odpovede. Následne boli naimplementované triedy, ktoré spracovávajú odpovede pri ich tvorbe. Ich zjednodušený diagram je možné vidieť na obrázku 3.2.

4. **Hodnotenie odpovedí** – Ako k tvorbe rôznych typov odpovedí, aj k ich oprave je potrebné pristupovať v závislosti od ich typu. Boli vytvorené triedy, ktoré obsahujú stratégie na hodnotenie pre každý typ odpovede.

5. **Verzovanie odpovedí** – Pridaná podpora pre rozlišovanie automatického a manuálneho odoslania odpovede a možnosť vrátiť sa k skôr odoslanej odpovedi v prípade potreby.

6. **Prepojenie s ostatnými mikroslužbami** – Keďže mikroslužba je dátami závislá na mikroslužbách Tests a Assignments, bolo potrebné naimplementovať komunikáciu s nimi. Jadro tejto komunikácie naimplementoval Bc. Radoslav Hašek v rámci svojej diplomovej práce, tento kód bol prevzatý z jeho mikroslužby a upravený pre konkrétne potreby mikroslužby Test Evaluations.

7. **Testovanie** – V rámci jednotkových testov bolo otestované správne fungovanie kľúčových častí systému – tvorby a autoamtického hodnotenia odpovedí. Popri tom boli otestované aj komponenty, ktoré sú v týchto dvoch častiach systému využívané.

Pomedzi ne boli zapracovávané rôzne úpravy kódu na požiadavky frontendu alebo na základe code review a funkcionality boli užívateľsky testované.

Voľba iteratívneho vývoja bola určite dobrým krokom, pretože počas vývoja sa objavovali nové výzvy, na ktoré bolo potrebné pružne reagovať. Vzhľadom na komplexnosť systému boli požiadavky v priebehu celého procesu upresňované a upravované. Niekoľkokrát prebehli aj konzultácie s ľuďmi, ktorí sa podieľajú na chode DBS portálu pre validáciu návrhu.

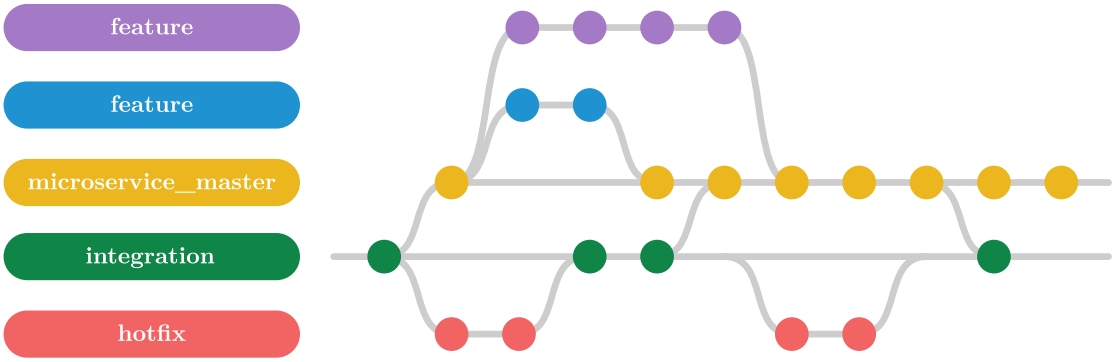
4.1.3.1 Git workflow

Keďže vývoj mikroslužby Test Evaluations prebiehal paralelne s vývojom mikroslužieb Assignments, Tests a Test Templates, bolo potrebné vymyslieť správny Git workflow, ktorý sa bude využívať naprieč týmito mikroslužbami.[1] Spolu s Bc. Radoslavom Haškom a Tomášom Doubom bolo dohodnuté riešenie, ktoré pre dané potreby funguje najlepšie.

Hlavnou vetvou testového modulu je vetva `test_module_integration`. Tá sa synchronizuje s `master` vetvou celého projektu. Každá z mikroslužieb, ktorá sa podieľa na testovom module má vlastnú master vetvu (`test_evaluations_master`, `tests_master`, `test_templates_master`, `assignments_master`). V rámci mikroslužby sa funguje podľa *Feature branch workflow*. Jeho podstatou je, že všetky funkcionality by sa mali vyvíjať v samostatných vetvách a až po ich dokončení sa zlúčia s master vetvou.

Následne sa tieto funkcionality dostanú do integračnej vetvy. Pre zlúčenie vetiev je nevyhnutné, aby kód prešiel cez *code review*, ktoré vykoná vždy aspoň jeden z vývojárov pracujúcich na testovom module. To zabezpečuje udržanie kvality napísaného kódu a taktiež napomáha udržiavať kód konzistentným naprieč mikroslužbami.

V prípade, že sa v integračnej vetve ocitne kód, ktorý obsahuje chybu, vytvorí sa *hotfix* vetva, v ktorej sa daný kód opraví a zlúči sa s integračnou vetvou. Keďže ide spravidla o malé zmeny, je možné ich dostať do integračnej vetvy takmer okamžite.



Obr. 4.1 Git workflow používaný pri vývoji testového modulu

4.2 Verzovanie odpovedí

Plánom do budúcna je zautomatizovať ukladanie odpovedí pre prípad, že študent zabudne odoslať odpoveď alebo nastane nejaká chyba počas testu. Frontend bude v pravidelných intervaloch posielať rozpracovanú odpoveď. Bol tomu teda prispôsobený aj backend. Do tela požiadavky na vytvorenie odpovede bol pridaný atribút *status*, ktorý môže byť buď *auto_save* alebo *manual_save*. Pri hodnotení odpovede sa bude brať do úvahy len odpoveď, ktorá má zadaný status

[1] Prístupy, ktoré sa dajú využiť pri používaní Gitu k dosiahnutiu maximálnej konzistencie a produktivity.

manual_save, avšak k automaticky uloženým odpovediam by mali študenti prístup počas testu a môžu sa k nim kedykoľvek vrátiť.

Verzovanie odpovede je riešené priamo v databáze pomocou triggeru. Každá zmena v tabuľke *answer* je automaticky zaznamenaná do tabuľky *answer_version*. Na pozadí sa každý záznam spracuje podľa definovaných pravidiel tak, aby v tabuľke *answer* ostala vždy najrelevantnejšia odpoveď. V nasledujúcej tabuľke sú spísané všetky prípady, ktoré môžu nastať pri zmene odpovede a k nim aj akcia, ktorú vykoná databázový trigger.

	Aktuálny status	**Nový status**	**Akcia**
1	**auto_save**	**auto_save**	Aktuálna odpoveď sa v tabuľke *answer* nahradí novou odpoveďou
2	**auto_save**	**manual_save**	Aktuálna odpoveď sa v tabuľke *answer* nahradí novou odpoveďou
3	**manual_save**	**manual_save**	Aktuálna odpoveď sa v tabuľke *answer* nahradí novou odpoveďou
4	**manual_save**	**auto_save**	Aktuálna odpoveď v tabuľke *answer* ostane, avšak v tabuľke answer_version sa vytvorí nový záznam o odpovedi typu *auto_save*. Tým sa zabezpečí, že manuálne odoslaná odpoveď nebude nikdy nahradená automaticky uloženou odpoveďou.

Tabuľka 4.1 Kombinácie statusov, ktoré môžu nastať pri verzovaní odpovedí

Pri získavaní odpovede študenta z mikroslužby sa spolu s odpoveďou posielajú automaticky aj všetky jej verzie. Objekt v tele odpovede vyzerá nasledovne.

```
{
  "answer_id": 1,
  "value": "Upraveny text odpovede",
  "answer_type": "text",
  "status": "manual_save",
  "is_demo": false,
  "question_id": 1,
  "student_test_id": 1,
  "created_at": "2023-05-03 12:41:07",
  "created_by": 123,
  versions: [
    {
      "status": "auto_save",
      "value": "Upraveny text odpovede",
      "created_at": "2023-05-03 12:45:21"
    },
    {
      "status": "manual_save",
      "value": "Text odpovede",
      "created_at": "2023-05-03 12:41:07"
    }
  ]
}
```

Výpis kódu 4.2 Vzorová odpoveď mikroslužby pri získavaní entity answer

4.3 Automatizácia záznamov v databáze

Ako je možné vidieť v návrhu databázy (3.1.1), každá tabuľka má určité stĺpce, ktoré nie je nutné nastavovať manuálne vždy pri vytorení alebo úprave záznamu. Príkladom sú napríklad stĺpce, ktoré zaznamenávajú čas vytvorenia entity, jej úpravy alebo ID používateľa, ktorý danú zmenu vykonal.

Iný prípad, kedy sa dá vypĺňanie hodnôt v databáze automatizovať je mazanie entít pomocou tzv. soft delete. Ide o mazanie, pri ktorom v databáze záznamy ostávajú uložené napriek ich vymazaniu. Entity, ktoré to povoľujú majú príznak, ktorý sa nastaví na *vymazané* a na backende sa takéto entity nebudú z databázy vôbec vyťahovať.

Pri každom vytvorení alebo úprave entity a jej následnom uložení spúšťa Doctrine udalosť, na ktorú je možné čakať a v momente jej spustenia vykonať preddefinovaný kód. Mechanizmus, ktorý sa na to dá použiť sa nazýva *Lifecycle subscriber*. Ide o triedu, ktorá môže obsahovať metódy ako `prePersist`, `postPersist`, `preUpdate` či `postUpdate`. Každý *subscriber* je potrebné najskôr zaregistrovať v konfiguračnom súbore. Následne je potrebné naimplementovať telo metód, ktoré sa bude vykonávať vždy, keď nastane daná udalosť.[28]

V entitách boli identifikované 3 akcie, ktoré by sa dali takýmto spôsobom zautomatizovať.

- **Mazanie entít** (tzv. soft delete)
- **Nastavovanie časového razítka**
- **Nastavovanie autora zmeny**

Pre entity teda boli vytvorené 3 rozhrania – `SoftDeletable`, `Timestampable` a `Blamable`, ktoré obsahovali potrebné metódy na nastavenie daných hodnôt (setters) a ich získavanie (getters). Následne boli vytvorené a zaregistrované 3 triedy, ktoré budú dané udalosti spracovávať – `SoftDeleteSubscriber`, `TimestampSubscriber` a `BlameSubscriber`.

Príklad triedy SoftDeleteSubscriber je možné vidieť vo výpise kódu 4.6. Podobným štýlom sú naimplementované aj zvyšné 2 triedy.

```
/**
 * Class takes care of soft deleting the entity
 * and prevents it from being deleted by Doctrine
 */
class SoftDeleteSubscriber implements EventSubscriber
{
  public function getSubscribedEvents(): array
  {
    return [
      Events::preRemove,
    ];
  }

  public function preRemove(PreRemoveEventArgs $args): void
  {
    $entity = $args->getObject();
    if (!$entity instanceof SoftDeletable) {
      return;
    }

    // Instead of deleting the entity, just set isDeleted to true
    $entity->setIsDeleted(true);

    $em = $args->getObjectManager();
    $em->persist($entity);
```

```
    $em->flush();

    // Prevent the entity from being deleted by Doctrine
    $em->getUnitOfWork()->detach($entity);
  }
}
```

Výpis kódu 4.3 Trieda SoftDeleteSubscriber

4.4 Hodnotenie odpovedí

Ako bolo popísané v návrhu v sekcii 3.3, automatická oprava využíva návrhový vzor *Strategy*. To znamená, že každý typ odpovede má svoju stratégiu hodnotenia a konkrétna implementácia sa vyberá až za behu podľa typu odpovede.

Kód v metóde `evaluate` bude následne veľmi jednoduchý a zahŕňa 2 hlavné operácie – získanie správneho evaluátora a následné spustenie hodnotenia zo získaného evaluátora.

Zjednodušená implementácia metódy `evaluate()` je v ukážke 4.4

```
// get correct evaluator based on answer type
$evaluator = $this->getEvaluator($answer->getAnswerType());

// run evaluation
$evaluation = $evaluator->evaluate($answer)
```

Výpis kódu 4.4 Ukážka kódu z metódy evaluate v triede AnswerEvaluator

Získanie správneho evaluátora spočíva v jednoduchom porovnaní typu odpovede, ktorý bol poslaný do metódy v argumente. Návratová hodnota metódy je typu `BaseEvaluator`, z ktorého dedia všetky ostatné evaluátory. Vďaka tomu je následné volanie metódy evaluate polymorfné.

```
private function getEvaluator(AnswerType $answerType): BaseEvaluator
{
  return match ($answerType) {
    AnswerType::TEXT => $this->textEvaluator,
    AnswerType::CHECKBOX => $this->checkboxEvaluator,
    AnswerType::RADIO => $this->radioEvaluator,
    AnswerType::SQL => $this->sqlEvaluator,
    AnswerType::RA => $this->raEvaluator,
    AnswerType::NORMALIZATION => $this->normalizationEvaluator,
    AnswerType::DIAGRAM => $this->diagramEvaluator,
    AnswerType::TRANSFORMATION => $this->transformationEvaluator,
    default => throw new InvalidArgumentException('Invalid answer type')
  };
}
```

Výpis kódu 4.5 Metóda getEvaluator v triede AnswerEvaluator

Rozšírenie o prípadné ďalšie typy odpovedí bude opäť veľmi priamočiare. Je potrebné vytvoriť implementáciu samotného evaluátora pre nový typ odpovede a do metódy `getEvaluator` je potrebné doplniť prípad, v ktorom sa namapuje nový typ odpovede na nový evaluátor. Následne bude systém plne podporovať jej automatickú opravu.

4.5 Asynchrónna automatická oprava

Ako je popísané v návrhu v sekcii 3.3.2, vyhodnocovanie odpovedí sa vykonáva s využitím asynchrónnej komunikácie. Bol na to využitý Symfony Messenger, ktorý poskytuje podporu

pre takýto spôsob komunikácie. Pre jeho používanie bolo potrebné vytvoriť 2 triedy – entitu pre správy, ktoré sa budú posielať a triedu, ktorá ich bude spracovávať.

4.5.1 Správy

Správy by mali obsahovať spravidla minimum informácií, aby bol ich prenos pamäťovo nenáročný. Pre opravu odpovede je potrebné poznať iba jej unikátny identifikátor. Bola teda vytvorená trieda `EvaluateAnswerMessage` s jediným atribútom – `answerId`. Pri vytvorení správy sa do nej vloží ID odpovede a správa sa vloží do queue, kde ostane až do jej spracovania.

4.5.2 Spracovanie správ

Keďže správy je potrebné spracovávať asynchrónne, je nevyhnutné spustiť konzumentov, ktorí budú na správy čakať. V *.env* súbore bola vytvorená premenná prostredia, ktorá definuje počet konzumentov, ktorí sa majú spustiť. Ďalej bol vytvorený bash skript, ktorý ich na základe danej premennej spustí na pozadí. Keďže budú bežať na pozadí, je potrebné zabezpečiť, aby bol ich výstup niekde zaznamenávaný. Zvolil sa spôsob logovania do súboru. Každý konzument bude mať vytvorený vlastný súbor, do ktorého je presmerovaný jeho štandardný aj chybový výstup.

```
#!/bin/bash

source .env

echo "Starting $MESSENGER_CONSUMERS_COUNT messenger consumer(s)..."

# create log directory if it doesn't exist
if [ ! -d "log" ]; then
  mkdir -p "log"
fi

# start consumers
for i in $(seq 1 $MESSENGER_CONSUMERS_COUNT)
do
    echo "Starting messenger consumer #$i..."
    nohup php bin/console messenger:consume async -vv
      > log/messenger-consume-output-$i.log 2>&1 &
done
```

Výpis kódu 4.6 Skript na spustenie konzumentov

4.6 API Dokumentácia

Postup tvorby doterajších mikroslužieb zahŕňal okrem databázového návrhu a tvorby rôznych diagramov, ktoré špecifikovali fungovanie mikroslužby aj tvorbu API dokumentácie v nástroji Swagger. Táto dokumentácia sa píše do súboru typu yaml. V ňom sa definujú jednotlivé endpointy, ich parametre, telo požiadaviek aj všetky návratové hodnoty.

4.6.1 Automatické generovanie

Proces manuálneho písania dokumentácie je relatívne pracný, keďže je potrebné zoznámiť sa so syntaxou yaml súborov, ako aj s potrebnými typmi a hodnotami, ktoré Swagger podporuje.

Ďalšou nevýhodou je, že všetky nadefinované triedy a návratové typy je potrebné manuálne zdokumentovať napriek tomu, že v kóde sú presne zadefinované.

V minulosti bolo v DBS portáli písanie dokumentácie čisto ručná záležitosť. Pri návrhu mikroslužby sa manuálne napísal OpenAPI yaml súbor, ktorý sa dal zobraziť priamo v gitlabe v prehľadnom UI. Z dlhodobého hľadiska však tento spôsob nie je udržateľný, keďže pri každej úprave API by bolo potrebné ručne zapracovať zmeny aj v konfiguračnom súbore pre API dokumentáciu. Z dôvodu pohodlnejšieho vývoja a lepšej presnosti dokumentácie padlo rozhodnutie preskúmať možnosti automatického generovania dokumentácie z kódu na základe vytvorených kontrolérov (controllers) a entít. Rozsah možností však nie je veľký. Rozhodlo sa vyskúšať bundle, ktorý je odporúčaný priamo v Symfony dokumentácii – *NelmioApiDocBundle*.

Bundle podporuje generovanie JSON objektu, ktorý sa dá priamo transformovať na konfiguráciu vo formáte yaml. Okrem toho má aj užívateľsky prívetivé UI, v ktorom je vidieť všetky endpointy a modely, ktoré sa používajú a je ich možné priamo aj volať. Pre používanie UI je potrebné, aby sa nainštalovali okrem *NelmioApiDocBundle* aj ďalšie 2 bundly - *Twig* a *Asset*.

Po ich inštalácii boli použité vhodné anotácie na oanotovanie všetkých endpointov v kontroléroch, ako aj atribútov vo všetkých entitách. Bundle bol používaný až do pokročilého štádia vývoja, avšak nakoniec bol z kódu odstránený.

Hlavným dôvodom bola neprehľadnosť kódu. Anotácie potrebné na vygenerovanie kvalitnej dokuemntácie, ktorá obsahovala všetky potrebné informácie zväčšovali kód niekoľkonásobne. Napríklad `StudentAnswerController` mal bez použitia anotácii pre generovanie dokumentácie 177 riadkov. Po ich doplnení mal kód riadkov 556. Väčšina kódu teda bola tvorená anotáciami, ktoré boli navyše duplikované naprieč všetkými endpointami. Kód sa teda stal veľmi neprehľadným a tým aj ťažšie udržiavateľným. Vo výpise 4.7 je možné vidieť príklad anotácií pri jednom z endpointov, pričom ide o jedny z kratších.

```
#[Security(name: 'bearerAuth')]
#[OA\Patch(
  path: "/test-evaluations/student-answers/{answerId}",
  operationId: 'updateStudentAnswer',
  summary: 'Updates a student answer.',
  requestBody: new OA\RequestBody(
    description: 'Update student answer',
    required: true,
    content: new OA\JsonContent(
      ref: new Model(type: UpdateAnswerRequest::class)
    )
  ),
  parameters: [
    new OA\Parameter(
      name: 'answerId',
      description: 'Answer identifier',
      in: 'path',
      required: true,
      schema: new OA\Schema(type: 'integer'),
    ),
  ],
  responses: [
    new OA\Response(
      response: 204,
      description: 'Answer was updated successfully.',
    ),
    new OA\Response(
      response: 400,
      description: 'Invalid request body.',
      content: new OA\JsonContent(
```

```
        type: 'array',
        items: new OA\Items(
          ref: '#/components/schemas/BadRequest'
        )
      ),
    ),
    new OA\Response(
      response: 401,
      description: 'Unauthorized.',
    ),
    new OA\Response(
      response: 403,
      description: 'Access denied - user does not have required scope.',
    ),
    new OA\Response(
      response: 404,
      description: 'Answer ID cannot be found.',
    ),
  ],
)]
```

Výpis kódu 4.7 Príklad potrebných anotácií pre generovanie dokumentácie endpointu

Pre pohodlnejšie používanie generovania dokumentácie by bolo potrebné tento bundle rozšíriť o funkcionality, ktoré by umožnili odstrániť duplicity v kóde, čo by bolo mimo rozsah tejto bakalárskej práce. Bundle bol teda nakoniec z kódu odstránený v rámci jedného commitu. Ak by sa teda v budúcnosti rozhodlo doimplementovať funkcionality, bude možné vrátiť všetky anotácie jednoducho späť.

Keďže všetky endpointy boli už oanotované, vygenerovaná dokumentácia bola extrahovaná z JSON objektu, upravená podľa potrieb a uložená do súboru `test_evaluations_openapi.yaml`. Ďalšie zmeny v dokumentácii sa budú ďalej zapracovávať priamo do tohto yaml súboru.

4.6.2 Členenie

Ako mikroslužba na hodnotenie, aj jej endpointy v dokumentácii sú členené do 4 hlavných častí – *Student answers*, *Reference answers*, *Evaluations* a *Notes*. Každá z nich obsahuje okrem definície URI endpointu aj jeho stručný popis, obsah tela requestu spolu s jeho príkladom, ako aj všetky možné návratové hodnoty.

4.7 Testovanie

Táto sekcia sa zaoberá procesom testovania mikroslužby. Stručne budú popísané spôsoby testovania, ktoré boli využité. Vzhľadom k rozsahu mikroslužby bolo z časových dôvodov možné vytvoriť regresné testy iba na niektoré kľúčové časti systému. Podľa ich vzoru je možné v budúcnosti vytvoriť testy aj na ostatné systémové súčasti. Zameranie teda bolo hlavne na manuálne testovanie, ktoré testuje fungovanie systému ako celku a je schopné vytvoriť rýchlejší obraz o chybách v ňom.

4.7.1 Manuálne testovanie

V rámci manuálneho testovania boli vykonané 2 typy testov. Prvým bolo testovanie volaním rozhrania API a kontrolou reálnych výsledkov s očakávanými výsledkami. Druhým je využívanie API pri vývoji frontendu aplikácie.

Testovanie volaním rozhrania bolo vykonávané priebežne po naimplementovaní každej funkcionality. Išlo o tzv. white-box testovanie, ktoré je ideálne na overenie medzných hodnôt a vzhľadom

na dobrú znalosť kódu aj otestovanie všetkých podmienok, ktoré môžu spôsobiť problémy v mikroslužbe. Počas neho bola odchytená väčšina nájdených chýb a celá mikroslužba bola na základe neho odladená. Na konci implementácie bol vykonaný ešte jeden „akceptačný“ test, ktorý zahŕňal volanie všetkých endpointov s rôznymi hraničnými aj povolenými hodnotami. Išlo sa podľa vopred pripraveného scenára, ktorý zahŕňal nasledujúce kroky:

1. Tvorba externých závislostí – otázok a testov v mikroslužbách Assignments, Test Templates a Tests. Bola vytvorená otázka na každý z 8 typov.
2. Vytvorenie možností pre otázky typu checkbox a radio button.
3. Tvorba a úpravy referenčných odpovedí.
4. Tvorba a úpravy odpovedí študentov.
5. Spustenie automatického hodnotenia pre všetky vytvorené testy.
6. Manuálne hodnotenie – prepísanie automatického aj tvorba nového a následná kontrola nastavenia hodnotenia na ekvivalentné odpovede.
7. Tvorba a úpravy poznámok.

V rámci jednotlivých krokov boli overené medzné hodnoty a všetky kontroly, ktoré mikroslužba vykonáva vrátane kontrolovania existencie závislostí v iných mikroslužbách. V tomto testovaní bola odhalená a opravená väčšina z chýb systému, ktoré sa vyskytli počas vývoja.

Popri vývoji mikroslužby bol paralelne vyvíjaný aj frontend na písanie testov z pohľadu študenta, na ktorom pracovala Dana Suchomelová. Mikroslužba sa teda sčasti začala využívať v praxi. Počas spolupráce bolo nájdených niekoľko chýb, ktoré sa nepreukázali pri manuálnom ani automatizovanom testovaní. Všetky z nich boli vzápätí opravené a aktuálne nie je záznam o žiadnych nových problémoch. Okrem toho bude neskôr na mikroslužbu napojený aj frontend na tvorbu a opravu testov z pohľadu učiteľa, na ktorom začal pracovať SP tím v aktuálnom letnom semestri B222.

4.7.2 Automatizované testovanie

Pri automatizovanom testovaní bolo zameranie najmä na regresné testy, ktoré zaručia, že kód funguje správne aj po zmenách v budúcnosti. Projekt je pripravený na využívanie nástroja PHPUnit, ktorý je zameraný na tvorbu jednotkových testov. V tejto časti boli otestované hlavné funkcionality, akými sú tvorba odpovedí, oprava odpovedí, pomocné nástroje na transformáciu a porovnávanie SQL dotazov, textových reťazcov a diagramov.

Keďže sú všetky vytvorené testy jednotkové, vo veľkej miere bolo využívanie mockovanie. Vďaka nemu bolo možné izolovať jednotlivé triedy a otestovať fungovanie samostatných častí.

Celkovo bolo vytvorených 364 jednotkových testov, v rámci ktorých bolo vykonaných 1377 kontrol výsledkov. Tieto testy pokrývajú tie najdôležitejšie časti systému a do budúcna majú slúžiť ako vzorové pre ďalšie testovanie.

Okrem odhalenia chýb v kóde pomohlo jednotkové testovanie odhaliť aj chyby v návrhu jednotlivých tried, ktoré komplikovali proces testovania. Dané zmeny boli zapracované, čím sa zvýšila kvalita celkového dizajnu mikroslužby.

Kapitola 5

Súhrn realizácie a návrh na budúci rozvoj

V tejto kapitole bude zhodnotený celkový priebeh realizácie so zameraním na splnenie funkčných a niektorých nefunkčných požiadaviek, čo poskytne čitateľovi ucelený pohľad na implementáciu aj na testovanie. Taktiež sa v nej diskutuje o tom, čo by bolo potrebné urobiť na zlepšenie projektu a na zabezpečenie jeho dlhodobej udržateľnosti.

5.1 Zhrnutie

Z praktickej časti práce sa podarilo naimplementovať väčšinu plánovaných vecí z návrhu. V tejto časti práce budú postupne prejdené a okomentované všetky funkčnosti z požiadaviek na nový systém zo sekcie 2.3.

Všetok naimplementovaný kód prešiel cez code review v rámci tímu, ktorý pracoval na testovom module a bol najmä manuálne, ale aj automatizovane testovaný.

5.1.1 Rôzne formáty odpovedí

Pri entite odpovede bolo využité dedenie z abstraktnej triedy. Každý typ odpovede má vlastnú triedu, ktorá dedí z abstraktnej triedy `Answer`. V zásade sa líšia iba dátovým typom odpovede. Radio button je celé číslo, checkbox je pole celých čísel, SQL a RA sú text atď. Bol na to využitý typ dedičnosti v databáze nazývaný Single Table Inheritance. Ide o spôsob, v ktorom sa všetky triedy, ktoré dedia z hlavnej triedy namapujú do jednej tabuľky.

V prípade potreby doplnenia nového typu odpovede je úprava jednoduchá – pridá sa nová trieda, ktorá dedí z abstraktnej triedy `Answer`.

5.1.2 Automatické hodnotenie

Pre každý typ odpovede bol naimplementovaný algoritmus na porovnávanie s referenčnou odpoveďou. Algoritmy boli inšpirované aktuálnou implementáciou DBS portálu a niektoré z nich zrefaktorované pre lepšiu čitateľnosť kódu. Medzi ne patrí algoritmus na porovnávanie diagramov a transformácií.

Všetky typy odpovedí (okrem typu normalizácia a radio button) majú aj záložné porovnávanie zhody s už ohodnotenými odpoveďami iných študentov. To slúži na automatické ohodnotenie

alebo navrhnutie hodnotenia v prípade nenájdenia zhody s referenčnou odpoveďou. Spomenuté výnimky tento mechanizmus nepotrebujú, keďže ich vyhodnotenie je plne automatické.

V tejto oblasti je možné do budúcna doplniť algoritmy, ktoré na základe starších hodnotení vypočítajú ideálne ohodnotenie odpovede, ktoré by sa navrhlo učiteľom pri manuálnej oprave. Mikroslužba v tomto stave zahŕňa navrhovanie hodnotenia len na základe jednej identickej odpovede, neberie sa do úvahy žiadna štatistika.

Tak isto by bola veľkým prínosom implementácia skupín ekvivalencie popísaná v 3.3.3, ktorá by zjednodušila opravovanie ekvivalentných odpovedí. Implementácia v rámci tejto bakalárskej práce zahŕňa len jednoduchší spôsob hľadania ekvivalentných odpovedí na základe hashu.

5.1.3 Manuálne hodnotenie

Systém má endpoint ako na manuálne pridanie, tak aj na úpravu už existujúceho hodnotenia odpovede. Vďaka nemu je vždy možné prepísať automaticky ohodnotenú odpoveď.

Pri odosielaní hodnotenia je nutné špecifikovať, či ide o hodnotenie v percentách alebo v bodoch. Nech je zadané ktorékoľvek z toho, backend automaticky dopočíta druhú hodnotu a uloží ju do databázy. Pri tom sa využíva volanie mikroslužby Tests, v ktorej sa nachádza informácia o maximálnom počete bodov za otázku.

Každé hodnotenie, ktoré je vytvorené systémom má nastavený jeden z 3 typov – *reference*, ak bola odpoveď opravená na základe referenčnej odpovede alebo *match*, ak bola odpoveď opravená na základe odpovede iného študenta a *reference*, ak bola odpoveď vytvorená ako odporúčanie pre manuálnu opravu. V prípade vytvorenia hodnotenia počas manuálnej opravy je tento typ nastavený na *manual*.

5.1.4 Verzovanie odpovedí

Verzovanie odpovedí je podporované v rozšírenom móde, kedy systém vie z frontendu prijímať buď odpovede, ktoré vytvoril a odoslal priamo študent, alebo odpovede, ktoré sa posielajú automaticky pre priebežné zálohovanie v prípade akéhokoľvek zlyhania systému. Pri získavaní detailov odpovede je k telu odpovede pripojený atribút *versions*, ktorý obsahuje všetky vytvorené verzie.

5.1.5 Identifikácia podvádzania

Keďže na identifikáciu podvádzania pri mikroslužbe, s ktorou sa komunikuje cez rozhranie API nie je veľa možností, využívajú sa 2 informácie, ktoré sú odosielané v hlavičkách požiadaviek – User-Agent (obsahuje detaily o klientovi, z ktorého bola požiadavka zaslaná) a IP adresa. Na základe nich je možné zistiť, či bol test písaný z jedného zariadenia.

Bol vytvorený endpoint, ktorý na základe starších verzií odpovedí vyhodnotí zhodu týchto 2 údajov naprieč všetkými verziami odpovedí v študentovom teste. Ak sa nezhodujú, je podozrenie, že test bol písaný z viacerých zariadení, čo môže znamenať porušenie pravidiel.

Keďže mikroslužba, ktorá bude zabezpečovať posielanie notifikácii v dobe písania tejto bakalárskej práce ešte nebola naimplementovaná, v prípade podozrenia z podvádzania nie je učiteľ nijako upovedomený. Do budúcna je teda odporúčané túto kontrolu spustiť v momente odoslania testu študentom a v prípade podozrenia notifikovať vyučujúceho.

5.1.6 Diskusia, poznámky

Hodnotenie môže mať priradený ľubovoľný počet poznámok, ktoré môžu byť 2 typov – *evaluation_comment* alebo *discussion*. Pridanie nového typu v prípade potreby je triviálne a vyžaduje iba jeho pridanie do enumu `NoteType`. Pri získavaní poznámok je následne tento typ posielaný pre každú poznámku pre jednoduché rozlíšenie jej typu na frontende.

5.1.7 Škálovateľnosť a vysoká záťaž

Automatické hodnotenie je schopné fungovať paralelne s využitím asynchrónnej komunikácie. V súbore s premennými prostredia je možné nastaviť množstvo konzumentov, ktorí budú odpovede vyhodnocovať a teda tým horizontálne škálovať automatickú opravu odpovedí. Vďaka tomuto mechanizmu nezáleží na počte odovzdaných testov v jeden moment. Všetky odpovede čakajúce na opravu sa vložia do queue a sú postupne spracovávané, čo zaručuje limitovanie záťaže.

5.2 Budúci rozvoj

Na základe výsledkov relizácie bude v tejto sekcii navrhnutý postup pre ďalší rozvoj mikroslužby. Taktiež tu budú spomenuté veci, na ktoré boli mimo rozsah tejto bakalárskej práce, ale sú z hľadiska ďalšieho rozvoja mikroslužby dôležité.

5.2.1 Testovanie

Z dôvodu vysokej časovej náročnosti implementácie a celkového rozsahu mikroslužby bol zvolený prístup dôkladného manuálneho testovania mikroslužby. Tým spôsobom bolo možné pokryť celú funkcionalitu aplikácie a zaručiť tak kvalitu dodaného riešenia. Okrem manuálneho testovania boli vytvorené aj regresné jednotkové testy, ktoré môžu slúžiť ako vzorové pre ďalší vývoj.

Pre dlhodobé bezproblémové fungovanie mikroslužby je podstatné, aby bola čo najväčšia časť kódu pokrytá regresnými automatizovanými testami. Je odporúčané vytvoriť nielen ďalšie jednotkové, ale aj integračné testy. Tie by sa mali zamerať na komunikáciu s ostatnými mikroslužbami, pretože tá je pre ich fungovanie kľúčová.

5.2.2 Asynchrónne vyhodnocovanie

V mikroslužbách, ktoré sú využívané pri automatickom hodnotení je potrebné vytvoriť endpointy, ktoré budú prístupné iba zvnútra Docker kontajneru a bude ich teda možné volať bez potreby overenia tokenu. Systém totiž nedisponuje žiadnym tokenom a akékolvek volania, ktoré vykonáva z vlastnej iniciatívy neprechádzajú z dôvodu zlyhania autorizácie. Keď budú tieto endpointy existovať a volania budú smerované na ne, v konfiguračnom súbore pre Symfony Messenger je možné nastaviť komunikáciu na asynchrónnu. Dovtedy je nutné, aby hodnotenie prebiehalo synchrónne, keďže vtedy sa k iným mikroslužbám pristupuje s využitím tokenu používateľa.

5.2.3 Evaluation_log

Vzhľadom na to, že hodnotenia odpovedí sa môžu meniť automaticky aj manuálne, je potrebné, aby existoval spôsob, akým bude možné sledovať ich zmeny. Na to je potrebné vytvoriť systém logov. Jeho databázová štruktúra je popísaná v sekcii 3.1.1. Realizované môžu byť 2 spôsobmi – buď priamo v databáze pomocou triggeru alebo v kóde ako event listener.;

5.2.4 Veľkosť mikroslužby

Vzhľadom na množstvo kódu v mikroslužbe dáva zmysel uvažovať nad jej rozdelením, kde by sa oddelili samotné CRUD operácie nad danými entitami od algoritmov na ich vyhodnocovanie. Porovnávanie diagramov, oprava normalizácie či porovnávanie transformácií vnášajú do kódu veľké množstvo prídavných tried, ktoré zväčšujú kódovú základňu mikroslužby. To môže predstavovať prekážku v jednoduchom pochopení kódu novými SP tímami a spomaľovať tým vývoj.

Záver

Hlavným cieľom tejto bakalárskej práce bola modernizácia systému na vyhodnocovanie odpovedí študentov v testoch pre predmet Databázové systémy. To zahŕňa implementáciu novej mikroslužby, ktorá je schopná plne nahradiť funkcionality starého DBS portálu s pridanými vylepšeniami.

Práca začala analýzou DBS portálu – systému využívaného na výučbu predmetu BI-DBS. Na základe nej bola navrhnutá nová štruktúra databázy spĺňajúca ako staré, tak aj nové biznisové požiadavky. Tá slúžila ako základ pre návrh systému na automatické a manuálne hodnotenie odpovedí.

Výsledkom práce je mikroslužba na správu a vyhodnocovanie odpovedí študentov. Mikroslužba podporuje tvorbu odpovedí a ich opravu. Okrem manuálneho hodnotenia je systém schopný vyhodnocovať odpovede študentov aj automaticky. To môže prebiehať buď synchrónne alebo asynchrónne v závislosti od aktuálneho nastavenia. Tento mechanizmus je schopný zabrániť preťaženiu systému v čase vysokého náporu, kedy by starý portál mohol zlyhať. Ďalej je systém schopný synchronizovať hodnotenia identických odpovedí študentov pre zachovanie maximálnej konzistentnosti pri oprave ekvivalentných odpovedí. Kód využíva princípy objektovo orientovaného programovania pre jednoduchú modifikovateľnosť do budúcna.

Mikroslužba má vystavené API rozhranie, ktoré je plne zdokumentované. API dokumentácia obsahuje všetky endpointy, s ktorými je možné pracovať spolu s ich popisom, príkladmi volania a všetkými možnými odpoveďami. Rozhranie mikroslužby je už čiastočne využívané na frontende, pričom sa očakáva plné využitie v budúcom semestri B231 v rámci predmetu BI-SP2.

Hlavným prínosom tejto práce je transformácia ťažko udržiavateľného kódu DBS portálu do ľahko rozšíriteľnej formy mikroslužby. To uľahčí prácu ďalším vývojárom, ktorí budú do portálu dopĺňať nové funkcionality. Ďalším prínosom je spríjemnenie písania testov študentom, keďže budú mať možnosť vracať sa k svojim predchádzajúcim odpovediam vďaka pridanému verzovaniu odpovedí. V práci bol taktiež navrhnutý spôsob rozšírenia vyhodnocovania odpovedí o skupiny ekvivalencie, ktoré urýchlia a vylepšia automatické opravovanie odpovedí, čím sa zníži množstvo manuálnej práce vykonávanej vyučujúcimi.

Dodatok A

Ukážky kódu

```
{
    "entities": [
        {
            "name": "Chovatel",
            "parent": null,
            "attr": [
                {
                    "name": "id",
                    "primary": true,
                    "unique": false,
                    "nullable": false,
                    "type": null
                },
                {
                    "name": "meno",
                    "primary": false,
                    "unique": false,
                    "nullable": false,
                    "type": null
                }
            ]
            },
            {
            "name": "Krmenie",
            "parent": null,
            "attr": [
                {
                    "name": "id",
                    "primary": true,
                    "unique": false,
                    "nullable": false,
                    "type": null
                },
                {
                    "name": "datum",
                    "primary": false,
                    "unique": false,
                    "nullable": false,
                    "type": null
```

```
                },
                {
                    "name": "mnozstvo",
                    "primary": false,
                    "unique": false,
                    "nullable": false,
                    "type": null
                }
            ]
        },
        {
        "name": "Zviera",
        "parent": null,
        "attr": [
            {
                "name": "id",
                "primary": true,
                "unique": false,
                "nullable": false,
                "type": null
            },
            {
                "name": "druh",
                "primary": false,
                "unique": false,
                "nullable": false,
                "type": null
            },
            {
                "name": "meno",
                "primary": false,
                "unique": false,
                "nullable": false,
                "type": null
            },
            {
                "name": "datum_narodenia",
                "primary": false,
                "unique": false,
                "nullable": true,
                "type": null
            }
        ]
        }
    ],
    "relations": [
        [
            {
                "entity": "Krmenie",
                "identifying": false,
                "optional": false,
                "cardinality": 0,
                "xor": null
            },
            {
                "entity": "Chovatel",
                "identifying": false,
```

```
                "optional": true,
                "cardinality": 1,
                "xor": null
            }
        ],
        [
            {
                "entity": "Zviera",
                "identifying": false,
                "optional": true,
                "cardinality": 1,
                "xor": null
            },
            {
                "entity": "Krmenie",
                "identifying": false,
                "optional": false,
                "cardinality": 0,
                "xor": null
            }
        ]
    ],
    "notes": []
}
```

Výpis kódu A.1 Vzorový JSON objekt odpovede typu diagram

Bibliografia

1. *What Is SDLC (Software Development Lifecycle)?* [online]. [cit. 2023-04-05]. Dostupné z: https://aws.amazon.com/what-is/sdlc/.
2. *SDLC – Waterfall Model* [online]. [cit. 2023-04-05]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm.
3. *SDLC – Iterative Model* [online]. [cit. 2023-04-06]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_iterative_model.htm.
4. *Functional and Nonfunctional Requirements: Specification and Types* [online]. 2021-07-23. [cit. 2023-04-19]. Dostupné z: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/.
5. GRADY, R.B. Measuring and Managing Software Maintenance. *IEEE Software*. 1987, roč. 4, č. 5, s. 35–45. Dostupné z DOI: 10.1109/MS.1987.231417.
6. STEPHENS, Rod. *Beginning Software Engineering*. 2. vyd. Wiley, 2022. ISBN 9781119901709.
7. NEWMAN, S. *Building Microservices*. O'Reilly Media, 2021. ISBN 9781492033998.
8. *Scalability* [online]. [cit. 2023-04-16]. Dostupné z: https://www.gartner.com/en/information-technology/glossary/scalability.
9. *Scalability* [online]. [cit. 2023-04-16]. Dostupné z: https://www.suse.com/suse-defines/definition/scalability/.
10. *Horizontal Vs. Vertical Scaling: Which Is Right For Your App?* [online]. [cit. 2023-04-16]. Dostupné z: https://www.missioncloud.com/blog/horizontal-vs-vertical-scaling-which-is-right-for-your-app/.
11. *Message Queues* [online]. [cit. 2023-04-24]. Dostupné z: https://aws.amazon.com/message-queue/.
12. JOHANSSON, Lovisa. *What is Message Queuing?* [online]. 2023-03-01. [cit. 2023-04-24]. Dostupné z: https://www.cloudamqp.com/blog/what-is-message-queuing.html.
13. NATAN, Ron Ben. *Implementing Database Security and Auditing*. 2005. ISBN 978-1-55558-334-7.
14. ZHIYUE, Yi. *Design a Table to Keep Historical Changes in Database* [online]. 2019-09-17. [cit. 2023-02-19]. Dostupné z: https://yizhiyue.me/2019/09/17/design-a-table-to-keep-historical-changes-in-database.
15. VALLABHANENI, S.R. *Corporate Management, Governance, and Ethics Best Practices*. Wiley, 2008. ISBN 9780470255803.
16. PLYSKACH, Andrii. *Refaktoring testové části backendu portálu dbs.fit.cvut.cz*. 2021. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

17. PLYSKACH, Andrii. *Modernizace a migrace DBS portálu*. 2023. České vysoké učení technické v Praze, Fakulta informačních technologií. Diplomová práca.

18. LUKAČÍN, Jakub. *Refaktoring backend části portálu dbs.fit.cvut.cz - semestrální práce*. 2021. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

19. PEJŠA, Petr. *Systém pro podporu testování v BI-DBS*. 2016. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

20. *Dynamické programování* [online]. [cit. 2023-02-09]. Dostupné z: `https://courses.fit.cvut.cz/BI-AG1/lectures/media/bi-ag1-p10-handout.pdf`.

21. KUBIŠ, Martin. *Překladač z relační algebry do SQL*. 2016. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

22. KUBIŠ, Martin. *Rozšíření překladače relační algebry*. 2019. České vysoké učení technické v Praze, Fakulta informačních technologií. Diplomová práca.

23. ŠACH, Martin. *Podpora tvorby a automatická oprava zjednodušeného relačního zápisu v portálu dbs.fit.cvut.cz*. 2020. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

24. ERBEN, Marek. *Automatické generování a oprava otázek na normalizaci databáze pro předmět BI-DBS*. 2018. České vysoké učení technické v Praze, Fakulta informačních technologií. Bakalárska práca.

25. HULL, E.; JACKSON, K.; DICK, J. *Requirements Engineering*. Springer London, 2010. ISBN 9781849964050.

26. *Strategy* [online]. [cit. 2023-04-23]. Dostupné z: `https://refactoring.guru/design-patterns/strategy`.

27. *xxHash* [online]. [cit. 2023-04-23]. Dostupné z: `https://php.watch/versions/8.1/xxHash`.

28. *Events* [online]. [cit. 2023-05-03]. Dostupné z: `https://www.doctrine-project.org/projects/doctrine-orm/en/2.14/reference/events.html`.

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `src` — zdrojové kódy
  - `implementation` — zdrojové kódy implementácie
    - `docker` — docker-compose.yaml, Dockerfile
    - `docs` — API dokumentácia
    - `TestEvaluations` — implementácia mikroslužby
  - `thesis` — zdrojový kód práce vo formáte LaTeX
    - `ctufit-thesis.tex` — hlavný zdrojový súbor práce
    - `attachments` — prílohy a zadanie práce
    - `img` — obrázky použité v práci
- `bp_jakub_pavlicko.pdf` — text práce vo formáte PDF