FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Modernizace a migrace DBS portálu
Student:	Bc. Andrii Plyskach
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Zadání: Portál DBS je velmi kladně hodnocený nástroj pro podporu výuky relačních databází na FIT ČVUT. Cílem této diplomové práce je zajistit jeho modernizaci a budoucí udržitelnost celého projektu.

Postupujte v těchto krocích:
1. Řádně analyzujte současný stav DBS portálu. Zaměřte se minimálně na rozšiřitelnost, snadnost vývoje a také udržitelnost.
2. Na základě analýzy navrhněte vhodné řešení, k zajištění udržitelnosti projektu do budoucna.
3. Pokuste se využít SP týmy pro realizaci vámi navrženého řešení, případně jeho pod částí.
4. Sám se na vývoji podílejte. Řádně oddělte vlastní vývojovou a realizační činnost od výkonu SP týmů či jiných studentů.
5. Zajistěte v maximální možné míře budoucí snadný rozvoj Vašeho řešení.
6. Shrňte dosažené výsledky, obhajte připravenost projektu pro další rozvoj/vývoj.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. září 2022 v Praze.

Diplomová práce

MODERNIZACE A MIGRACE DBS PORTÁLU

Bc. Andrii Plyskach

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Hunka
2. ledna 2023

Obsah

Seznam obrázků

Seznam tabulek

Chtěl bych poděkovat především vedoucímu diplomové práce Ing. Jiřímu Hunkovi za možnost pokračovat ve vývoji a zlepšení DBS portálu, za poskytnuté informace týkající se všech jeho důležitých funkcí a také za cenné rady ohledně řízení vývojových týmů. Dále bych chtěl poděkovat za spolupráci všem, kteří se zúčasnili vývoje DBS portálu v rámci předmětu „Softwarový projekt“. Mé poděkování patří také Ing. Oldřichu Malcovi, Ing. Filipu Dolníkovi a Bc. Maxovi Hejdovi za jejich doporučení a rady během vývoje. Nakonec děkuji své rodině, která mě v průběhu celého studia podporuje.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. ledna 2023 .....

Abstrakt

Tato diplomová práce se zabývá analýzou příčin a problémů majících negativní dopad na údržbu a budoucí rozvoj DBS portálu. Na základě analýzy těchto problémů jsou navržena řešení, která by je měla odstranit. Hlavním řešením je vytvoření nové architektury tohoto portálu, která by byla více odolná vůči problémům spojeným se způsobem jeho vývoje. Nejprve byla provedena podrobná analýza vývoje tohoto systému a nalezeny jeho nedostatky, které brání efektivitě práce a jsou příčinou zhoršení celkového stavu projektu. Na základě analýzy byla vytvořena nová architektura DBS portálu. Ta by měla více odpovídat současným reáliím. Také byla navržena řešení několika dalších problémů spojených s vývojem portálu v rámci SP týmů. Nakonec se práce věnuje návrhu, implementaci a testování autorizace uživatelů v rámci nové architektury DBS portálu a jeho budoucím rozvojem.

Klíčová slova DBS portál, DDD, mikroslužby, architektura, Docker, REST API, PHP, Symfony

Abstract

This diploma thesis deals with the analysis of the causes and problems having a bad impact on the maintenance and future development of the DBS portal. Based on the analysis of these problems, solutions are proposed that should eliminate these problems. The main one of these solutions is the creation of a new architecture of this portal, which would be more resistant to problems associated with the way it is developing. At the begining, a detailed analysis of the development of this system was carried out and its shortcomings were found, which hinder the efficiency of the work and cause the deterioration of the overall state of the project. Based on this analysis, a new architecture of the DBS portal was created, which should be more in line with current realities. Solutions to several other problems associated with the development of the portal within the SP teams were also proposed. Finally, the work is devoted to the design, implementation and testing of user authorization within the new DBS portal architecture and its future development.

Keywords DBS portal, DDD, microservices, architecture, Docker, REST API, PHP, Symfony

Seznam zkratek

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
CQRS	Command-Query Responsibility Segregation
CSS	Cascading Style Sheets
ČVUT	České vysoké učení technické v Praze
DBS portál	Systém pro podporu výuky předmětu BI-DBS
DDD	Domain Driven Design
DI	Dependency Injection
DQL	Doctrine Query Language
DTO	Data transfer object
ETL	Extrakt Transform Load
FIT	Fakulta informačních technologií
GIT	Název systému správy verzí
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IdP	Identity Provider
IMAP	Internet Message Access Protocol
JS	JavaScript
JSON	JavaScript Object Notation
JWT	Json Web Token
MVC	Model-View-Controller
NEON	Nette Object Notation
OAuth	Open Authorization
ORM	Object Relation Mapping
PHP	PHP: Hypertext Preprocessor
POP3	Post Office Protocol
QueryBuilde	Rozhraní umožnující vytvářet dotazy v jazyku DQL
RA	Relační algebra
Redmine	Systém pro řízení projektu
REST	Representational State Transfer
SAML	Security Assertion Markup Language
Slack	Komunikační systém
SMTP	Simple Mail Transfer Protocol
SP	Service Provider
SP tým	Tým v rámci předmětu BI-SP1 nebo BI-SP2
SQL	Stands for Structured Query Language
SOAP	Simple Object Access Protocol
SRP	Single Responsibility Principle
SSO	Single Sign-On
SSL	Secure Sockets Layer
Systém KOS	Studijní informační systém
SW	Semester work
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Kapitola 1
Úvod

Vývoj softwarového systému je vždy náročná, dlouhodobá a pracná činnost, která vyžaduje určitou skupinu schopných lidi, jejich trpělivost a motivaci. Navíc platí, že čím déle trvá vývoj, tím je větší pravděpodobnost neúspěchu nebo vzniku problémů spojených se zastaralými technologiemi, odchodem lidí se znalostmi tohoto systému apod. Když projekt stále opouští lidé se znalostmi systému a technologií, v systému se více a více objevují chyby, špatné konstrukce v kódu, nesprávné použití knihoven a spoustu dalších problémů, jako jsou například zastaralé technologie. To nedovoluje použití nových knihoven a možnosti, které nabízí nové verze jazyka PHP. V našem vývojovém týmu se odehrála obdobná situace s vývojem DBS portálu, systémem pro podporu výuky předmětu Databázové systémy (BI-DBS), který je vyučován na Fakultě informačních technologií ČVUT v Praze. Systém je vyvíjen od roku 2013 a počet lidí, kteří se na tomto vývoji podíleli, už dosáhl několik desítek a stále prudce roste. To znamená, že vývoj tohoto projektu je čím dál tím složitější, protože každý rok přicházejí noví lidé, kteří se většinou pouze seznamují s technologiemi na projektu a nemají dostatek zkušeností. Proto školení nových lidí bývá dost složité a často není dostatek času na to, aby se plně zorientovali v takovém velkém projektu. Navíc už bylo o tomto projektu napsáno a obhájeno více než deset závěrečných prací [1–13], které buď něco zlepšovaly, nebo přidávaly nějaké nové funkce.

V této diplomové práci navazuji jak na svou bakalářskou práci[1], tak i na jiné závěrečné práce [2–13] o DBS portálu, ve kterých je definována velká část funkcí systému. Tyto funkce se budu snažit migrovat do nového systému spolu s SP týmy s použitím principů a myšlenek uvedených v této práci. Cílem této diplomové práce bude prozkoumání problémů spojených s údržbou a rozvojem systému, existujících požadavků a návrhů, které byly už publikovány v existujících závěrečných pracích, a navržení kompletní architektury nového systému, který bude složen z několika menších částí (mikroslužeb) komunikujících mezi sebou pomocí REST[1] rozhraní.

Každý z nás se určitě setkal s problémem nedokončení nějakého projektu, úkolu či práce. Ve své bakalářské práci jsem se snažil vylepšit současný testový modul DBS portálu, který se dostal do stavu, v němž se budoucí údržba stala dost složitá a drahá. Bohužel jsem nebyl vzhledem k velikosti projektu schopen plně dokončit implementaci v rámci své bakalářské práce a některé věci zůstaly nedokončené. Proto dalším cílem této diplomové práce bude analyzovat rozdělanou práci a maximálně využít její myšlenky. S použitím těchto myšlenek si kladu za cíl přepracovat celkovou architekturu systému a dosáhnout stavu, ve kterém už bude možné vidět nějaký výsledek provedené práce.

Hlavním přínosem této práce bude nová, flexibilní architektura DBS portálu, která dovolí

[1]REST je zkrátka pro REpresentational State Transfer, to je architektonický návrhový styl, který popisuje jednotné rozhraní mezi fyzicky oddělenými komponentami a umožňuje tvorbu distribuované architektury systému. Komunikace je zajištěna na základě protokolu HTTP. [14]

rozdělit aktuální monolitní aplikaci na větší množství menších částí. Tyto menší aplikace budou plně nezávislé na implementacích ostatních, proběhne tedy výměna monolitní architektury za architekturu mikroslužeb. Veškeré závislosti systémů budou zabezpečené pouze pomocí REST API. To umožní budoucím SP týmům zabývat se implementací pouze nějaké malé části portálu, což bude rozhodně vhodnější pro málo zkušené lidi. Navíc si pak budou schopni zvolit pro implementaci i jiné jazyky, pokud to budou potřebovat. Výsledek by měl nabízet takovou volnost, aby při potřebě změnit nějakou z částí systému tato změna co nejméně ovlivnila zbytek systému.

Dalším výstupem mé diplomové práce bude sjednocený slovník pojmů, postupů, konvencí a dokumentace systému, který dovoluje mít jeden pohled na naši doménu a ukazuje, jak by se mělo postupovat při implementaci nových rozšíření.

Kapitola 2

Analýza současného stavu projektu

Po dokončení své bakalářské práce[1] jsem se kvůli nedostatku času a zahájení magisterského studia projektu nevěnoval. Během této doby se na projektu vystřídalo několik SP týmů a byly obhájeny další závěrečné práce[11–13]. Také se objevily nové poznatky na frontendu, který se nově implementuje pomocí jazyka JavaScript a knihovny Vue.js. Na backendu, který jsem implementoval v rámci bakalářské práce, se moc nepracovalo. SP týmy se zabývaly opravou vzniklých chyb v současném DBS portálu. Z tohoto důvodu moje implementace zůstala téměř tak, jak byla implementovaná v mé bakalářské práci. Velká část frontendu byla implementovaná v rámci bakalářské práce a na ni navázal SP tým, kterému se podařilo tuto implementaci dále rozšiřovat a upravovat dle aktuálních požadavků.

Na projektu jsem se začal podílet v zimním semestru v roce 2021, abych dokončil to, čemu jsem se věnoval ve své bakalářské práci. Objevili se studenti, kteří potřebovali SP tým, a proto jsme sestavili nový tým. Ten se začal zabývat backend vývojem.

Každý tým se v rámci předmětů BI-SP1 a BI-SP2 skládá z 4 až 6 lidí a má jednoho vedoucího týmu. Tento vedoucí zodpovídá za řízení tohoto týmu, rozdělení práce a komunikaci v týmu. Vedoucí těchto předmětů je Ing. Jiří Hunka, který je zodpovědný za vedení předmětů. Jeho úkolem je organizace práce na projektu, validace výsledků a hodnocení práce studentů. Také zadává úkoly, které je nutné splnit pro procvičení látky předmětu BI-SI1 a definuje požadavky pro získání započtu. Osobně jsem se na vývoji podílel jako projektový manager, který se zabýval řízením backend týmů, komunikací mezi týmy, definicí softwarových požadavků, kontrolou implementovaných řešení a dalších činnosti, jež popisuji v této diplomové práce.

2.1 Vedení týmu vývoje a vzniklé problémy

Začátek seznámení se s projektem přinesl mnoho problémů pro vývojové týmy. Noví lidé na projektu bez znalostí systému a technologií potřebují hodně času na to, aby se v systému vyznali a začali alespoň trochu chápat jeho funkce. V zimním semestru 21/22 jsme měli 3 týmy: dva pracovaly s frontend vývojem a jeden s vývojem backendu. Jeden z týmů zůstal v původní podobě, kterou měl během předmětu BI-SP1, a ten vykazoval nadprůměrný výkon. Ostatní byly složené z různých lidí a ne každý z nich měl zkušenost s projektem a technologiemi, které se používaly, což velmi brzdilo výkon v týmu a na projektu celkově.

Osobně jsem neměl žádnou zkušenost s vedením týmu, a proto byl na začátku zmatek. Backend tým, za který jsem byl zodpovědný, nebyl zorientovaný a měl problém s pochopením nové implementace, kterou jsem dělal v rámci bakalářské práce. Často kladli členové týmu nějaké

dotazy, nebo implementovali úkoly s chybami. Navíc vedoucí tohoto týmu také neměl zkušenosti a svoji první týmovou práci rozdělil takovým způsobem, že byli všichni na sobě závislí, a nemohli tak pracovat, dokud jejich kolega nedokončil svou práci. Během tří týdnů nepřinesla tato práce žádný dobrý výsledek. Kromě toho jsem byl jediný, kdo definoval REST API rozhraní pro DBS portál. Ostatní vždy se mnou řešili nějaké problémy s tímto spojené, což ubíralo čas, který jsem mohl věnovat jiné práci. Také jsem se občas snažil dělat nějaké code review, abych zaškolil lidi, kteří s těmito technologiemi pracují poprvé a mají s tím problémy. Ve výsledku jsem byl zcela zatížen a nemohl jsem efektivně pomáhat. Hlavní můj problém byl ten, že jsem se snažil dělat všechno sám a nedelegoval jsem úkoly na lidi, kteří by je byli schopni vyřešit. Tady bych chtěl velmi poděkovat vedoucímu práce, Ing. Jiřímu Hunkovi, za to, že mě upozornil na tento problém. Poté, co jsem si uvědomil příčinu, jsem byl schopen snížit svoje zatížení a vyřešit problém s časem. Volný čas jsem pak mohl věnovat řešení důležitějších problémů na projektu.

Ovšem toto byla jenom jedna z drobných potíží a byl to můj osobní problém, který nebyl tou hlavní příčinou všech problémů na projektu. Některé z nich už jsem popsal výše, ale pro lepší představu zkusím všechny sjednotit do následujícího přehledu:

- Zastaralé technologie, nutnost dodělávat funkce, které jsou v novějších knihovnách implementované.
- Neznalost technologie.
- Nepochopení myšlenek založených v implementaci.
- Špatné pochopení toho, co se aktuálně implementuje a k čemu to je. Nepochopení stavu projektu jak v rámci týmu, tak i mezi samotnými týmy.
- Špatné rozdělení práce v týmu.
- Připojení dalších členů týmu, potřeba školení.
- Chybějící code review od ostatních členů týmů.
- Závislost všech týmů na mně jako návrháři rozhraní.
- Závislost mezi týmy, která se řešila pouze na schůzkách a nebo se mnou.
- Neevidované požadavky.

2.2 Důvody vzniku objevených problémů

2.2.1 Velikost systému a jak vznikla myšlenka mikroslužeb

Mezi hlavní důvody vzniku těchto problémů patří nedostatek motivace. Část lidí, pracujících na projektu, nebylo dostatečně motivováno k vytvoření dobrého výstupu. DBS portál je dost velký projekt pro lidi, kteří dosud třeba nikdy nepracovali s tak rozsáhlými projekty. Pro představu: zdrojový kód tohoto portálu má kolem 120000 – 140000 řádků[1], bez ohledu na některé jeho části, které jsou vedené jako samostatné projekty a instalované knihovny. Začít se orientovat v takto velkém projektu vyžaduje spoustu času a úsilí. Tento čas nejsou lidé bez motivace projektu ochotni věnovat, a proto se pak objevují problémy s pochopením systému. Navíc je tento projekt implementován pomocí starších verzí frameworku Nette a jazyka PHP, který znemožňuje používání nových konstrukcí jazyka a současných knihoven. Například nebylo možné použít pro implementaci REST API rozhraní současné knihovny, jež by se nám hodily. Museli jsme používat

[1] Spočítáno pomocí bash skriptu:

- PHP soubory přibližně 70000 řádků,
- SQL skripty kolem 22000 řádků,
- JS skripty kolem 4000 řádků,
- zbytek: různé konfigurační soubory, HTML šablony a další.

knihovnu, která by podporovala integraci s Nette. Ta nebyla pro naše požadavky vhodná. Hodně věcí jsme museli implementovat sami, a proto jsme strávili dost času implementací něčeho, co by se dalo použít již hotové. To všechno snížilo motivaci jednotlivých lidi věnovat tomu čas. Z toho pak vyplývají problémy, které jsem popsal výše: s pochopením systému, s neznalostí technologií, s časem potřebným na zaškolení, s celkovým pochopením práce na projektu.

Položil jsem si otázku: „Co by se dalo udělat, aby se tato produktivita zlepšila?“ Prvním krokem bylo oddělit testovou část DBS portálu a implementovat ji jako samostatný systém komunikující se zbytkem pomocí REST API. Výhodou byla možnost použít nové technologie: novou verzi jazyka (PHP8) a Symfony framework, který by se k tomu hodil více. Navíc bychom měli mnohem menší systém a byl by zaměřen na jedno konkrétní využití. Ve výsledku by byl sytém menší, jednodušší a snadněji rozšiřitelný.

Důvodem přechodu na Symfony byla nedostatečná podpora REST API v Nette frameworku. Spoustu hotových věcí v Symfony by se v Nette muselo implementovat a udržovat SP týmem, tj. zbytečný narůst udržovaného kódu a zbytečná náročnost systému. Navíc jsem začal ve své bakalářské práci implementovat testový modul pomocí Doctrine ORM, což je součást Symfony a existující kód se dá snadno migrovat do nového systému. Po diskuzi s vývojovým týmem se jim myšlenka líbila, a to hlavně díky možnostem přejít na novější verze knihoven a možnosti rozdělit zodpovědnost.

Následně jsem zkusil analyzovat výhody a nevýhody tohoto řešení a společně s vyučujícími předmětu BI-DBS a lidmi, kteří se v minulosti zabývali vývojem tohoto portálu, jsme vedli zajímavou diskusi. Ve výsledku jsme došli k tomu, že můj návrh vyřeší tento problém jen zčásti, ale za pár let se tento systém zase dostane do současného stavu a bude potřeba ho celkově upravovat. Každý rok k nám přicházejí v rámci softwarových projektů (SP) noví studenti, kteří nemají moc zkušeností s technologiemi, a kód se tedy bude s každým rokem plnit chybným použitím jazyka a knihoven. Časem se dostane do současného stavu, kdy bude jeho údržba hodně drahá a náročná. Tím jsme si nadefinovali další dost důležitý požadavek: potřebujeme mít možnost po pár letech vyjmout libovolnou část systému, upravit ji a integrovat zpět. Toto mě navedlo na koncept, který se stal základem této práce, a to jsou mikroslužby.

Mikroslužby je pojem z oblasti vývoje softwarových aplikací, který označuje variantu softwarové architektury orientovanou na služby. Bohužel neexistuje přesná definice, která by vymezovala tento pojem, ale postupem času došlo ke shodě na následujících charakteristikách a výhodách těchto aplikací: [15, 16]

- Možnost na sobě nezávislých nasazení.
- Mohou být implementované pomocí různých technologií (jazyků, databází, dle toho, co se nejvíce hodí).
- Zodpovídají za konkrétní byznys funkce.
- Jsou malými součástkami, které jsou ohraničené svým kontextem.
- Komunikují se svým okolí pomocí technologicky nezávislých protokolů (HTTP).
- Mikroslužby jsou autonomně vyvíjené, nezávisle nasaditelné, dobře škálovatelné, snadno testovatelné.

Použití mikroslužeb má také svoje nevýhody: [15]

- Vysoká závislost na sítích. Na rozdíl od monolitních aplikací se v mikroslužbách posílá mnohem více požadavků na server, protože získáváme veškerou informaci na stránce z více různých mikroslužeb, což vyžaduje několik asynchronních volání.
- Vyžaduje dobrý návrh a rozdělení na mikroslužby, aby nedocházelo k problémům spojeným se závislostmi, tj. jedna mikroslužba by neměla obsahovat žádné další znalosti o jiné, kromě veřejného rozhraní.
- Interní volání jiných mikroslužeb může způsobovat další problémy.

- Automatizované testování reálného provozu je nezbytné k tomu, aby se zamezilo vzniku chyb, které by mohly ovlivnit stabilitu aplikace.
- Architektura mikroslužeb potřebuje pokročilý monitoring, který dokáže upozornit na vzniklé problémy.
- Sdílení knihoven mezi jednotlivými mikroslužbami. V našem případě je vyřešeno pomocí „DBS Utils bundle“, který je popsán v podkapitole 4.4.
- Chyby mohou vznikat v rámci jedné mikroslužby, ale objevovat se v jiné části systému. Proto je potřeba mít globální přístup ke sběru logů a trasování aplikace. V našem případě budeme používat proxy „Traefik“, který bude směrovat jednotlivé požadavky na konkrétní mikroslužby a bude provádět monitorovací funkci, viz podkapitola 4.2.

Výsledkem tohoto rozhodnutí budou menší části systému, tj. snížení složitosti systému a snížení problémů s pochopením systému jako celku. Tým si zvolí jeden ze systémů a s ním začne pracovat, proto se souběžně nemusí zabývat zbytkem portálu. Navíc mikroslužby můžeme vyvíjet nezávisle na sobě. Pokud máme 2 vývojové týmy, mohou se paralelně zabývat různými částmi a navzájem se vůbec neovlivňovat, což se nám stávalo v případě monolitního DBS portálu. Můžeme také použít novější technologie. To nám dovolí psát čistější kód a používat více existujících knihoven. Ve výsledku máme méně kódu, který potřebujeme udržovat a upravovat, flexibilnější vývoj, lepší škálovatelnost a mnoho dalších výhod. Jednoduchost systému ve výsledku způsobuje jednodušší pochopení a tím i větší motivaci pracovat a produkovat dobrý výsledek. Porovnání výsledků práce dvou týmů na starém a novém systému uvedu v následujících kapitolách. Rozdíl je tam opravdu viditelný.

2.2.2 Nekonzistence v názvech

Dalším důvodem problému s pochopením systému je pojmenování tříd, objektů, souborů, tabulek atd. Na systému každý rok pracují jiní lidé, což způsobuje zvětšení nekonzistence mezi názvy různých objektů v kódu. Například v portálu můžeme nalézt databázovou tabulku test_created_test. Člověk, který to vidí poprvé, si bude myslet, že to asi bude tabulka vytvořeného testu pro studenta. Začne něco implementovat a za chvíli zjistí, že existuje tabulka test_student_test, která se dá interpretovat jako test studenta. A co znamená ta původní tabulka, tento člověk vědět nebude. Až po delší analýze kódu zjistí, že tabulka test_created_test představuje termín pro nějakou sadu testů. To vůbec neodpovídá pojmenování této tabulky a zmate lidi, kteří s tím pracují. Ve své bakalářské práci jsem se snažil podobné názvy eliminovat, ale část z nich stále zůstala v kódu. Opravil jsem jen část s testy, ale ve zbylé části systému tyto problémy zůstaly. Pro jejich celkové vyřešení zkusím zadefinovat slovník pojmů a procesů, který sjednotí naši doménu v realitě s doménou implementační, viz podkapitoly 3.3 a 3.4. Navíc zkusím zadefinovat jmenné konvence pro práci s verzovacím systémem GIT, viz podkapitola 4.1.1, dále konvence pro vytvoření databázových objektů a konvence pro návrh REST API, které bude potřeba striktně dodržovat, aby lidé, v budoucnu pracující s tímto systémem, byli schopni rychle pochopit implementaci jakékoliv mikroslužby, viz podkapitola 4.1.

2.2.3 Motivace a code review

V zimním semestru 21/22 jsme měli v rámci předmětu BI-SP2 tým, který se zabýval vývojem DBS portálu. Na začátku semestru jsme jako vždy doporučovali studentům dělat navzájem code review, aby byl výsledek práce kvalitní. Měli jsme mnoho dalších důvodů, jako například zaškolení lidí, lepší pochopení systému, aktuální situace na projektu apod. Tento tým pracoval ještě na monolitní architektuře DBS portálu. Jeden z problémů byl v tom, že lidé v týmu nedělali code review pořádně. Uvedu příklad: student zaznamenal, že udělal code review, a poté se kód dostal ke mně na kontrolu. Když jsem ho zkontroloval, našel jsem hned logickou chybu v kódu. Problém byl

v tom, že se přistupovalo k neexistující proměnné, což by ve výsledku způsobilo chybu programu. To znamená, že se student maximálně podíval na kód v systému Gitlab, ale nepochopil, co ten kód znamená a co dělá. Lze z toho odvodit, že lidé buď nevěděli, jakým způsobem se má dělat code review, nebo neměli motivaci to udělat pořádně. Po ověření tohoto kódu jsem se rozhodl vysvětlit výhody správně provedeného code review. Rozepsal jsem v komunikačním systému Slack detailně, proč a jak by se mělo dělat code review. Ve výsledku se kvalita kódu trochu zlepšila, ale celkově to nemělo velký vliv. Domnívám se, že příčinou byla nízká motivace lidí v týmu. Z tohoto důvodu jsem byl více zatížen, protože jsem vždy objevil nějaké problémy v kódu, když jsem ho validoval.

V letním semestru 21/22 jsme měli v rámci předmětu BI-SP1 už 2 nové týmy. V této době jsme začali pracovat na mikroslužbách. Každý tým si zvolil jednu, pracoval na její analýze, návrhu a následně implementaci. Týmy vycházely z rozdělení systému popsaném v kapitole 3.5. Celkově měli lidé v týmu lepší motivaci pracovat na projektu, než tomu bylo v předchozím týmu. Předpokládám, že to bylo ovlivněno především faktem, že začínali pracovat na novém systému, používali nové technologie. Také absolvovali počáteční školení, pomocí kterého si mohli „osahat“ používaný framework a dostat zpětnou vazbu, viz podkapitola 2.3. Kromě toho řešili pouze malou část systému a nezabývali se návaznostmi na zbytek DBS portálu. Jeden z týmů dělal perfektní code review ve školním verzovacím systému Gitlab, což značně zvětšilo počet odhalených chyb. Když jsem nakonec dostával úkol ke kontrole, mohl jsem jen upozornit na nějaké drobnosti s dodržením konvence – viz podkapitola 4.1, ale celkový výstup byl kvalitní. Jiný tým zahrnoval méně zkušené lidi, proto byl jejich výstup o trochu horší. Ale ve srovnání s týmem, který jsme měli v předchozím semestru, byly výsledky lepší a tento tým se dost rychle učil. Abychom mohli rychleji naučit lidi z druhého týmu, dohodli jsme se, že si budou týmy dělat code review navzájem. To by mělo pomoct lépe předávat zkušenost mezi nimi a naučit je psát kvalitnější kód. Ve výsledku se druhý tým zlepšil v použití konstrukcí jazyka a frameworku.

Mohlo by z toho vyplynout, že jednodušší systém je snadnější na pochopení, lidé mají větší motivaci pracovat v týmu a vytvářet lepší výsledek. Navíc takto malý systém (kolem 10000 řádků kódu) můžeme snadno udržovat a rozšiřovat. Lidé jsou schopni rychleji pochopit detaily systému a začít s tím pracovat. Také novější technologie a framework Symfony nabízely mnohem pohodlnější vývoj a snazší pochopení. Vývojové týmy nebyly nuceny vymýšlet nové kolo, ale použily existující knihovny, které ve starém projektu a frameworku Nette nebylo možné použít, což zrychlilo pochopení a vývoj aplikace. To je v aktuální situaci důležité, protože každý semestr k nám přicházejí noví lidé a jiní od nás odcházejí.

Také existuje možnost, že lidé z těch dvou týmů byli zodpovědnější než ti z předchozího semestru a měli zkušenějšího vedoucího, který zvládl dobře řídit vývoj. Druhý letošní tým neměl moc zkušeností, a kdybych ho porovnal s loňským týmem, měl lepší výstup. První tým byl ještě produktivnější, stihl dokončit jednu mikroslužbu a rozpracovat další.

2.2.4 Závislosti mezi týmy a výsledky jejich práce

Koordinovat velké množství nezkušených lidí bývá často výzvou, navíc pokud s tím nemáte dostatek zkušenosti. Na začátku bylo vše dost složité. Závislosti v týmu a mezi týmy byly velké. Jakékoliv problémy každý tým řešil přímo se mnou nebo s vedením SP týmů. Navíc každý z týmů měl problém s pochopením toho, co dělá jiný tým. Jelikož jsem neměl dostatek zkušeností s organizací práce, snažil jsem se každému pomoct a ve výsledku jsem byl přetížen. Řešení tohoto problému bylo kupodivu poměrně jednoduché. Každý tým by měl vytvořit úkol pro jiný tým a ten by ho měl zpracovat. To vyřešilo problém i s neevidovanými požadavky, které chodily mezi mnou a týmy v systému Slack[2]. Další problém byl spojen s celkovou situací na projektu, neměli jsme představu o tom, co se na projektu odehrávalo a co jednotlivé týmy stihly udělat za poslední 2 týdny. Pro lepší pochopení stavu jednotlivých týmů jsem navrhl vytváření krátkého stručného

[2]Slack je obchodní komunikační platforma nabízející velkou sadu funkcí pro posílání zpráv mezi lidmi. Byla vyvinuta americkou softwarovou společností Slack Technologies v roce 2013. [17]

reportu, který by znázornil výkon týmu za poslední 2 týdny. Hlavním bodem bylo to, aby byl tento report krátký a jeho sestavení nezabíralo příliš času. Proto jsem navrhl několik bodů:

- Název iterace.
- Plánovaný rozsah práce – úkoly, které tým plánoval udělat za 2 předchozí týdny: číslo úkolu, název, řešitel.
- Vyřešeno – úkoly, které byly vyřešené: číslo úkolu, název, řešitel.
- Nestihlo se – úkoly, které se nestihly: číslo úkolu, název, řešitel, důvod.
- Plán na další iteraci: Co budeme dělat dále? – Úkoly, které se plánují dělat v průběhu následujících dvou týdnů.

Ve výsledku jsme se mohli na schůzce podívat, co řešil každý z týmů, jak se mu to dařilo a také zjistit možné problémy, které nastaly. Jiné týmy se vždy mohly podívat, co řešili jejich kolegové a zjistit stav jejich práce. Příklad takového reportu je uveden na obrázku A.1 v příloze. Každý z týmů měl také svůj vlastní projekt v systému Redmine[3], což způsobilo problémy s vyhledáváním potřebných informací o projektu. Proto jsem se rozhodl sjednotit vývoj mikroslužeb do jednoho projektu a rozdělit úkoly dle jednotlivých verzí. Každá z verzí odpovídá mikroslužbě, a proto si může každý tým jednoduše vyfiltrovat tu svoji. Navíc to dává celkový pohled na stav projektu. Každý si může zobrazit pokrok v rámci jednotlivých verzí a vědět, jak je na tom konkrétní mikroslužba.

Během času se ukázalo, že reporty vytvářené týmy nejsou moc efektivní a částečně duplikují verze, které se používají v systému Redmine. Proto bylo společně s týmy rozhodnuto, že pro přehled stačí pouze možnosti systému Redmine. Aktuální stav jednotlivých týmů lze zjistit na stránce „Plán“ systému Redmine v projektu „DBS-Microservices“. Tam jsou pro každou verzi vidět rozpracované či dokončené úkoly a také, co se pro tu nebo jinou mikroslužbu aktuálně dělá.

2.3 Zaškolení nových týmů

Pokud se nezkušený člověk, který nikdy nepracoval na velkém projektu, najednou dostane do projektu, jako je „DBS portál“, potřebuje hodně úsilí a motivace, aby se v něm zorientoval. To zabírá obrovské množství času. Pokud ale ještě k tomu vůbec nezná technologie, které se používají na projektu, počet hodin strávených na tom, aby pochopil základní postupy, rychle roste. Jelikož velká část lidí na projektu pracuje pouze jeden semestr a poté odchází, nemáme dost času, aby lidé byli schopni pochopit systém a začít efektivně pracovat. Systém vždy programují nezkušení lidé, což způsobuje spoustu problémů s údržbou samotného kódu. Existuje malý testový projekt, který měl za cíl naučit nové lidé pracovat s jazykem PHP a frameworkem Nette. Tento projekt je dost neefektivní v zaškolování lidí. Za prvé bude nezkušený člověk potřebovat několik hodin, aby byl schopen instalovat a zprovoznit tento projekt lokálně. Za druhé jsou úkoly v tomto zkušebním projektu jednoduché a neučí věcem, které by se při programování reálného systému hodily. Navíc je DBS portál oproti testovému projektu obrovský systém se spoustou závislostí. Proto se na začátku člověk špatně orientuje v kódu. Potřebuje čas, aby mohl všechno pochopit a začít vykonávat úkoly. Tady jsme došli k závěru, že tak obrovský projekt není vhodný pro tento způsob vývoje, kde se pořád mění lidé a jsou nutná nová školení.

Nová architektura DBS portálu nabízí větší množství malých aplikaci, v nichž se bude nezkušený člověk schopný vyznat během několika hodin. Abych zkrátil dobu zkoumání tohoto systému a studování technologií, rozhodl jsem se vytvořit novou zkušební aplikaci. Jelikož se bude nově používat framework Symfony, nemůžeme tak vycházet z původní aplikace, ale potřebujeme úplně novou.

[3]Redmine je systémem s otevřeným zdrojovým kódem, který se používá pro řízení projektů. Umožňuje evidenci požadavků a chyb, nastavení termínů, vykazovaní času apod. [18]

Především musí být instalace a spuštění zkušebního systému co nejjednodušší. Proto jsem použil docker-compose.yaml[4] soubor, kde jsem nadefinoval kontejnery a jejich nastavení potřebné pro provoz aplikace. Pro spuštění tohoto zkušebního projektu pak stačí pouze:

1. instalovat docker[5],
2. naklonovat si repositář z projektu v systému Gitlab,
3. vytvořit a spustit kontejnery pomocí příkazové řádky nebo klienta.

Celkové zprovoznění aplikace zabere maximálně desítky minut, proto se člověk může zaměřit právě na naučení frameworku a jazyka PHP. Více je o spuštění a konfiguraci projektu popsáno v souboru README.md v repositáři tohoto testového projektu. Tam jsou i jiné informace o projektu, doporučené nástroje, úkoly, tipy k nim a způsob odevzdání.

Školení je prováděno následujícím způsobem. Velká část zkušebního systému je již implementovaná, aby se student mohl inspirovat existujícím kódem. To mu pomůže naučit se číst cizí kód a používat konstrukce, které jsou už implementované. Tento způsob je vhodný právě pro ty, kteří pracují s těmito technologiemi poprvé, protože mohou vidět způsob použití.

Úlohy na implementaci funkcí zkušebního systému jsou zadané tak, aby člověk mohl vyzkoušet hlavní možnosti frameworku, aby byl připraven na práci s reálným systémem, věděl, jak funguje konkrétní konstrukce frameworku a jazyka a uměl je správně použít při řešení úkolů na projektu. Úlohy jsou zaměřené na:

- vytvoření nových endpointů[6], metody: POST, PUT, GET, DELETE,
- definice požadavku/odpovědí (HTTP),
- práci se Symfony službami a DI (Dependency Injection),
- práci s entitami (Doctrine),
- práci s QueryBuilder, sestavení požadavků v jazyce DQL (Doctrine Query Language),
- vytvoření migrací,
- tvorbu dokumentace a další.

Po vyzkoušení tohoto testového projektu jsem obdržel celkem dobré výsledky. Studenti strávili na instalaci poměrně málo času, což bylo jedním z cílů tohoto projektu. Jen u některých byl problém s instalací, protože aktuální obraz kontejneru nebyl připraven na použití v operačním systému Windows. Proto bude v následující verzi tohoto projektu potřeba využít nějaký jiný obraz pro tvorbu kontejnerů.

Co se týče implementace, většina lidí docela rychle a správně pochopila principy tohoto frameworku a implementovala tyto úlohy správně. Nejvíce dělalo studentům problém sestavení databázových dotazů pomocí QueryBuilder v jazyce DQL. Proto v dalších verzích tohoto zkušebního projektu plánuji přidat pro inspiraci více ukázek sestavení takových dotazů.

Tento zkušební projekt byl velkým přínosem pro studenty. Když začali implementovat mikroslužby, byl výsledný kód docela kvalitní. Jediný problém, který jsem pozoroval, byl s optimalizací načítání entit a se zbytečnými dvousměrnými vazbami mezi entitami domény. Tyto týmy většinou používaly načítání entit z databáze pomocí Doctrine frameworku, který jako libovolný jiný ORM framework neumí efektivně načíst data z databáze a může způsobovat velké množství požadavků. Proto do další verze zkušebního projektu plánuji také přidat úlohu na prozkoumání počtů požadavků generovaných frameworkem, aby byli studenti schopni pochopit tento problém.

[4]Docker compose je jednoduchý nástroj umožnující spuštění více kontejnerového prostředí na jednom hostitelském stroji. Často se používá pro vytvoření vývojového prostředí, když pracujeme s více kontejnery. Detailněji se lze dozvědět v podkapitole 4.6.7.

[5]Docker je nástroj s otevřeným zdrojovým kódem pro zjednodušení procesu automatického nasazení softwarových aplikací, viz podkapitola 4.6.7.

[6]Endpoint představuje nějaký zdroj a operaci, kterou lze s ním provést. Můžeme provádět různé operace, např. čtení, vytvoření či modifikací dat. Je tvořen na základě protokolu HTTP a používá jeho metody.

Kdybych měl celkově zhodnotit tento zkušební projekt, řekl bych, že byl úspěšný a pomohl studentům zorientovat se v technologiích, které používáme, a naučil je některým správným konstrukcím používaného frameworku a jazyka. Bylo překvapením, že ve výsledku stihl jeden z týmů implementovat až dvě mikroslužby.

Kapitola 3

Návrh

Existující DBS portál je monolitní aplikace, ve které jsou implementované veškeré funkce systému: byznys funkce, uložení a zobrazení dat, volání externích služeb a další. To nám způsobuje velký problém s jakoukoliv úpravou systému. Každá úprava vyžaduje velký zásah do kódu, kterého se přímo netýká, což je ve výsledku příčinou velkého množství zanesených chyb. Budu se proto snažit navrhnout lepší řešení, tj. rozdělení systému na více malých částí pomocí architektury mikroslužeb.

Jedním ze způsobů migrace monolitní aplikace do architektury mikroslužeb je postupná implementace nových a migrace částí funkcí starého systému do nových mikroslužeb. V tomto případě to není zcela vhodné, protože jsou už navržené nové funkce systémů, které byly popsány v bakalářské práci[1], vyžadující jiný způsob uchovávání dat a jiný pohled na fungování systému. Tato metoda postupného přechodu by mohla fungovat, pokud by byl na projektu stejný tým vývojářů od začátku až do konce vývoje. Jelikož toto nelze zaručit, budu muset zvolit jednodušší metodu, dle které navrhnu architekturu a dělení systému od začátku. To je jednodušší, protože když je hotová architektura, každý tým vývojářů si může zvolit jednu z mikroslužeb, udělat podrobnou analýzu, navrhnout řešení a implementovat ho. Během své práce může část kódu zkopírovat ze starého systému, nebo implementovat nově dle aktuálních potřeb. Studenti se tedy budou věnovat pouze implementaci potřebných funkcí systému a nebudou se zabývat promýšlením celkové architektury, což může být pro méně zkušené lidé problematické. Každá z mikroslužeb by měla být malou součástí systému, kterou bude jeden z týmů schopný implementovat v rámci předmětu BI-SP1 a BI-SP2.

Výsledkem práce těchto týmů by měly být plně implementované mikroslužby, které jsou připravené pro použití. Cílem je, aby se v dalších iteracích SP projektů mohl každý z týmů zaměřit na konkrétní malou část systému (jednu mikroslužbu) a tím neovlivnil funkčnost zbytku systému.

V důsledku složité komunikace mezi lidmi se často stává, že dochází k neporozumění mezi vedoucím předmětu (zákazníkem) a SP týmem (tým vývojářů). Problém spočívá v tom, že zákazník zná doménu výborně, ale SP tým se teprve začíná s doménou seznamovat. Z důvodu neúplné znalosti domény lidé v týmu často chápou věci jiným způsobem, než byly myšleny. Tato skutečnost způsobuje chyby v kódu, problémy s pojmenováním, často vede k chybné implementaci a zbytečně strávenému času. Abych mohl zlepšit pochopení domény, za prvé jsem vytvořil slovník pojmů a procesů pro naši doménu, viz podkapitola 3.3, za druhé jsem zavedl jmenné konvence a některé z Domain Driven Design (DDD) postupů. Správné použití Domain Driven Design je složité a není vhodné pro neustále se měnící složení týmů. Proto jsem se rozhodl použít jen některé jednoduché postupy, které dovolí zjednodušit chápání systému, ale studenti nebudou nuceni používat něco, co vůbec nechápou, viz následující podkapitola 3.1.

3.1 Základní koncepty Domain Driven Design (DDD)

Domain Driven Design (DDD) je přístup návrhu aplikace, který se snaží navrhnout softwarový program takovým způsobem, aby výsledek představoval model systému či procesu v realitě. Tento model je vyvíjen spolu s takzvaným **doménovým expertem**, který může vysvětlit, jak se má chovat systém v realitě. Rozumí všem procesům v daném problému a umí vždy rozhodnout, jak má vypadat model. [19] Našim doménovým expertem je Ing. Jiří Hunka, který se zabývá výukou předmětu BI-DBS a zná veškeré procesy odehrávající se během výuky. Navíc má zkušenosti s vývojem a provozem aplikací.

3.1.0.1 Tři pilíře DDD:

- Společný jazyk – jednoduchý jazyk orientovaný na byznys, který je vytvořen pro konkrétní problém a neobsahuje technické pojmy či koncepty. Definuje slovník pojmů, definic, popisu procesů pro tvorbu modelu. Hlavním účelem tohoto jazyka je zabránění vzniku nedorozumění mezi všemi účastníky projektu včetně doménového experta. Musí být jednoznačný a přesný alespoň v ohraničeném kontextu. [20, 21]
- Ohraničený kontext – reprezentuje subdoménu, která může mít svůj vlastní společný jazyk, architekturu a implementaci. Jeden pojem v různých ohraničených kontextech může znamenat úplně odlišné věci. To má následující výhody[20]:
  - Každý pojem je v rámci subdomény definován jednoznačně.
  - Rozdělení modelu na menší části dovoluje jednodušeji navrhovat softwarové moduly.
  - Zjednodušuje integraci s jinými částmi systému, externími komponentami či legacy kódem[1].
- Kontextová mapa – jednoduchý diagram, který reprezentuje mapu propojení mezi subdoménami a způsob komunikace mezi nimi. [20]

3.1.0.2 Implementace

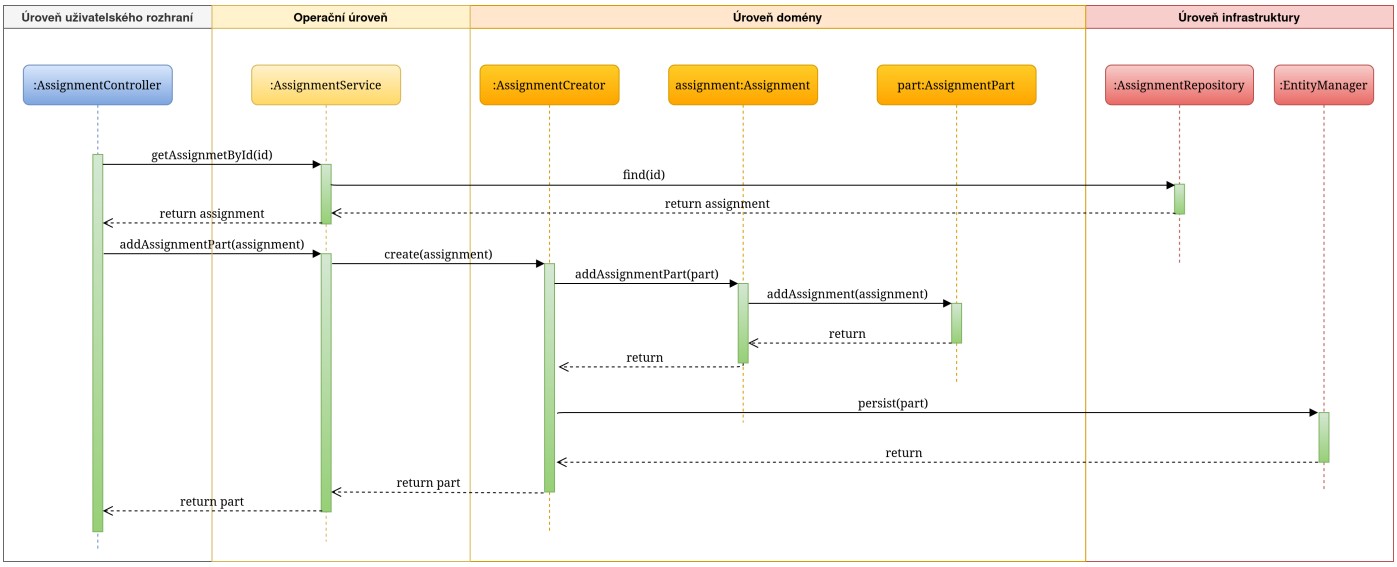
Pojmy, koncepty a procesy popsané společným jazykem tvoří základ objektově orientovaného návrhu aplikace. DDD poskytuje jasné pokyny, jakým způsobem by měly objekty spolu interagovat, a rozděluje objekty do následujících kategorií[19]:

- Value objects – neměnné objekty, které nemají identitu a mohou být využité pro několik entit. Tyto objekty zůstávají neměnné po celou dobu svého života a mohou mít podčásti. Například objekt představující datum se může skládat ze dne, měsíce a roku.
- Entities – objekty, které mají svou identitu. Například „Otázka“, „Zadání“, „Test“ jsou entity. Víme, že dva testy se stejným jménem nebudou představovat stejný objekt, protože stejný test píše více lidí a identita testu je stanovená jak samotným testem, tak i člověkem, který ho píše.
- Agregates – struktury objektů mající jeden kořen. Přístup k entitám uvnitř této struktury je dovolen jen pomocí kořene. Například v naší doméně máme entitu „Zadání“, která vlastní jednotlivé podčásti zadání. Tyto podčásti mohou být implementované jako součást agregátu „Zadání“ a libovolný přístup k nim se bude provádět pomocí entity „Zadání“, protože objekt „Zadání“ je kořenem.

[1] Legacy kód – starší zdrojový kód, který je převzatý od někoho jiného a nebo ze starší verze softwaru. Může to být jakýkoliv kód, kterému moc nerozumíme. [22]

DDD také používá pojem „služba“. Služba je komponenta softwaru odpovídající za nějakou konkrétní byznys funkci či operaci. Na rozdíl od entit a hodnotových objektů nedrží stav, ale uchovává byznys logiku konkrétního procesu. DDD dělí všechny služby do několika úrovní:

- Úroveň uživatelského rozhraní – definuje, jakým způsobem může uživatel interagovat se systémem. V našem případe veškerá komunikace probíhá pomocí REST API (protokol HTTP). Proto je u nás tato úroveň zodpovědná za zpracování požadavků, zpracovaní vzniklých chyb a tvorbu odpovědí. Tato úroveň by neměla obsahovat žádné jiné funkce. [23] Kdyby bylo potřeba změnit způsob komunikace se systémem, jednoduše se vymění pouze tato úroveň. Framework Symfony má pro ni skvělou podporu, a proto ji můžeme jednoduše implementovat pomocí Symfony Controllers.
- Operační úroveň – definuje operace a procesy, které by měl řešit systém, a rozloží je na operace doménové úrovně. Neměla by obsahovat složitou logiku a uchovávat důležitou informaci, ale může obsahovat informaci o průběhu nějakého procesu či operace. [23]
- Úroveň domény – definuje doménu, zde je obsažená veškerá byznys logika aplikace. Tato úroveň je hlavní algoritmickou částí systému. [23]
- Úroveň infrastruktury – poskytuje technickou podporu pro vrchní úrovně: kontinuitu existence objektů, komunikaci na operační úrovni, uložení a vyhledávání objektů apod. [23] V našem případe tyto činnosti zajišťuje Symfony framework pomocí ORM, DI a dalších technologií.



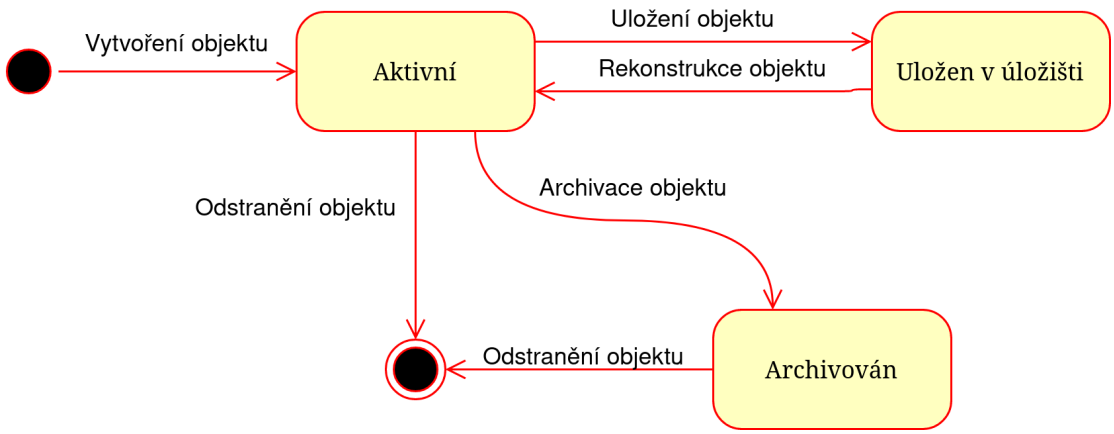
Obrázek 3.1 Rozdělení služeb do jednotlivých úrovní

Na obrázku 3.1 lze vidět proces tvorby části testového zadání, který je složitou operací, proto je veškerá byznys logika umístěna na úrovni domény. Pokud by vytvoření objektu vyžadovalo jen volání konstruktoru, je možné toto implementovat uvnitř služby AssignmentService pro zjednodušení kódu. Ale obecně se doporučuje pro každé vytvoření objektu používat „vytvářecí návrhové vzory“ [24]. Zadání je zde vytvářeno dle jednoho z typů: text, diagram, obrázek, normalizace, databáze. AssignnmentCreator je rozhraním skrývajícím implementaci konkrétní třídy tvořící objekty pro konkrétní typ. Více o doménovém modelu je možné nalézt v mé bakalářské práci: „Refaktoring testové části backendu portálu dbs.fit.cvut.cz“ [1].

3.1.0.3 Životní cyklus objektů

Objekty v softwarovém systému mají svůj životní cyklus: objekt je vytvořen, může chvíli existovat, pak může být změněn a na konci zanikne. Také můžeme objekty archivovat. Některé objekty jsou potřebné jen pro výpočet, tedy jsou vytvořeny s voláním konstruktoru a zaniknou po dokončení vykonávané operace. Jiné představují složitější konstrukce, jako jsou entity a hodnotové objekty. Tyto objekty mají mnohem složitější životní cyklus, jsou mezi sebou složitě propojené a vyžadují pro uložení úložiště. Pro vytvoření objektů se doporučuje používat továrny (Factories), pro uložení a obnovení úložiště (Repositories). Každý složitější objekt by měl mít svou továrnu (službu), která ví, jakým způsobem je tento objekt tvořen a co je k tomu potřeba. Pro uložení objektů se doporučuje používat služby (Repositories), které zapouzdří veškerý kód týkající se vyhledávaní a uložení objektů. Ostatní služby komunikují s těmito službami a vůbec nepotřebují vědět o tom, jakým způsobem se objekt zrekonstruuje nebo uloží do úložiště. [23]

V našem případě se tato doporučení hodí, protože budeme používat distribuovanou databázi. Každá mikroslužba bude mít vlastní databázi, viz podkapitola 3.2.2. Pro získání objektů se použije podobná služba, která dovolí zapouzdřit komunikaci s jinými mikroslužbami. Stavový diagram životního cyklu objektu lze nalézt na obrázku 3.2.



Obrázek 3.2 Životní cyklus objektu

3.1.0.4 Výhody použití DDD:

- Zaměření na toho, kdo rozumí procesům v byznyse. Jsou to lidé, kteří se v této sféře pohybují a každodenně ji využívají. Díky nim je přizpůsoben návrh tak, aby lépe odpovídal doméně a procesům z této domény a také aby byl výsledný návrh pochopitelný pro člověka z této domény. Když je návrh ve výsledku odsouhlasen doménovým expertem, je menší pravděpodobnost, že programátoři vytvoří chybné řešení, které by neodpovídalo požadavkům byznysu. [20]
- Oddělení zodpovědností: doporučuje se soustředit se v jeden okamžik pouze na jednu poddoménu. Tato poddoména může mít svoje vlastní úkoly, postupy, procesy, terminologie. Také může mít svého vlastního doménového experta, který se nejlépe vyznává v této části byznysu. [20] Rozdělení aplikace dle různých poddomén přináší:
  - Flexibilitu – lze snadno upravovat části projektu bez vedlejších účinků.
  - Lepší strukturu a organizaci kódu.
  - Snadnost testování.

- Výsledný software, který tvoříme, je blíže k doméně, a tedy blíže k zákazníkovi. [20]
- Lepší komunikace mezi byznysem a vývojáři pomocí společného jazyka. [20]

3.2 Návrhové vzory pro implementaci mikroslužeb

3.2.1 Dekompozice

Jelikož řeším rozdělení monolitické aplikace do mikroslužeb, potřebuji použít nějaký způsob dekompozice. Obecně existuje několik způsobů, jak nějaký problém dekomponovat. V tomto případě jsem pro rozdělení použil návrhové vzory „Dekompozice dle poddomény“ a „Dekompozice dle byznys požadavků“, viz sekce 3.2.1.1 3.2.1.2. Tyto způsoby jsou podobné, a proto budu vytvářet poddomény zaměřené na konkrétní byznys proces. Pro přehled uvádím několik dalších návrhových vzorů a zdůvodním volbu toho či jiného návrhového vzoru, který budu používat pro návrh architektury nového DBS portálu.

Každá z mikroslužeb by měla být dostatečně malá, aby ji vývojový tým do 5 lidí zvládl implementovat za stanovenou dobu. Doporučuje se pro návrh používat princip z objektově orientovaného návrhu: Princip jednotné odpovědnosti (SRP). Aplikace tohoto přístupu na návrh mikroslužeb dovolí navrhovat mikroslužby zaměřené na konkrétní byznys požadavek, kde každá z nich definuje malou část úzce souvisejících funkcí systému. [25]

3.2.1.1 Dekompozice dle byznys požadavků

Tento návrhový vzor dělí systém z pohledů obchodních procesů konkrétní organizace. Obchodní proces je něco, co byznys dělá v rámci vytvoření svého produktu. [25] Například:

- Správa objednávek – zodpovídá za vyřízení objednávek.
- Doručení produktu – zodpovídá za doručení produktu konkrétnímu zákazníkovi.
- Správa zákazníků – zodpovídá za evidenci zákazníků.

Ve výsledku se bude dělení na mikroslužby provádět dle konkrétních obchodních procesů. Získáme pak následující mikroslužby: mikroslužbu pro správu objednávek, mikroslužbu pro doručení produktu zákazníkovi, mikroslužbu pro evidenci zákazníků. [25] Tento návrhový vzor má následující výhody:

- Stabilní architektura, protože obchodní procesy jsou stabilní a málokdy se celkově mění.
- Organizace vývojových týmů je udělaná tak, aby produkovaly obchodní hodnotu, a ne technické funkce.
- Mikroslužby jsou slabě propojené (nízká provázanost) a je zajištěna vysoká soudržnost.

3.2.1.2 Dekompozice dle poddomény

Definuje mikroslužby odpovídající poddoménám dle Domain-Driven Design (DDD) principů, viz podkapitola 3.1. Doména každého velkého systému se skládá z poddomén. Každá poddoména zodpovídá za jinou část byznysu. Poddomény jsou klasifikované následovně:

- Jádro (Core) – nejdůležitější část domény, která je nezbytná pro fungování byznysu.
- Podpůrná (Supporting) – souvisejí s tím, co podnik dělá, ale nejsou tak důležité. Mohou být implementované interně nebo i externím dodavatelem.

- Obecná (Generic) – není specifická pro organizaci a může být implementovaná pomocí běžných existujících systémů na trhu.

Výhody této dekompozice jsou skoro stejné jako u předchozí, protože výsledný návrh bude velmi podobný. Rozdíl je tedy jen v celkovém pohledu na byznys. My se budeme dívat na systém z pohledu DDD, proto budeme používat tento návrhový vzor. [26] Většina mikroslužeb definovaných v této práci představuje jádro systému, tedy jsou typu „Core“. Výjimkami je pouze několik z nich, které se zabývají sběrem statistik a rozesíláním notifikací. Pro tyto mikroslužby by se dala použít nějaká existující řešení na trhu, proto je můžeme klasifikovat jako typ „Generic“, viz podkapitola 3.5.

3.2.1.3 Self-contained service

V rámci systému se stává, že pro vyřešení jednoho požadavku uživatele musí mikroslužba zavolat několik jiných části systému a něco tam vytvořit, nebo validovat nějaké informace. Problém nastává v případě, že je některá z mikroslužeb nedostupná, nebo přetížená. Uživatelovi se vždy zobrazí chyba, pokud je alespoň jedna z vnitřních služeb nedostupná, což snižuje celkovou dostupnost systému. Klíčovou nevýhodou je tady synchronní interní volání. Alternativou k synchronnímu volání je použití návrhových vzorů CQRS a Sága. Podrobně se lze o těchto návrhových vzorech dozvědět v sekcích 3.2.2.5 a 3.2.2.3.

Tento návrhový vzor umožňuje zbavit se synchronních voláních, tj. navrhnout mikroslužbu tak, aby uživatel nečekal na odpověď od interních mikroslužeb. To můžeme udělat dvěma způsoby: implementovat interní mikroslužby jako moduly naší mikroslužby, nebo spolupracovat s těmito mikroslužbami s použitím návrhových vzoru CQRS a Sága.

Dle návrhového vzoru CQRS se používá replika dat z jiné mikroslužby, která dovoluje přistoupit k její datům bez posílání požadavku. Tato replika je pouze pro čtení. Pro asynchronní zápis dat do interních mikroslužeb se používá návrhový vzor Sága. Ten dovoluje poslat asynchronní požadavek, který se uloží do fronty a zpracuje se v okamžiku, kdy bude tato služba bude dostupná. [27]

Tento vzor má následující výhody a nevýhody:

- Hlavní výhodou je zvýšení dostupnosti a doby odpovědí na požadavek. [27]
- Nevýhodou tady bude používání návrhových vzorů CQRS a Saga, které způsobí zvýšení využití zdrojů na serveru a obecně složitější architekturu. Navíc bude kvůli použití Ságy méně přímočaré API. [27]

3.2.1.4 Služba pro tým

V případě, že máme dlouhodobě několik týmů vývojářů, můžeme rozdělit architekturu tak, aby konkrétní část systému spravoval jeden tým. Tyto týmy by měly být na sobě co nejméně závislé. Každý tým by odpovídal za konkrétní byznys funkci či proces. To je mnohem lepší, než kdyby pracoval na všech částech systému najednou. Tým může pracovat autonomně a nezávisle. Chtěli bychom, aby každý tým odpovídal za jednu mikroslužbu, ale může jich být i více, protože jedna mikroslužba může být pro vývojový tým 4-6 lidí příliš malá na to, aby byla využita celková kapacita týmu. Pokud jeden z týmů potřebuje provést nějakou změnu v jiné mikroslužbě, vytvoří požadavek na jiný tým, který se zabývá jejím vývojem. Další výhodou tohoto způsobu je, že pokud jeden tým dlouhodobě udržuje jednu část systému, výsledný zdrojový kód bude mít vyšší kvalitu. Tento přístup má i svoje nevýhody. Týmy mohou vyvíjet jen malou část systému, a proto nemusí vůbec pochopit celkové procesy, které probíhají v byznysu. [28]

Tento způsob jsme úspěšně použili pro vývoj v době, kdy jsme měli 2 týmy vývoje. Jeden z týmů řešil Connections mikroslužbu a druhý se zabýval Configurations mikroslužbou. To přinášelo mnohé výhody, protože každý tým věděl o své mikroslužbě všechno a nebyl problém s nepochopením nějakého procesu či funkce. V dalším semestru jsme měli pouze jeden vývojový

tým, a proto se musel zabývat celým systémem. Ve výsledku se objevily drobné problémy s nepochopením použití několika funkcí některých mikroslužeb. Přestože se je podařilo úspěšně vyřešit, snížila se produktivita týmu, protože opravování chyb zpozdilo správnou implementaci.

Proto doporučuji vedoucímu předmětu zajistit, aby, pokud je to možné, pracoval jeden tým v jeden moment pouze s jednou až dvěma mikroslužbami. Ideální je pracovat na jedné mikroslužbě, ale ve skutečnosti se mikroslužby volají navzájem. Z tohoto důvodu se bude muset často pracovat na více než jedné mikroslužbě, protože máme jenom jeden vývojový tým.

V našem případě DBS portál vždy vyvíjejí studenti, proto dochází často k výměně lidí na projektu, což nám nedovoluje plně využít tento způsob. Ale pokud se jeden tým zabývá pouze jednou mikroslužbou, zlepší to produktivitu týmu. Jestliže někdo nebude něco vědět, budou mu ostatní schopni poradit. Tady je důležitý ohraničený kontext, v rámci kterého se tým bude schopen mnohem rychleji zorientovat a začít produkovat funkční a správný kód.

3.2.2 Způsob uložení dat

Každý systém zpracovává nějaká data, a proto potřebuje nějaké perzistentní úložiště, do kterého může tato data ukládat. V rámci monolitní architektury se pro uložení dat používá jedna databáze a problém je vyřešen. Ale z pohledu mikroslužeb je proces uložení dat mnohem složitější. Z toho důvodu uvedu několik návrhových vzorů, které budou velmi užitečné pro správu dat a pro další s tím spojené problémy.

V rámci mého návrhu padla volba na návrhový vzor „Databáze pro službu“. Pro takto malý projekt by se dala použít i jedna databáze, ale u nás se na projektu vždy vyměňují nezkušení lidé, proto je velmi důležité zajistit lokalitu dat. Pokud by to nebylo zajištěno, mohlo by se pak stát, že by vývojáři ukládali data jinam, než by bylo potřeba, což by ve výsledku mohlo zase způsobit problémy s provázaností, které máme teď. Navíc by mohly nastat problémy se synchronizací nebo ztrátou dat, pokud by se omylem přistupovalo ke stejným datům z různých mikroslužeb. Proto budu používat přístup „Database server per service“ v rámci návrhového vzoru „Databáze pro službu“, který je popsaný v následující podkapitole. Vytvoření hranic je správným postupem majícím za cíl zabránit lidem vytvářet špatné vazby, které tam být nesmí. [29]

Distribuované uložení dat dovolí ukládat data nezávisle na ostatních mikroslužbách, a pokud nějaká z nich bude potřebovat data z jiné mikroslužby, bude komunikovat s touto mikroslužbou pomocí rozhraní. Další výhodou je, že nemusím využívat jen jeden typ databáze. Jedna mikroslužba může používat relační databázi, ale jiná jednoduše využije například NoSQL databázi. Nevýhodou tohoto řešení je, že klademe větší nároky na návrháře jednotných mikroslužeb. Ten musí totiž zajistit synchronizaci mezi nimi během implementací složitějších procesů probíhajících ve více než jedné mikroslužbě. Proto se v této práci budu snažit navrhnout rozdělení na mikroslužby tak, aby bylo možné jednoduše zajistit synchronizaci a nezávislost dat. Dále popíšu jednotlivé návrhové vzory, které mohou být použité při implementaci.

3.2.2.1 Databáze pro službu

Každá mikroslužba vlastní svoji databázi. To znamená, že každá mikroslužba může získávat a ukládat informace nezávislé na ostatních. Změna struktury databáze nebo typu úložiště nemá žádný vliv na ostatní mikroslužby. K této databázi může přistupovat přímo jen ta mikroslužba, která je za ni zodpovědná. Ostatní mikroslužby, pokud potřebují data uložená v této databázi, musí o ně požádat pomocí veřejného rozhraní. [29, 30] Pokud používáme relační databázi, lze tento přístup implementovat třemi různými způsoby[29]:

- Private tables per service – privátní skupina tabulek pro jednu mikroslužbu v rámci jedné databáze.
- Schema per service – pro každou mikroslužbu je přiděleno databázové schéma.

- Database server per service – každá mikroslužba má svoji vlastní databázi.

Tento přístup má následující výhody[29, 30]:

- Zajišťuje nezávislost mezi mikroslužbami.
- Dovoluje používat různé druhy databáze pro různé potřeby mikroslužeb. Například pro správu a uložení obyčejných informací můžeme použít relační databázi, ale pro uložení uživatelů a vazeb mezi nimi můžeme použít grafovou databázi Neo4j, která podporuje různé algoritmy pro vyhledávání v grafech.
- Umožňuje každé mikroslužbě mít dle potřeby různou úroveň bezpečnosti dat.
- Nabízí lepší možnosti pro škálovaní konkrétní mikroslužby.

Nevýhody distribuovaného úložiště [29, 30]:

- Složitost správy více databází různých typů.
- Implementace složitějších procesů, které probíhají nad několika různými mikroslužbami, je složitá.
- Je potřeba dodatečně zajistit synchronizaci a konzistenci dat.

3.2.2.2 Sdílená databáze

Pro všechny mikroslužby, které implementujeme, se používá jedná sdílená databáze. Každá mikroslužba může přímo přistupovat k datům, která patří jiné mikroslužbě. To je vhodné používat, pokud implementujeme nějaké složité procesy vyžadující ověření nějakých vlastnosti z velkého množství jiných mikroslužeb a potřebujeme zajistit, aby nám nikdo během této kontroly nezměnil data. Pokud jsou přístupy k databázi špatně implementované, může to způsobovat velkou závislost mezi mikroslužbami, které eliminují všechny výhody architektury mikroslužeb. [31]
Výhody sdílené databáze [31]:

- Programátor používá známé databázové transakce pro zajištění konzistence dat.
- Správa jedné databáze je jednodušší.

Nevýhody sdílené databáze [31]:

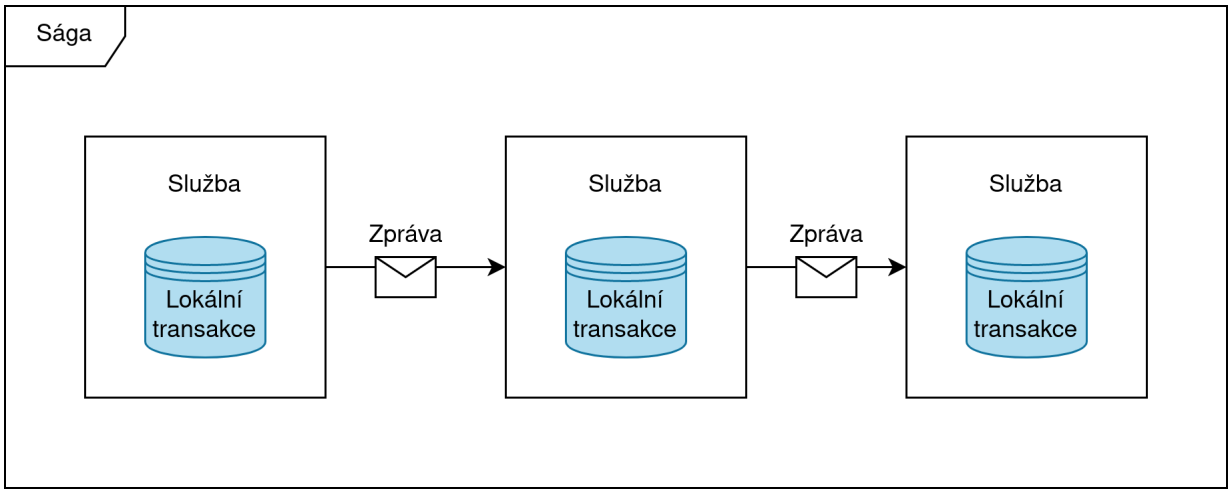
- Programátor pracuje na procesech, které ovlivňuji jiné částí systému, proto musí komunikovat s kolegy a nepoškodit jejich kód, což snižuje jeho výkon.
- Služby přistupující k jedné databázi se mohou potenciálně rušit. Například pokud jedna služba vytvoří zámek na objekt A a druhá v ten okamžik vytvoří zámek na objekt B. A pokud první bude chtít vytvořit zámek na objektu A, bude muset čekat na dokončení práce druhé služby. To může vést k uváznutí (deadlock).
- Jedna databáze nemusí splňovat požadavky, které klademe na uložení a vyhledávání data.

3.2.2.3 Návrhový vzor: Sága

Pokud používáme návrhový vzor „Databáze pro službu“, potřebujeme pro složité transakce probíhající v několika mikroslužbách způsob, kterým bychom mohli zajistit ACID vlastnosti. Tyto problémy řeší návrhový vzor Sága. To je způsob zajištění konzistence a synchronizace dat napříč mikroslužbami během provádění složitého procesu. Je to posloupnost lokálních transakcí, které jsou prováděny v jednotlivých mikroslužbách a po provedení posílají volajícímu odpověď informující o svém stavu (informují, že je potřeba udělat další krok, který závisí na stavu proběhlé

transakce). Pokud transakce selže, Sága opraví původní stav aplikace, který byl do provedení těchto operací. [32] Jednoduše řečeno je to posílání zpráv mezi službami, znázorněno na obrázku 3.3. Iniciátor tohoto procesu se musí nějakým způsobem dozvědět o výsledcích tohoto volání. Obecně existují tři způsoby:

- Synchronní – iniciátor čeká na dokončení Ságy, tj. až dostane schválení nebo odmítnutí všech požadavků od jiných služeb. [33]
- Event (asynchronní) – volající hned dostane od iniciátora zprávu s identifikátorem informací. Až bude proces dokončen, iniciátor vytvoří a pošle událost (např. websocket, web hook atd.) volajícímu. [33]
- Pooling (asynchronní) – iniciátor hned odpoví volajícímu a pošle informaci o tom, jak získat potřebná data, až bude proces dokončen. Volající se pak periodicky dotazuje na server a čeká, až bude operace dokončená. [33]



Obrázek 3.3 Komunikace mezi mikroslužbami pomocí návrhového vzoru Ságа

Existují dva přístupy, pomocí kterých lze implementovat návrhový vzor Sága:

- Orchestrace – definuje se jeden centralizovaný koordinátor, který zpracovává všechny transakce a říká účastníkům, jaké transakce by se měly provádět a kontroluje jejich splnění. [32]
- Choreografie – účastníci si mezi sebou vyměňují zprávy a publikují události, které spouští místní transakce v jiných službách bez jediného centralizovaného koordinátora. [32]

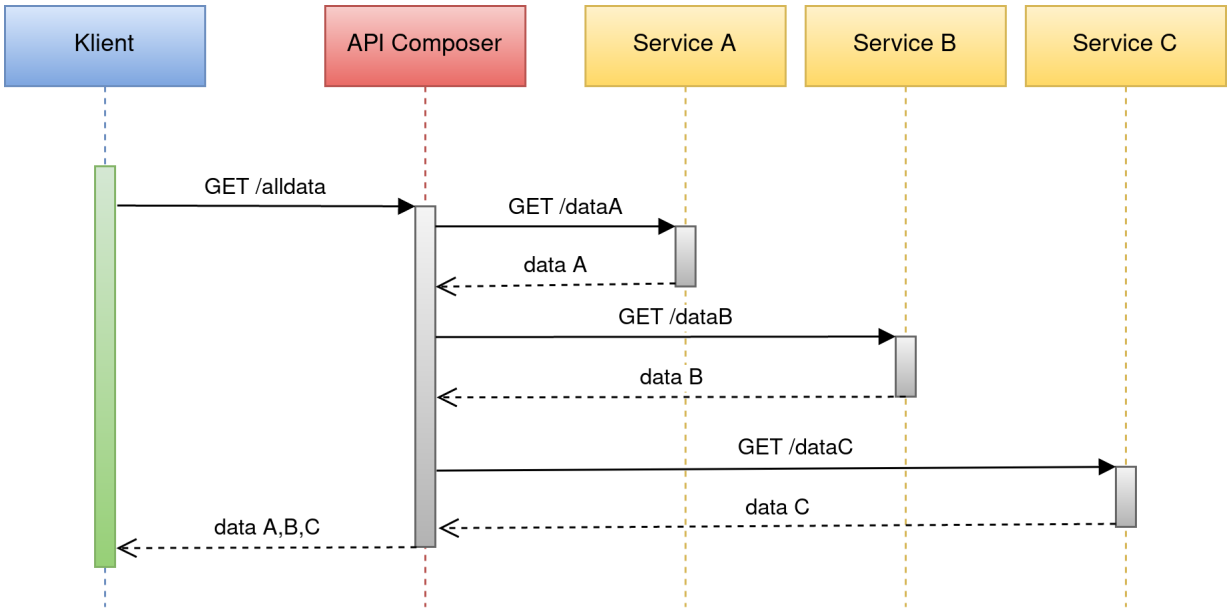
Hlavní výhodou tohoto návrhového vzoru je zajištění konzistence dat napříč více mikroslužbami během složitého procesu bez použití distribuovaných transakcí. Navíc umožňuje implementovat proces vrácení provedených změn. Nevýhodou tohoto vzoru je složitost implementace. Programátor musí zajistit správnost probíhání procesů a také vrácení původních změn. Rovněž se mohou objevovat různé anomálie, které vznikají špatným návrhem Ságy a mohou způsobovat velké problémy s daty. V případě jeho implementace se doporučuje dobře navrhnout veškeré procesy a také implementovat monitoring probíhání těchto procesů. [32, 33]

3.2.2.4 Složení API

Pokud používáme distribuované úložiště pro mikroslužby, potřebujeme někdy data z více mikroslužeb najednou. To můžeme implementovat pomocí skladatele API (API Composer). To je

jednoduchá služba, která se zabývá dotazováním několika mikroslužeb. Klient místo toho, aby volal přímo každou mikroslužbu, zavolá skladatele. Tento skladatel zavolá všechny služby obsahující potřebná data, získá odpovědi, sjednotí výsledky ze všech mikroslužeb a následně je pošle klientovi.

Skladatel pomáhá nejen zjednodušit rozhraní pro klienty, může i snížit latenci během zobrazení dat. Pro získaní všech dat musí občas klient počkat, až se načtou všechna data, sjednotit je a poté je může zobrazit. Pokud použijeme skladatele, nepotřebuje klient vědět, že jsou jeho data z různých služeb. Proto klient nemusí vytvářet plno požadavků na různé mikroslužby, navíc dovoluje měnit skryté rozhraní bez ovlivnění klientů. [34, 35] Diagram znázorňující průběh komunikace můžete vidět na obrázku 3.4.



Obrázek 3.4 Komunikace pomocí API Composer

Výhody tohoto návrhového vzoru jsou následující [34, 35]:

- Skrývá informaci o původu dat a složitosti spojené s jejich sjednocením a zpracováním.
- Zjednodušení rozhraní pro klienty.
- Dokáže skrýt špatná rozhodnutí během návrhu a umožní jednoduchou úpravu bez ovlivnění všech klientů, kteří systém používají.
- Může skrývat starší systém, ze kterého postupně přecházíme na nový.

Také to má svoje nevýhody [34, 35]:

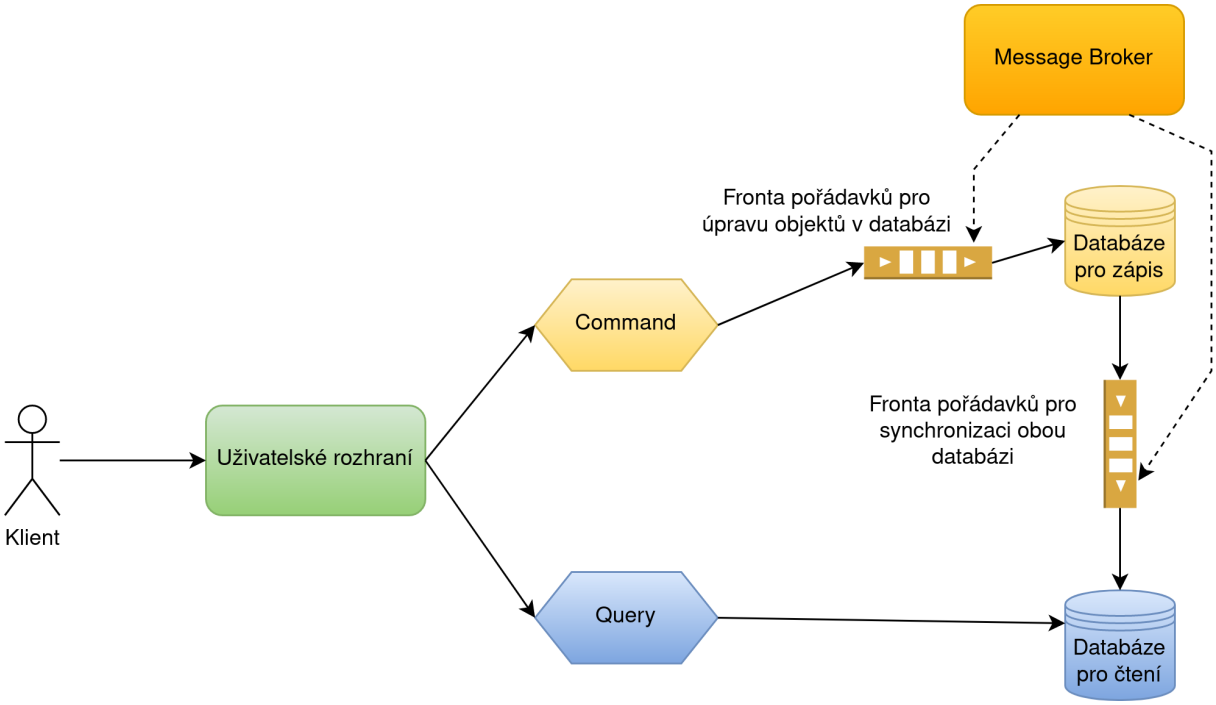
- V případě, že je velikost dat obrovská, skladatel je velmi neefektivní.
- Nedovolí zajistit konzistenci mezi daty z různých mikroslužeb.
- Skladatele je potřeba implementovat a udržovat.
- Celková dostupnost dat se může zhoršit s větším množstvím mikroslužeb.

3.2.2.5 Oddělení odpovědnosti za požadavek (CQRS)

Tato zkratka znamená Command and Query Responsibility Segregation. Je to jeden z důležitých návrhových vzorů pro dotazování mezi mikroslužbami umožňující rozdělit čtení a zápis do datového úložiště. [36]

Většinou se pro aplikace používá jedna databáze, která zpracovává jak složité databázové vyhledávání, tak i CRUD[2] operace nad touto databází. Problém nastává v případě složitější logiky aplikace, kde získávání a modifikace dat může způsobovat problémy. Například při provádění CRUD operací musíme vždy zajistit všechny validace, aby byla data validní. To může vyžadovat vytvoření zámků[3] pro některé databázové objekty, což bude způsobovat čekání jiných požadavků, a proto se může zhoršit dostupnost celého systému.

CQRS přistupuje k problému z pohledu principu oddělení zodpovědností a odděluje čtení a zápis do datového úložiště. CQRS definuje pojmy „Commands“ a „Queries“, kde „Commands“ představují požadavky pro zápis vypadající jako úlohy, které lze zpracovávat asynchronně pomocí message broker systému[4]. „Queries“ představují čtecí požadavky, které nikdy nemodifikují data v úložišti, ale pouze je čtou. Doporučuje se fyzicky rozdělit datové úložiště do dvou databází. To nám umožní zlepšit výkon aplikace, její škálovatelnost a bezpečnost. Datové úložiště pro čtení dat lze škálovat nezávisle na úložišti pro zápis. Navíc můžeme zvolit různé typy databází pro zápis a čtení dat. Synchronizaci dat mezi dvěma databázemi můžeme provádět pomocí message broker systému. [36] Znázornění tohoto návrhového vzoru lze vidět na obrázku 3.5.



Obrázek 3.5 Návrhový vzor SQRS

[2]CRUD – zkratka pro čtyři základní operace s daty: Create, Read, Update a Delete. Jsou to tyto operace: vytvoření, čtení, úprava a mazání dat, které může uživatel provádět na nějakém datovém úložišti. [37]

[3]Zámek je synchronizačním primitivem zajištujícím výhradní přístup k některému systémovému prostředku. V rámci databázových systémů to může být: datová stránka, tabulka, záznam, pole atd. [38]

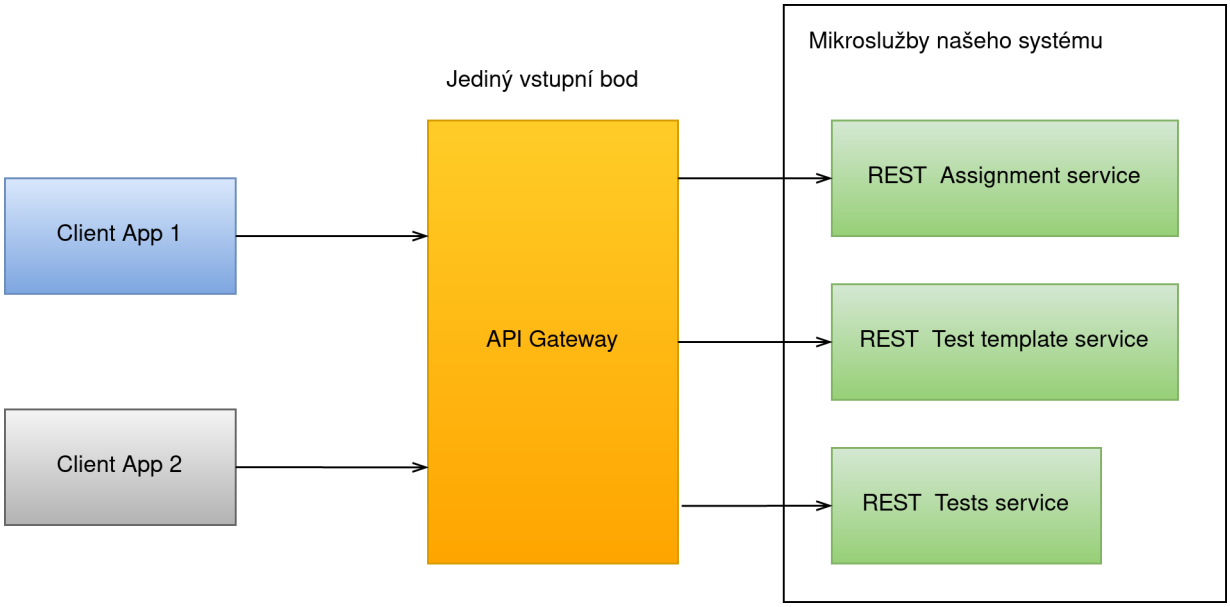
[4]Message Broker – software, který umožňuje komunikaci mezi systémy pomocí výměny formálně definovaných zpráv. Komunikace může probíhat mezi systémy vytvořenými pomocí různých technologií nebo používajícími různé způsoby zasílání zpráv. [39]

3.2.3 Externí API

3.2.3.1 API gateway

API gateway je nástroj pro správu API rozhraní mezi klientem a sadou mikroslužeb, který funguje jako reverzní proxy[5]. Tento proxy přijímá veškeré požadavky od všech klientů, agreguje potřebné informace z více různých mikroslužeb a vrací sestavenou odpověď klientovi. [41]

Pokud máme aplikaci implementovanou pomocí mikroslužeb, potřebná data mohou být uložená ve více službách, proto je někdy vhodné implementovat službu, která bude zprostředkovávat agregovaná data z více různých mikroslužeb v rámci jednoho volání. Tento druh API gateway je také složením API, které jsem popisoval v kapitole 3.2.2.4. Kromě složení API umí tento nástroj mnohem více. Může se například používat pro autorizaci uživatelů, ověření přístupů nebo omezení množství služeb pro veřejné rozhraní. Pomocí toho můžeme jednoduše implementovat sběr různých statistik o používání našeho veřejného API rozhraní. Umožňuje také implementovat jediný vstupní bod pro naše API rozhraní, který lze implementovat pomocí existujících řešení na trhu. [41] Má to obdobné nevýhody jako složení API: může být neefektivní z pohledu výkonu, potřebujeme ho implementovat a udržovat. Na obrázku 3.6 lze vidět diagram komunikace prostřednictvím API gateway nástroje.

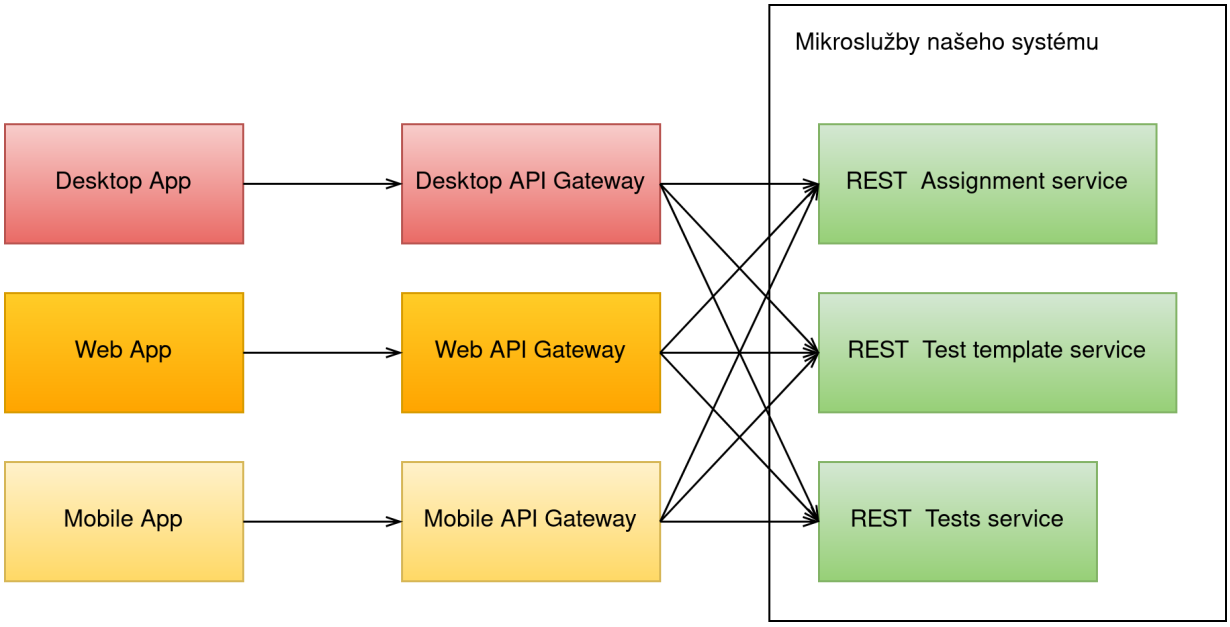


Obrázek 3.6 API Gateway

3.2.3.2 Backends for frontends

Tento návrhový vzor je variantou API gateway, která pro každý druh klienta vytvoří samostatnou proxy službu umožnující zapouzdření mikroslužeb. Místo jediného vstupního bodu jich budeme mít více, a to pro každý typ externí aplikace. Například můžeme mít jednu proxy pro desktop aplikaci, druhou pro webový prohlížeč a třetí pro mobilní aplikaci. To nám umožní rozdělit zodpovědnost a zpřístupnit pouze ty funkce a ta data, která potřebuje konkrétní klient.

[5]Reverzní proxy server je systémem, který funguje jako prostředník mezi klientem a některou sadou systémů. Přebírá všechny požadavky od klientů a směruje je na jednotlivé systémy dle konkrétních podmínek. Může také zpracovávat nějaké požadavky sám, záleží na jeho konfiguraci. Reverzní proxy může zajišťovat různé druhy činnosti: bezpečnost, šifrování, vyvažování zátěže (load balancing), cache, komprese. Příkladem takových systémů jsou: Traefik, Nginx a další. [40]



Obrázek 3.7 Backends for frontends

To se doporučuje používat jen v případě, že máme více různých druhů klientů, kteří potřebují provádět různé složité agregace mezi službami. Nevýhodou je, že potřebujeme udržovat více aplikací, navíc nám to zvýší zatíženost serveru. Proto je před implementací tohoto vzoru potřeba zvážit, jestli nám to přinese výhody, nebo ne. [42] Diagram tohoto návrhového vzoru můžeme vidět na obrázku 3.7.

3.2.4 Bezpečnost

Pro každou aplikaci potřebujeme zajistit bezpečnost jejích dat, aby se každý uživatel mohl spolehnout na to, že jsou jeho data v bezpečí a může k nim kdykoliv přistoupit. Proto potřebujeme zajistit autentizaci[6] a autorizaci[7] uživatelů. V monolitní aplikaci DBS portál se používá autentizace a autorizace implementovaná pomocí Nette frameworku. Pro API rozhraní našich mikroslužeb to použít nelze, proto potřebuji nějaký jiný způsob přihlášení, který by byl vhodný pro mikroslužby. Dále uvedu několik způsobů přihlášení k API rozhraní, jejich výhody a nevýhody. Nakonec se podívám na způsob přihlášení do našich mikroslužeb, který jsem vytvořil, a vysvětlím důvody jeho použití.

3.2.4.1 Basic HTTP Authentication

Jedná se o nejjednodušší způsob autentizace uživatelů pro API rozhraní. Používá se „base64“ formát pro kódovaní přihlašovacích údajů, které se poté ukládají do hlavičky Authorization HTTP protokolu a kontrolují se na serveru. Nevýhodou je to, že se tady heslo posílá v každém požadavku v nezašifrované podobě. To není dostatečně bezpečné. Pokud by se někde stalo, že data putují v nešifrované formě (nepoužívá se protokol HTTPS), nic nebrání útočníkovi získat hesla uživatelů.

[6] Autentizace je proces ověření identity konkrétního uživatele. Uživatel musí prokázat, že je to opravdu on. Obvykle můžeme autentizaci provádět pomocí uživatelského jména a hesla. Autentizovat se dá nejen uživatele, ale i jiné objekty, např. přístup k nějakému API rozhraní. [43]

[7] Autorizace je proces ověření oprávnění uživatele provést nějakou akci či získat nějaká data. Obvykle probíhá po autentizaci. [43]

Navíc neumožňuje vícefaktorovou autentizaci. [44, 45] Komunikace mezi klientem a serverem je jednoduchá [45]:

1. Klient si zkusí zobrazit stránku vyžadující autentizaci, bez zadání hesla a uživatelského jména.
2. Server odpoví HTTP kódem 401 a poskytne informace o tom, že se klient snaží přistoupit k chráněné oblasti.
3. Klient vyplní přihlašovací údaje a znovu se pokusí přistoupit.
4. Server ověří přihlašovací údaje, a pokud jsou správné, dovolí přístup.

3.2.4.2 API Key Authentication

Tento přístup je variantou „Basic HTTP Authentication“, který používá strojově generované řetězce pro vytvoření jedinečných párů skládajících se z identity uživatele a přístupového API tokenu. Místo toho, abychom pokaždé posílali uživatelské jméno a heslo, posíláme jen tento generovaný řetězec. Výhodou je, že neposíláme heslo v každém požadavku. Proto pokud by došlo k nějakému incidentu, získá útočník jen tento řetězec. Navíc můžeme zrušit všechny přihlašovací tokeny a tím zamezit přístup. V případě basic autentizace to udělat nemůžeme, uživatel si musí svoje heslo měnit sám. [44]

3.2.4.3 JWT Token

JSON Web Token (JWT) je otevřený standard (RFC 7519) definující jednoduchý a bezpečný způsob přenosu informací pomocí objektu JSON. Tyto informace jsou bezpečné a můžeme jim důvěřovat, protože jsou digitálně podepsané[8]. Tento token můžeme podepsat pomocí tajného klíče nebo páru veřejného/soukromého klíče. Pokud je potřeba, tak můžeme tyto tokeny šifrovat a tím skrývat informaci, kterou nesou. [47]

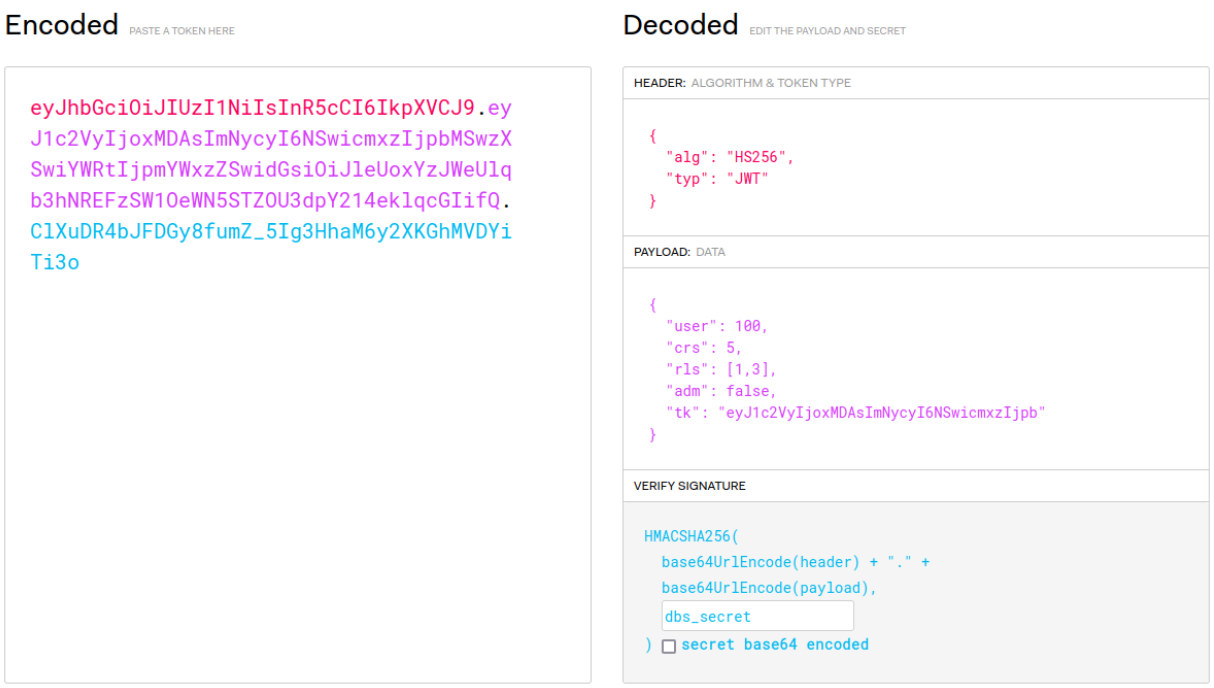
JWT token má tři části: hlavičku, tělo a podpis. Typicky má formát: hhhhh.ttttt.ppppp. V rámci hlavičky se obvykle posílají dvě hodnoty: typ tokenu a algoritmus, který se používá pro podpis. Tělo se skládá z uživatelsky definovaných hodnot, které potřebujeme přenášet mezi systémy. Typicky to bývá identita uživatele, jeho oprávnění apod. Doporučuje se vytvářet krátké názvy klíčů a přidávat do těla tokenu jen nutné hodnoty, protože se tyto informace vždy posílají s každým požadavkem. Může se zdát, že několik informací navíc nebude mít žádný vliv, ale pokud se bude posílat několik tisíc požadavků za vteřinu, může to mít dopad na výkon. Navíc webové servery mohou mít omezení na velikost HTTP hlaviček, což může způsobit problémy. [47, 48] Existují už standardně definovaná jména klíčů, například „exp“ – doba expirace, „typ“ – typ tokenu a další. [48] Pokud dobře nastavíme tělo tohoto tokenu, na serveru můžeme eliminovat získání informací o uživatelích z databáze a tím zvýšit odezvu systému.

Po sestavení se hlavička a tělo zakóduje do base64 formátu zvlášť a spojí se s digitálním podpisem pomocí teček, aby byl dodržen formát popsaný výše. Digitální podpis se vytvoří z hlavičky a těla zakódovaného do base64 formátu pomocí zvoleného algoritmu. [47, 49]

Aby byl server schopen tento token získat, používá se hlavička Authorization HTTP protokolu s použitím Bearer schématu. Například: Authorization: Bearer <token>. Server si tuto hlavičku přečte, dekóduje informaci a ověří podpis tohoto tokenu. Ověření probíhá následovně: nejprve se vygeneruje nový podpis z hlavičky a těla tokenu, který dostaneme z požadavku. Pak se porovná tento podpis s podpisem zaslaným v požadavku. Pokud jsou tyto podpisy stejné, znamená to, že byl tento token vydán naším serverem a můžeme dovolit přístup. Jinak přístup nedovolíme. Pokud nepřidáváme dodatečné šifrování, měly by být tyto tokeny platné relativně krátkou dobu, po jejímž uplynutí by se měly stát neplatnými. Obvykle to bývá 60 minut, může

[8] Digitální podpis je matematická technika, která se používá pro ověření autentičnosti a integrity zprávy, digitálního dokumentu či softwarového programu. Může to být obyčejný hash, který je vytvořen z dané zprávy. Tento podpis nám může poskytnout důkaz o původu a identitě dat. [46]

to být i méně, pokud jsou tyto informace důležité, tj. pokud by útočník nějakým způsobem tento token odchytil, měl by přístup k systému pouze krátkou dobu, což velmi snižuje pravděpodobnost způsobení škody. Komunikace mezi klientem a serverem probíhá pomocí protokolu HTTPS, je tedy šifrovaná. Jediná možnost, kde by token mohl uniknout, je počítač klienta, což nemůžeme ovlivnit. Možnost, že se někdo dostane na server a získá tam práva privilegovaného uživatele, se považuje za málo pravděpodobnou. [47, 49] Příklad JWT tokenu můžeme vidět na obrázku 3.8.



Obrázek 3.8 Příklad JWT tokenu. Zdroj: JSON Web Token Debugger

3.2.4.4 OAuth 2.0

OAuth 2.0 je autorizační protokol (standard RFC 6749), který dovoluje aplikacím třetích stran získat omezený přístup ke zdrojům nějaké aplikace. Tedy konkrétní uživatel, který používá systém A, dává oprávnění systému B používat jeho data ze systému A, aniž by musel v systému B zadávat svoje přístupové údaje k systému A a tím dávat neomezený přístup k jeho datům bez možnosti zrušení. V případě OAuth protokolu je přístup k datům omezen pomocí tzv. „scopes“ – ty definují, ke kterým datům může systém B přistoupit. Tyto přístupy uvede uživatel v době potvrzení souhlasu. Velkou výhodou je, že může uživatel tyto přístupy kdykoliv zrušit pomocí autorizačního serveru a systém B pak nebude schopen k těmto datům přistoupit. Mimo jiné se také nabízí možnost obnovení přístupového tokenu pomocí tzv. obnovovacího tokenu. Tento token je obvykle generován spolu s přístupovým tokenem a po uplynutí doby jeho platnosti ho můžeme pomocí obnovovacího tokenu obnovit. Tento protokol umožňuje přístup jak webovým aplikacím, tak i aplikacím na straně serveru či desktop aplikacím. [50, 51]

3.2.4.5 OpenID Connect

OpenId Connect je jednoduchý autorizační protokol, který je založen na OAuth 2.0 frameworku. Tento protokol vylepšuje základní OAuth 2.0 protokol a umožňuje jednotné přihlášení (SSO)

do více různých služeb. Tyto služby jsou pak schopné používat nejen identitu uživatele, ale i nějaká uživatelská data, ke kterým uživatel schválil přistup. Tato data se obvykle ukládají do tzv. ID tokenu. Do JWT tokenu ukládáme informaci, která je vyžádána službou a poté schválena uživatelem. Tedy spolu s identitou uživatele můžeme získat nějaké dodatečné informace, např. jeho jméno, příjmení, e-mailovou adresu apod. [52]

Celkové přihlášení probíhá velmi podobně jako u OAuth protokolu. Hlavní rozdíl je v tom, že se generuje ID token obsahující veškeré informace o uživateli. Tento token je obvykle digitálně podepsán, tedy server vždy může zkontrolovat správnost těchto údajů. Klient obdrží potřebné informace o uživateli, tedy nemusí posílat dodatečné požadavky pro získání uživatelských údajů. Další výhodou je to, že je založen na OAuth 2.0 protokolu, a proto nabízí veškeré jeho výhody. Například umožňuje autorizaci z různých druhů zařízení – mobilní či desktop aplikace, webové prohlížeče apod. [52, 53]

3.2.4.6 Shibboleth

Shibboleth je projektem s otevřeným zdrojovým kódem poskytujícím službu Single Sign-On (jednotné přihlášení). Tedy pomocí jednoho přihlášení může uživatel využívat více různých služeb a nepotřebuje pro každou zadávat svoje přístupové údaje. Shibboleth se používá pro řízení přístupů k online zdrojům a především pro korporativní prostředí, kde chce každá z organizací sdílet svoje data s ostatními. Bývají to nejčastěji vysoké školy či státní instituce. Například vysoké školy tak mohou umožňovat přístup ke svým materiálům uživatelům spolupracujících organizací. Takové instituce se sdružují do skupin, které se označují jako federace. V rámci naší univerzity máme federaci, která se jmenuje cvutID. Hlavní výhodou Shibboleth je, že tento nástroj nabízí možnost poskytovat podporu pro SSO mimo organizaci uživatele a zajišťuje bezpečnost osobních údajů tohoto uživatele. [54–56] Obsahuje několik prvků [54, 55]:

- Webový prohlížeč – představuje klienta, který chce získat data.
- Identity Provider (IdP) – poskytovatel identity uživatele, jeho účelem je autentizace uživatele.
- Service Provider (SP) – webový server obsahující chráněná data, ke kterým potřebuje přistoupit klient.

Komunikace mezi IdP a SP probíhá pomocí posílání zpráv v jazyce SAML[9]. Informace o uživatelích se předávají pomocí XML souborů. Tyto zprávy jsou digitálně podepsané, čímž zaručují důvěryhodnost a bezpečnost. Nevýhodou je to, že jazyk SAML byl navržen pro webové aplikace, a proto není vhodný například pro mobilní aplikace. Také jeho konfigurace není tak jednoduchá. [54, 55, 57]

3.2.5 Autorizace uživatelů v DBS portálu

Abych mohl autentizovat uživatele, potřebuji nějakým způsobem získat jeho identitu. V aktuální monolitní aplikaci se přihlášení provádělo pomocí Shibboleth, viz sekce 3.2.4.6. Tento způsob není moc vhodný pro API rozhraní našich mikroslužeb. Pro RESTful API rozhraní je vhodnější použít například protokol OAuth, který je flexibilnější a jednodušší k implementaci. Také nepotřebuji sdílet přístup napříč organizacemi, ale jen získat uživatelskou identitu, proto zvolím protokol OAuth. [57]

[9] SAML (Security Assertion Markup Language) – bezpečný, široce používaný protokol pro sdílení osobních údajů napříč systémy. Pro přenos dat se používají XML datové struktury a jednoduchý HTTP nebo SOAP protokol. [57]

Pro naše účely použiji FIT Auth server[10], který umožní vygenerovat přihlašovací token pomocí protokolu OAuth. Tento token si mohu uložit do úložiště na serveru a pak pomocí porovnání validovat přístup. Identitu uživatele mám a mohu s ní pracovat. Nastává tady jiný problém: zbytečné načítání dat o uživatelích z databáze. Tento token sice řekne, co je to za uživatel, ale není schopen říct, jaké oprávnění v rámci našeho systému tento uživatel má. Neumožňuje tedy jednotné přihlášení. Je nutné se při každém požadavku dotazovat na Configurations a Auth mikroslužby, aby za prvé ověřil identitu a za druhé získal informace o kurzech uživatele a jeho oprávněních v rámci kurzu. Tedy každý požadavek, který pošle uživatel, vždy posílá 2 synchronní požadavky na jiné mikroslužby, aby jen získal potřebná data o uživateli. Nejde o vhodné řešení, protože to významně snižuje dostupnost celého systému. Pokud bude v jeden okamžik v systému přihlášeno 500 uživatelů, což by se mohlo stát, obdržíme minimálně 1000 dodatečných požadavků. Navíc se vždy pro každého uživatele v rámci architektury mikroslužeb vyžaduje více než jedno volání, aby se mohla zobrazit webová stránka. Proto se tento počet najednou násobně zvyšuje. Potřebuji tedy najít způsob dle kterého bych mohl odstranit tato zbytečná volání.

Nabízí se tady použít JWT token, který umožňuje validaci bez nutnosti dodatečně volat autorizační mikroslužbu a může nést potřebná data o uživateli. Neexistuje ale žádný autorizační server vydávající tyto tokeny a umožnující jejich propojení s fakultním účtem studentů. Proto budu pro účely našeho systému potřebovat implementovat v rámci autorizační mikroslužby mechanismus vydávání tokenů, které jsou propojené s těmito účty uživatelů.

Myšlenka je následující: autentizaci uživatele provedu pomocí OAuth protokolu a autorizaci budu provádět na základě vytvořeného JWT tokenu, do kterého uložím data o uživateli, jeho rolích a kurzu. Také tam budu muset uložit dobu expirace a OAuth token pro verifikaci. Jelikož se uživatel nejčastěji pohybuje v rámci jediného kurzu, přihlašovací token bude validní jen pro tento daný kurz. Pokud by byl validní pro všechny kurzy, mohlo by se stát, že tělo JWT tokenu hodně naroste. Důvod je jednoduchý: každý uživatel má v rámci jednoho kurzu více rolí. Velikost tokenu se tedy stává závislá na datech, což je nebezpečné, protože některé servery mohou mít omezení na velikost HTTP hlaviček. Nevýhodou je také to, že se bude muset posílat více dat. To hlavně platí pro vyučující, kteří dlouhou dobu vyučují tento předmět a mají možnost přistupovat k několika kurzům.

Naše Auth mikroslužba ověří identitu uživatele a na základě ní vygeneruje přístupový token. Tento způsob autorizace je zjednodušenou verzí OpenId Connect protokolu, kterou implementuji v rámci autorizační mikroslužby. Navíc pokud budou mít jiné mikroslužby možnost ověřit digitální podpis tokenu, nebudou muset provádět zbytečná volání při každém přístupu do API rozhraní. Toto jsem implementoval v autorizační mikroslužbě, která je popsaná v kapitole 4.3

3.3 Slovník obecných pojmů

3.3.1 Uživatelé

- **Student** – student, který si zapsal a studuje předmět BI-DBS.
- **Učitel** – vyučující, který přednáší, cvičí nebo zkouší předmět BI-DBS. Dle toho, co dělá, se pak definuje podrole: „Přednášející“, „Zkoušející“, „Cvičící“.
- **Garant** – člověk, který zodpovídá za celkovou výuku předmětu BI-DBS. Jeho zodpovědností je konfigurace veškerého nastavení předmětu. Také je zároveň učitelem.

[10]FIT Auth server – autorizační server (Zuul OAAS), který se zabývá vydáním tokenů dle OAuth 2.0 protokolu a umožňuje přístup do fakultních systémů (např. KOSapi, VVVSapi, Usermap API, …). Pro použití je potřeba se přihlásit do AppManager a vytvořit vlastní projekt. [58]

3.3.2 Základní nastavení systému

- **Import semestru** - získání všech informací o aktuálním semestru ze systému KOS. Kopírují se informace o všech paralelkách, zkouškách, uživatelích, rolích v aktuálním semestru.
- **Paralelka** – reprezentuje paralelku v předmětu BI-DBS, kterou vede jeden z vyučujících. Může to být přednáška, proseminář, cvičení.
- **Uživatel** – reprezentuje informaci o konkrétním uživateli a jeho rolích, které jsou popsané výše v sekci 3.3.1.
- **Kurz** – reprezentuje předmět BI-DBS, datum začátku semestru a jeho ukončení. Kurz se vypisuje v systému KOS.
- **Zkouška** – reprezentuje zkoušku z předmětu BI-DBS, která byla vypsána v systému KOS.
- **Mimořádná akce** – reprezentuje nějaký vypsaný termín pro test v semestru, odevzdání semestrálních práci či konzulace, kde je nutné přihlášení uživatele v systému KOS.
- **Místnost** – konkrétní existující místnost v prostorách ČVUT. Je exportovaná ze systému KOS.
- **Testový uživatel** – každý vyučující má vlastního testového studenta v každé paralelce, již učí. Je to profil neexistujícího studenta, který vyučující používá pro ukázku práce se systémem.

3.3.3 Databázová přípojení

- **Přípojení** – databázové přípojení, které slouží pro zobrazení, vyhodnocení, porovnání, transformace, validace sql/ra požadavků. Obsahuje veškeré informace potřebné pro spojení s příslušnou databází. Umožní se připojit k různým databázovým strojům.
- **Přístupy k přípojení** – povolení použít konkrétní přípojení konkrétním uživatelem. Uživatel může nastavit viditelnost přípojení dle paralelky, role, uživatele, kurzu, zkoušky či testu.
- **Výchozí přípojení** – přípojení, které se bude využívat při tvorbě semestrální práce.
- **Konceptuální model** – model reality, který je tvořen studentem pro jeho databázi v rámci semestrální práce.
- **Relační model** – model databázových tabulek s vazbami, atributy a cizími klíči.
- **Databázové schéma** – struktura databázových tabulek s atributy.
- **Data modeller** – nástroj umožnující vytvářet konceptuální modely.
- **Transformace modeller** – nástroj umožnující vytvářet konceptuální modely, které se automaticky transformují do SQL skriptu umožňující vytvořit odpovídající databázi.
- **SQL dotaz** – požadavek na výběr nebo vložení dat z databází napsaný v jazyce SQL.
- **RA dotaz** – požadavek na výběr dat z databází napsaný v relační algebře.
- **Transformace** – přeložení konceptuálního schématu do relačního.
- **Přeložení RA požadavku** – transformace dotazu v relační algebře na odpovídající dotaz v jazyce SQL.
- **Redaktor dotazů** – textový redaktor sql/ra požadavků umožňující provádět sql/ra požadavky na zvolené databázi a zobrazovat výsledky.

3.3.4 Testy

- **Konfigurace testů** – počty bodů, které může student získat během testu, zkoušky či ústní zkoušky. Nastavuje garant.

- **Zadání k testu** – obecný popis problému, který by měl student řešit. Například: popis databáze, databázový model, schéma apod. Ke každému zadání se vytváří několik otázek, které musí zachovávat pořadí. Zadání, které nemá otázky, je nevalidní a nelze ho použít v testu. Zadání může obsahovat: text, obrázek, model, transformaci, normalizaci nebo databázové přípojení, viz sekce 3.3.3.
- **Otázka v testu** – textový popis toho, co by měl student udělat v rámci konkrétního zadání. Otázka má nastavený maximální počet získaných bodů za správnou odpověď a učitele, který ji bude hodnotit po odevzdání. Otázka má právě jedno zadání a také musí obsahovat informaci o druhu odpovědi. Otázka může mít nastavené štítky, které ji zatřídí do některé z kategorií.
- **Historizace otázky** – zachování validní historie testů, pokud je některá z otázek změněna. Všechny testy by měly správně odkazovat na starou verzi otázky.
- **Referenční odpověď na otázku** – správná odpověď, kterou vytvoří učitel. Je to vzorová odpověď, dle které probíhá automatické hodnocení. Máme tyto druhy odpovědí:
  - textová odpověď,
  - výběr jedné správné možnosti,
  - výběr více správných možností,
  - sql,
  - ra,
  - konceptuální model,
  - transformace,
  - normalizace.
- **Studentova odpověď na otázku** – odpověď studenta na konkrétní otázku v konkrétním testu. Po ohodnocení se stane vzorovou pro tvorbu doporučení. Toto doporučení bude nabízet vyučujícím v průběhu hodnocení konkrétní počet bodů. Uchovává správnost odpovědí v procentech, tedy podíl získaných bodů do maximálního počtu bodů.
- **Varianta odpovědí na otázku** – jedna z referenčních odpovědí, která může být správná, nebo ne. Je určena pro typ odpovědí: „Výběr jedné správné možnosti“, „Výběr více správných možností“.
- **Štítek** – rozšiřuje informaci o objektu, do kterého je přiřazen. Každý štítek patří právě do jedné skupiny štítků.
- **Druh nebo skupina štítků** – shromáždí štítky konkrétního druhu. Každá skupina má jednu z úrovní viditelnosti: „Soukromý“ nebo „Veřejný“. Soukromou skupinu může vidět jen její autor. Veřejnou vidí všichni učitelé. Také skupina definuje způsob použití štítků ve skupině (disjoint mode), tedy kolik štítků ze skupiny lze přiřadit k jedinému objektu. Dva mody: „Pouze jeden štítek ze skupiny“, „Libovolné množství štítků ze skupiny“.

  Příklad:
  - Skupina(libovolné): otázka vhodná pro test.
  - Štítky: „Demo test“, „Zkouška“, „Test v semestru“, „Kvíz“.
  - Způsob použití: libovolné množství štítků ze skupiny.
  - Skupina(unikátní): složitost otázky.
  - Štítky: „Lehká“, „Průměrná“, „Těžká“, „Zabiják“.
  - Způsob použití: pouze jeden štítek ze skupiny.
- **Testová šablona** – shromažďuje veškeré informace potřebné pro vytvoření varianty testu. Obsahuje informace o délce testu, celkovém počtu bodů za test, počtu variant v testu, přiřazené otázky k testu, přiřazené automatické otázky apod. Také může mít nastavené štítky, které ji začlení do nějaké kategorie.

- **Automatická otázka** – představuje některou posloupnost štítků, dle kterých se v době generování testu vybere otázka.
- **Termín testu** – termín testu, který seskupí varianty testu. V rámci jednoho termínu může být více variant testu. Může obsahovat reference na termíny zkoušek a testů ze školního systému KOS, pokud je to zkouška nebo test v semestru. Termín představuje celkové okno pro test. Pokud je naplánována zkouška ve čtvrtek mezi 12:00 a 14:00, studenti mohou začít pracovat na svém testu (test studenta) jen během této doby. Ve 14:00 končí test pro všechny účastníky. Termín spustí vyučující, až poté může student spustit vlastní test.
- **Varianta testu** – instance šablony testu, obsahuje veškeré informace o testu, které jsou potřeba. Je tvořena ze šablony testu. Tuto variantu nelze po vytvoření měnit. Vůbec není závislá na šabloně, ze které byla vytvořena. Je použitelná pouze pro termín testu, ve kterém byla vytvořená. Může být dvojího druhu:
  - Statická – otázky pro test jsou zvolené striktně, ve výsledku má každý student stejný test.
  - Automatická (dynamická) – otázky pro test se zvolí na základě nastavených štítků. Každý student může mít různý test.
- **Test studenta** – test, který patří konkrétnímu studentovi, má vlastní dobu zpracování, ale nemůže trvat déle, než trvá termín. Můžeme mít několik druhů testu: „Test v semestru“, „Demo test“, „Zkouška“, „Kvíz“. Probíhá v konkrétní místnosti, pokud je to zkouška a nebo test v semestru. Tento test spouští student.
- **Automatické hodnocení testu** – hodnocení testu po odevzdání studentem pomocí automatu. Pokud otázka nebyla ohodnocena automatem, hodnotí ji vyučující pomocí manuálního hodnocení.
- **Manuální hodnocení otázek** – anonymní hodnocení otázek vyučujícím. Vyučující hodnotí mnoho stejných otázek z více různých studentských testů, a proto neví, čí test hodnotí.
- **Odevzdaný test** – studentův test, který byl vyplněn a odevzdán k hodnocení.
- **Ohodnocený test** – studentův test, který byl ohodnocen automatickým či manuálním hodnocením.

3.3.5 Notifikace

- **Notifikace po odevzdání testu** – vyučujícímu je oznámeno, že někdo ze studentů odevzdal test.
- **Notifikace po ohodnocení testu** – studentovi je oznámeno, že byl test ohodnocen.
- **Notifikace po odevzdání semestrální práce** – vyučujícímu je oznámeno, že konkrétní student odevzdal svoji semestrální práci.
- **Notifikace po ohodnocení semestrální práce** – studentovi je oznámeno, že vyučující ohodnotil jeho semestrální práci.
- **Notifikace při psaní zkoušky** – studentovi je oznámeno, že bude psát zkoušku.

3.3.6 Semestrální práce

.

- **Semestrální práce** – studentova práce, kterou student vyvíjí v průběhu celého semestru a po odevzdání všech její částí dostává zápočet. Semestrální práce je rozdělená na jednotlivé iterace. Současně se semestrální práce skládá s následujících částí:
  - název a popis,

  - konceptuální schéma,
  - diskuze smyček v rámci konceptuálního schématu,
  - relační schéma,
  - create skript – sql skript, který je určen pro vytvoření nové databáze a může být proveden s použitím výchozího přípojení uživatele,
  - insert skript – sql skript, který je určen pro vytvoření nových záznamů v databázi a může být proveden s použitím výchozího přípojení uživatele,
  - dotazy nad zvolenou databází – sql/ra/textové dotazy nad zvolenou databází, které splňují konkrétní kategorie,
  - závěr.

- **Iterace v semestrální práci** – podčást semestrální práce, která má stanovené konkrétní požadavky, jež student musí splnit. Pokud tyto požadavky nejsou splněny, student nemůže dostat zápočet. Iterace může být závislá na předchozích iteracích. Každá iterace se validuje a student ji odevzdává nezávisle na ostatních. Odevzdání představuje kopii práce studenta v okamžik odevzdání.
- **Požadavek v iteraci** – každá iterace má seznam požadavků, které musí student splnit. Je tady definováno, jaké části semestrální práce patří konkrétní iteraci. Obsahem požadavku jsou předdefinovaná omezení na obsah semestrální práce. Příklady takových požadavků:
  - „Alespoň 3 přirozené dotazy“.
  - „Počet typů entit v konceptuálním schematu >= 7“.
  - „Pokryté kategorie C, D“.
- **Odevzdání semestrální práce** – kopie stavu semestrální práce studenta po ověření splnění požadavků, které jsou kladeny na iteraci, a poslaná ke kontrole učitelem.
- **Ohodnocená semestrální práce** – konkrétní odevzdání semestrální práce, které bylo ohodnoceno učitelem.
- **Odmítnutá semestrální práce** – konkrétní odevzdání semestrální práce, které bylo odmítnuto učitelem.
- **Znovu odevzdaná semestrální práce** – odevzdání semestrální práce, které bylo vráceno učitelem, opraveno a znovu odevzdáno studentem nebo odevzdáno studentem podruhé.
- **Historie odevzdání** – seznam prací odevzdaných studentem.

3.4 Slovník procesů

V předchozí podkapitole jsem definoval veškeré pojmy, které aktuálně používáme v doméně. Tyto informace nejsou ovšem dostačující pro pochopení domény. Proto potřebujeme také definice veškerých procesů, které se v rámci domény odehrávají. Tyto definice a seskupení procesů do konkrétních skupin dovolí novým účastníkům projektu mnohem lépe pochopit systém již od začátku. Abychom byli také schopni správně navrhnout rozdělení monolitické aplikace do mikroslužeb, potřebujeme správně definovat procesy, které se odehrávají v doméně. Z pohledu mikroslužeb neexistuje centrální logika aplikace, ale jen jednotlivé procesy. [59] V této podkapitole zkusím zadefinovat procesy odehrávající se během výuky předmětu BI-DBS. Tyto procesy budou rozdělené do sekcí dle zaměření.

3.4.1 Konfigurace předmětů

- Zobrazení dostupných předmětů pro import ze systému KOS.
- Import semestru ze systému KOS.

- Import termínů testů nebo zkoušek ze systému KOS.
- Nastavení požadavků klasifikace.
- Kopírování konfigurace předmětů z jiného semestru.
- Vyhledávání a zobrazení kurzů, paralelek, uživatelů, zkoušek, místností importovaných ze systému KOS.

3.4.2 Konfigurace semestrální práce

- Vytvoření a zobrazení požadavků pro semestrální práci.
- Vytvoření a zobrazení iterací semestrální práce.
- Přiřazení konkrétních požadavků na semestrální práci jednotlivým iteracím.
- Nastavení podmínek klasifikace pro jednotlivé iterace.
- Přehled a nastavení termínů pro odevzdání jednotlivých iterací semestrální práce.
- Nastavení a import termínů odevzdání semestrálních prací ze systému KOS.
- Zobrazení všech požadavků pro semestrální práci.

3.4.3 Databázová přípojení

- Správa databázových přípojení (zobrazení, vytvoření, úprava, odstranění).
- Zobrazení databázového schématu.
- Zobrazení struktury databázových tabulek.
- Sdílení vlastního databázového připojení mezi jiné uživatele, paralelky, kurzy, role, zkoušky.
- Transformace ra dotazů do sql.
- Vyhodnocení sql a ra dotazů.
- Testování shodností několika sql dotazů.

3.4.4 Autorizace uživatelů

- Vytvoření autorizačního tokenu.
- Obnovení autorizačního tokenu.
- Smazání autorizačního tokenu.
- Změna identity uživatele.
- Navrácení původní identity uživatele.

3.4.5 Globální štítky

- Správa a zobrazení štítků.
- Správa a zobrazení skupin štítků.
- Nastavení viditelnosti štítků: globální nebo lokální.
- Nastavení kategorie, pro kterou mohou být tyto štítky použity: testová šablona, otázky, SQL dotazy v semestrálních pracích apod.

3.4.6 Notifikace

- Nastavení notifikací pro konkrétního uživatele.
- Zobrazení všech notifikací.
- Oznámení vyučujícímu, že někdo ze studentů odevzdal test.
- Oznámení studentovi, že test byl ohodnocen vyučujícím.
- Oznámení vyučujícímu, když konkrétní student odevzdá svoji semestrální práci.
- Oznámení studentovi, když vyučující ohodnotí jeho semestrální práci.
- Oznámení studentovi, když byl přiřazen ke zkoušce.

3.4.7 Tvorba zadání testu

- Zobrazení a správa zadání dle konkrétních typů (podzadání).
- Zobrazení a správa otázek ke konkrétnímu zadání.
- Nastavení štítků pro konkrétní otázku.
- Automatický výběr otázek na základě nastavených štítků.
- Ověření použitelnosti otázek pro test. Například pokud pro otázku chybí odpověď, tak není použitelná pro test.

3.4.8 Testové šablony

- Zobrazení a správa testových šablon.
- Nastavení štítků pro konkrétní testovou šablonu.
- Nastavení otázek pro konkrétní testovou šablonu.
- Nastavení automatických otázek pro konkrétní testovou šablonu.
- Duplikace testové šablony.

3.4.9 Testy

- Zobrazení a správa testových termínů.
- Zobrazení a správa variant testu.
- Přidávání variant testu do termínu.
- Správa místností pro test.
- Přidávání studentů k variantě testu.
- Přidávání paralelek k variantě testu.
- Přidávání času k termínu a nebo ke konkrétnímu studentskému testu.
- Zahájení termínu: spuštění všech variant testu.
- Zahájení studentského testu studentem.
- Ukončení termínu: ukončení všech variant testu.
- Zobrazení studentských testů.
- Ukončení studentských testů.

3.4.10 Hodnocení testu

- Zobrazení a správa referenčních odpovědí dle typu otázky.
- Nastavení referenčních odpovědí.
- Zobrazení a správa studentových odpovědí na konkrétní test.
- Zobrazení studentových odpovědí, které se využívají pro hodnocení dalších odpovědí.
- Sběr IP logů studentů, kteří zpracovávají test a přidávají nové odpovědi.
- Přidávání poznámek k odpovědím: komentáře učitele, studenta a nebo automatu.
- Automatické asynchronní hodnocení odpovědí studentů.
- Manuální hodnocení odpovědí studentů, které umožní hodnotit následující neohodnocené odpovědi. Například vyučující ohodnotí odpověď, všechny stejné neohodnocené odpovědi jsou ohodnoceny stejným počtem bodů.

3.4.11 Tvorba semestrální práce

- Tvorba názvu a popisu semestrální práce.
- Tvorba konceptuálního schématu v semestrální práci.
- Ověření správnosti konceptuálního schématu.
- Volba databázového přípojení pro semestrální práci.
- Tvorba popisu smyček konceptuálního schématu.
- Tvorba relačního schématu semestrální práce.
- Vytvoření create skriptu pro semestrální práci.
- Ověření správnosti create skriptu.
- Provedení create skriptu nad zvoleným databázovým přípojením.
- Vytvoření insert skriptu pro semestrální práci.
- Ověření správnosti insert skriptu.
- Provedení insert skriptu nad zvoleným databázovým přípojením.
- Přidávání a správa dotazů nad zvoleným databázovým přípojením.
- Provedení dotazu s použitím zvoleného databázového přípojení.
- Transformace dotazu v relační algebře do jazyka SQL nad zvoleným databázovým přípojením.
- Nastavení kategorie dotazu v jazyku SQL.
- Tvorba závěru.
- Přidávání zdrojů.
- Přehled semestrálních prací.

3.4.12 Odevzdání a hodnocení semestrální práce

- Validace splnění požadavků pro semestrální práci.
- Validace správnosti vytvořených skriptů.
- Zobrazení a vytvoření nového odevzdání. (Vytvoření kopie aktuálního stavu semestrální práce).
- Odevzdání semestrální práce studentem.
- Hodnocení semestrální práce vyučujícím.
- Přidávání bonusových bodů za semestrální práce vyučujícím.

- Odmítnutí semestrální práce vyučujícím.
- Opětovné odevzdání semestrální práce studentem.
- Přidávání komentářů k jednotlivým částem semestrální práce.
- Zobrazení rozdílů mezi jednotlivými odevzdáními studenta.
- Přehled odevzdání studentů.

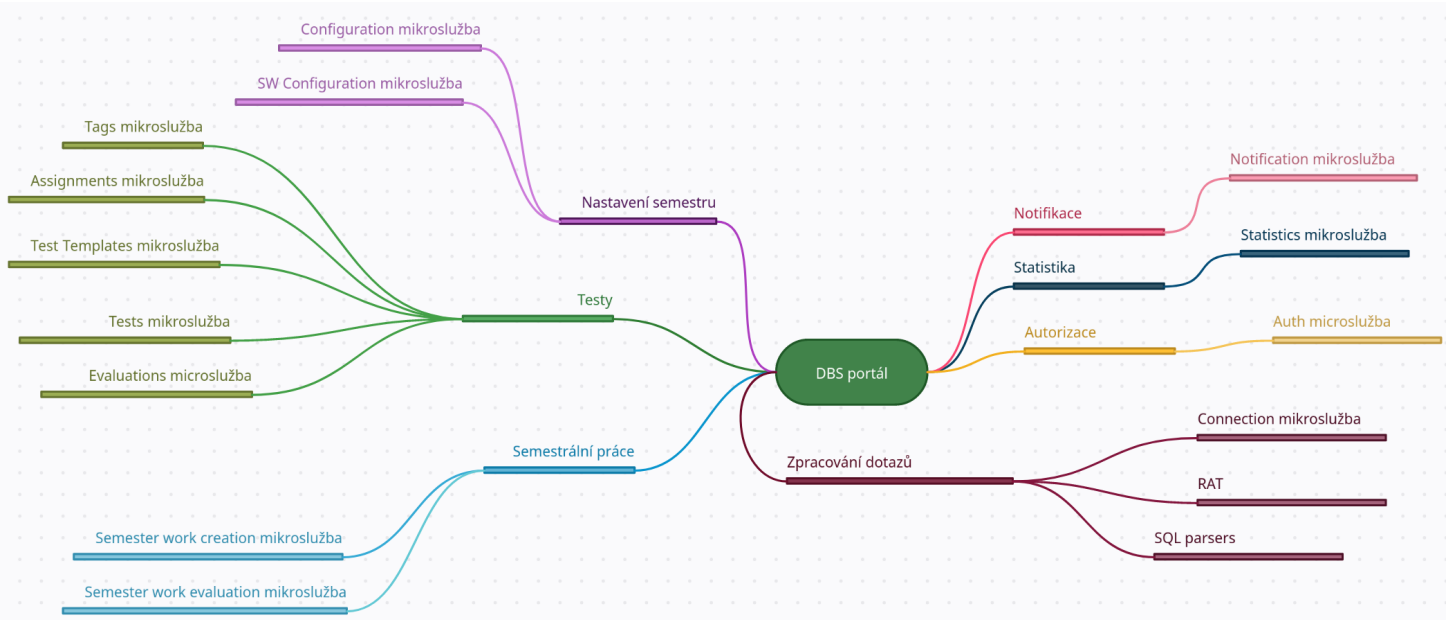
3.4.13 Export hodnocení do systémů KOS a FIT Klasifikace

- Vyhledávání všech hodnocení studenta ve všech částech DBS portálu a export celkového hodnocení do systému KOS a FIT Klasifikace.
- Přehled celkového hodnocení studentů.

3.4.14 Vyhodnocení statistik

- Sbíraní statistik ze všech částí systému.
- Přehled úspěšnosti studentů, vyučujících.
- Přehled úspěšnosti zadání a otázek v testech.

3.5 Rozdělení do mikroslužeb

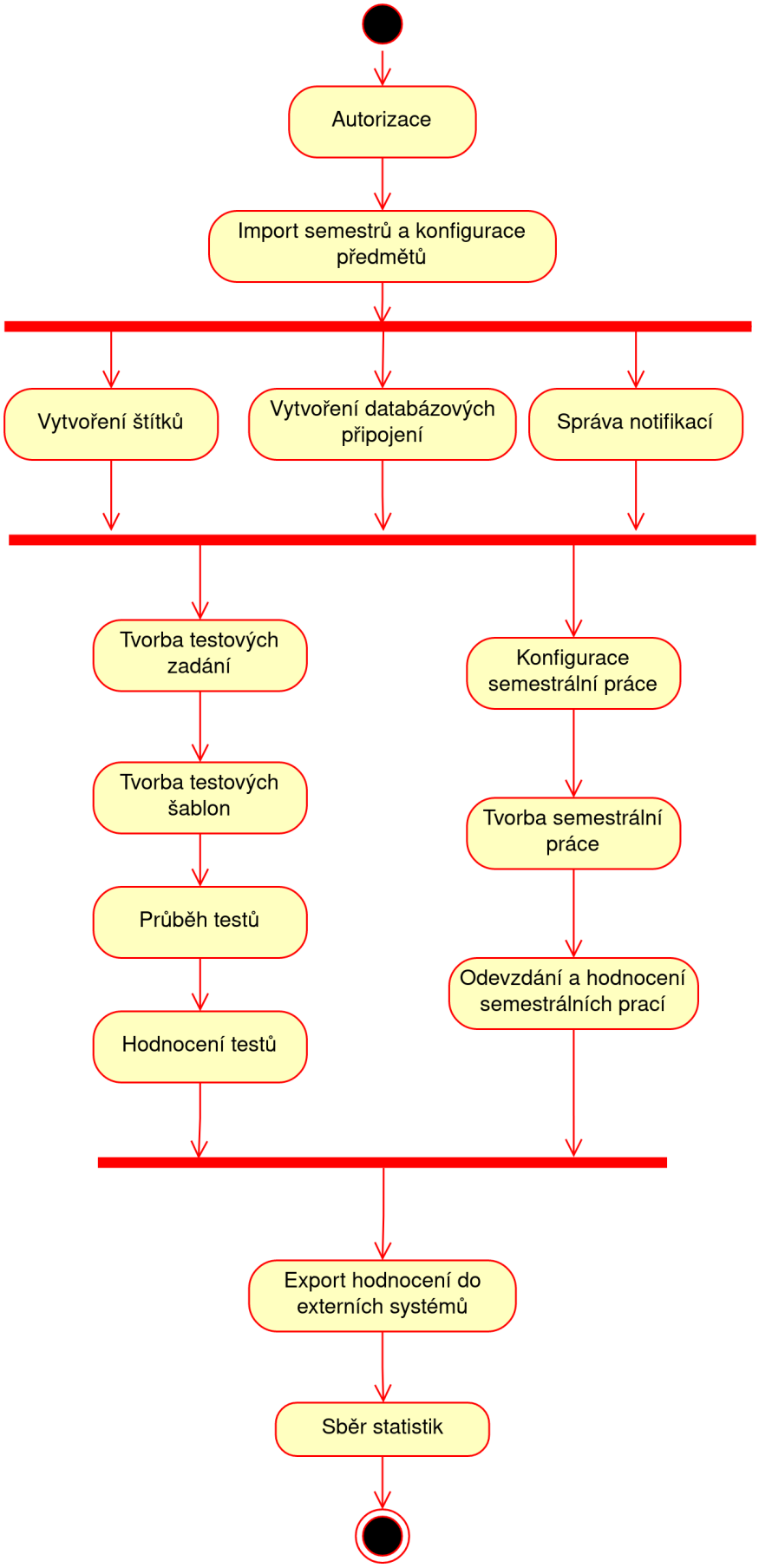


Obrázek 3.9 Road mapa rozdělení DBS portálu na jednotlivé mikroslužby

V předchozí podkapitole jsem zadefinoval a rozdělil související procesy v systému do skupin. Na základě toho mohu definovat několik hlavních procesů, které probíhají postupně a jsou závislé na předchozích. Tyto procesy jsou: „Autentizace a autorizace uživatelů“, „Konfigurace předmětu a import studentů, kurzů, paralelek“, „Správa databázových přípojení“, „Správa štítků“, „Správa notifikací“, „Správa testových zadání a otázek“, „Správa testových šablon“, „Správa testů a jejich průběh“, „Vyhodnocení testů“, „Nastavení semestrálních prací“, „Vytvoření semestrálních prací“,

„Hodnocení semestrálních prací“, „Export hodnocení do externích systémů“ a „Správa statistik“. Všechny procesy jsou znázorněné pomocí diagramu na obrázku 3.10. Tento diagram neodpovídá plně specifikaci diagramu aktivit, ale jen znázorňuje, jak uživatel postupně provádí každý z procesů, který je závislý na předchozích. Každý z těchto procesů definuje nezávislou doménu, která umožňuje postupný tok dat z jedné domény do jiné. Tedy předchozí proces končí začátkem následujícího, což tvoří tu nezávislost domén, kterou potřebuji. Na základě těchto domén vytvořím samostatné celky systému – mikroslužby. Tady použiji návrhový vzor „Dekompozice dle poddomény“, který představuje jeden ze způsobů návrhu mikroslužeb popsaný v podkapitole 3.2. Pro znázornění rozdělení DBS portálu do mikroslužeb jsem vytvořil road mapu, kterou můžete vidět na obrázku 3.9. Náš systém se bude skládat z následujících mikroslužeb:

- „Auth microservice“ – doména přihlášení uživatele do systému pomocí přihlašovacího tokenu. Tato doména představuje procesy spojené s autorizací uživatele pomocí přístupového tokenu. Zodpovídá za přihlášení uživatele do systému a další funkce, jako je například obnovení přihlašovacího tokenu, změna identity uživatele apod., viz sekce 3.4.4. Popis implementace této mikroslužby lze nalézt v podkapitole 4.3.
- „Configurations microservice“ – doména konfigurace celkového semestru. Je velmi důležitá, protože obsahuje nejdůležitější informace v systému, tj. informace o uživatelích, kurzech, paralelkách, zkouškách apod. Je nezbytná pro fungování systému jako celku. Zodpovídá za nastavení semestru, za import semestru ze systému KOS a další funkce, které jsou popsané v předchozí kapitole v sekci 3.4.1.
- „Connections microservice“ – doména databází a práce s nimi. Tato mikroslužba zodpovídá za správu databázových přípojení uživatelů, provádění databázových dotazů, jejich porovnání a běžnou práci s databází. Všechny funkce jsou popsané v sekci 3.4.3.
- „Tags microservice“ – postavená nad doménou štítků. V rámci této domény se zabýváme jen správou štítků. Podrobný popis procesů, které tady probíhají, lze nalézt v sekci 3.4.5 a 3.3.4.
- „Notifications microservice“ – odpovídá doméně notifikací uživatelů. Tato mikroslužba je zodpovědná za zobrazení, nastavení a vytvoření notifikací uživatelů. Ostatní mikroslužby ji mohou použít pro oznámení uživatele o nějaké události, viz 3.4.6.
- „Assignment creation microservice“ – doména tvorby zadání pro testy a zkoušky z předmětu BI-DBS. Mikroslužba zodpovídá za vytvoření a validaci zadání a otázek pro testy. Výsledkem tvorby je validní zadání testu, které lze používat v různých testech (může být použito ve více testových šablonách). Více se o procesu tvorby zadání pro test můžete dozvědět v podkapitolách 3.4.7 a 3.3.4.
- „Test Templates microservice“ – odpovídá doméně tvorby šablony pro test. Tento proces začíná tvorbou šablony z existujících zadání a štítků a výsledkem je validní testová šablona, ze které lze generovat libovolný typ testu. Samotný vytvořený test je nezávislý na šabloně a není uložen do této mikroslužby. Podrobně lze o těchto procesech přečíst v podkapitolách 3.4.8 a 3.3.4.
- „Tests microservice“ – postavená nad doménou průběhu testu. Zodpovídá za veškeré procesy průběhu testu a jeho vytvoření z testové šablony, přiřazení uživatelů k testu, spuštění testu, ukončení testu atd. Začíná se vytvořením testu z testové šablony a končí ukončením studentského testu. Dále proces přechází do „Test Evaluation microservice“. Více o průběhu testu lze nalézt v předchozích podkapitolách 3.4.9 a 3.3.4.
- „Test Evaluation microservice“ – zodpovídá za doménu hodnocení testu. Po odevzdání testu studentem začíná jeho hodnocení. Každá otázka je hodnocena dle konkrétního typu. Proces končí vyhodnocením testu, viz podkapitoly 3.4.10 a 3.3.4.
- „SW Configurations microservice“ – doména konfigurací semestrální práce. Zodpovídá za nastavení požadavků v iteraci, vytvoření iterace, kopírování konfigurace z předchozích



Obrázek 3.10 Diagram znázorňující tok procesů v systému

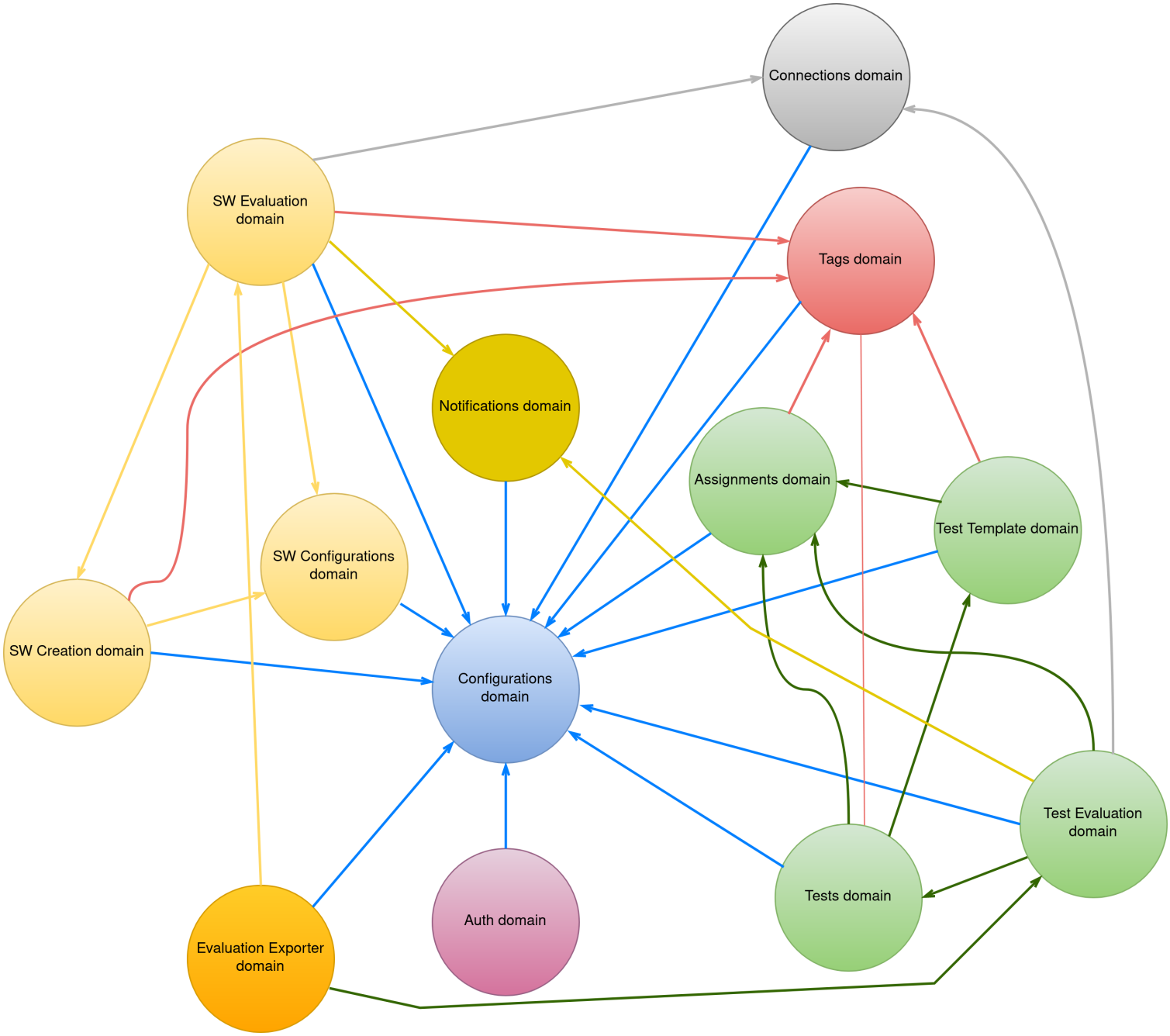
semestrů, nastavení termínů odevzdání apod. Podrobný popis procesů lze nalézt v sekci 3.4.2.

- „SW Creation microservice“ – doména tvorby semestrální práce. Proces začne po nastavení všech požadavků na semestrální práci. V rámci této domény uživatel tvoří svou vlastní semestrální práci, kterou ve výsledku odevzdává k hodnocení. Každé odevzdání představuje kopii aktuální semestrální práce studenta, kterou už zpracovává jiná mikroslužba: „SW Evaluation microservice“. Podrobně o tvorbě semestrálních prací viz podkapitoly 3.4.11 a 3.3.6.
- „SW Evaluation microservice“ – představuje doménu hodnocení semestrálních prací. Proces začíná správou odevzdání práce studenta a končí ohodnocením práce studenta vyučujícím. Podrobný popis hodnocení semestrálních prací lze nalézt v podkapitolách 3.4.12 a 3.3.6.
- „Evaluation exporter microservice“ – doména sbírání hodnocení studentů. Mikroslužba se zabývá sbíráním a uložením hodnocení z různých částí systému. Proces začne, pokud vyučující ohodnotí některou ze semestrálních prací nebo některý ze studentských testů. Skončí uložením hodnocení a jeho exportem do externích systémů, viz podkapitola 3.4.13.
- „Statistics microservice“ – představuje doménu statistik. Proces začíná sběrem statistik o úspěšnosti od začátku semestru. Skončí vyhodnocením úspěšnosti na konci semestru. Podrobněji se lze dozvědět v podkapitole 3.4.14.

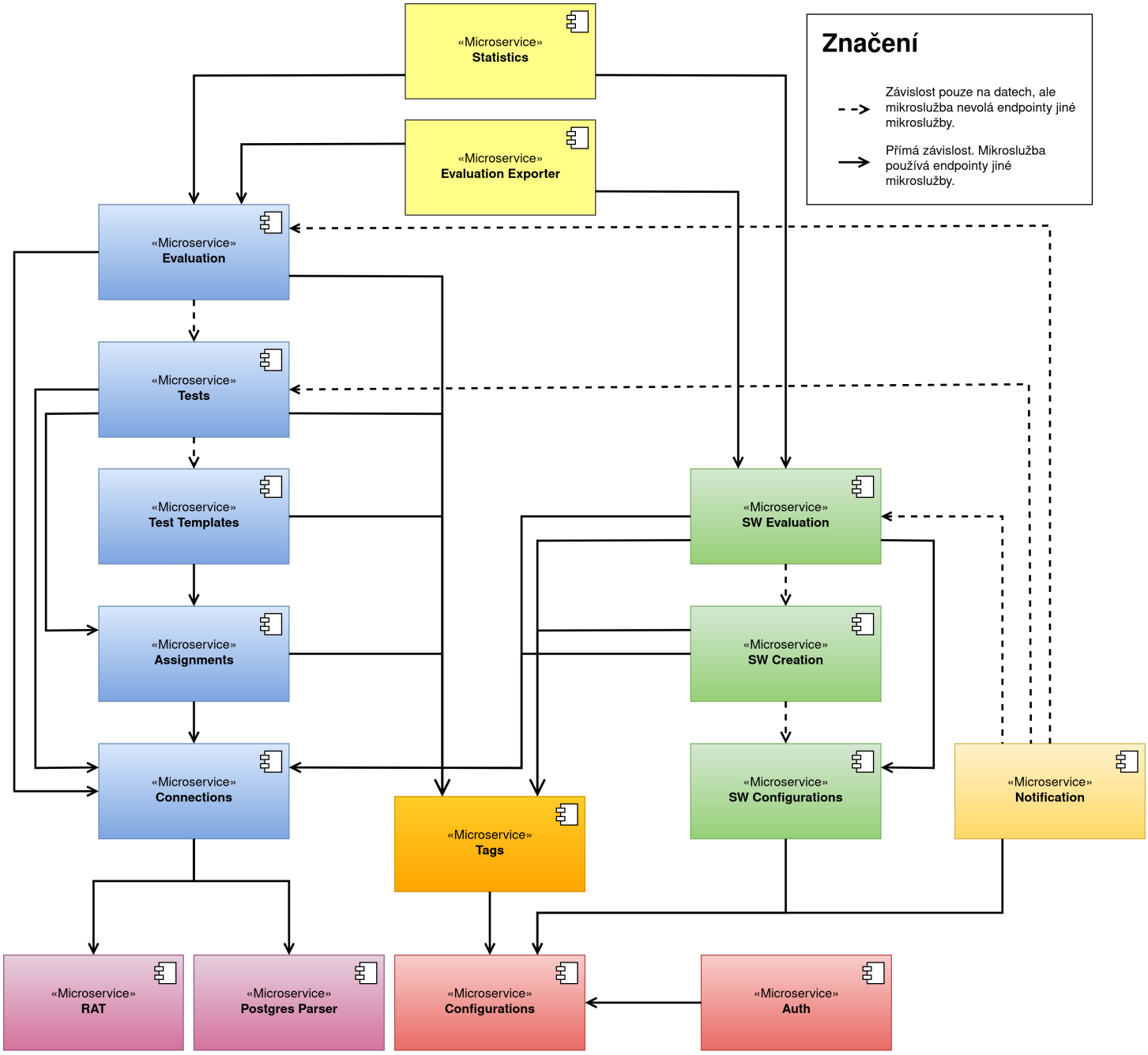
Výhodou tohoto rozdělení je to, že každá následující mikroslužba potřebuje pouze číst data z předchozí a nemusí je zapisovat. To zaručuje velkou nezávislost celku a jednodušší úpravu v budoucnu. Ovšem jedna z mikroslužeb, konkrétně „Configurations microservice“, představuje jádro systému a na ní je závislá většina jiných mikroslužeb. Proto bude její úprava vyžadovat větší pozornost. Za spojení všech těchto mikroslužeb do jednoho celku bude odpovídat integrační služba (frontend), která si bude držet potřebný mezistav během přechodu z jedné mikroslužby do druhé. Pro uložení dat používá každá mikroslužba vlastní databázi, což zajistí správnost dat, protože přístup k těmto datům zprostředkovává tato mikroslužba a nikdo jiný by je napřímo získat neměl, viz sekce 3.2.2.1. Kontextovou mapu celé domény DBS portálu můžeme pozorovat na obrázku 3.11. Na této mapě jsou dobře vidět datové závislosti mezi doménami. Tyto závislosti jsou jednosměrné a často jsou závislé na jiných mikroslužbách jen kvůli ověření správnosti objektů před uložením. Například většina mikroslužeb musí přímo volat mikroslužbu Configurations. Ale jiné mikroslužby, např. mikroslužby odpovídající za tvorbu a spuštění testu, potřebují jen validovat existenci objektů pomocí volání svých sousedů, ale celkově budou integrované pomocí integrační služby: API Gateway, viz sekce 3.2.3.1. Tyto závislosti mezi mikroslužbami lze podrobněji vidět na obrázku 3.12.

Tento návrh definuje pouze celkovou architekturu DBS portálu, specifikaci a omezení jednotlivých mikroslužeb, ale nic neříká o jejich vnitřním návrhu a implementaci. Podrobný technický popis architektury jsem uvedl v podkapitole 4.2. Celkově je to vymyšleno tak, aby si každý z SP týmů mohl zvolit nějakou mikroslužbu a začít na ní pracovat od začátku, tj. nejprve analyzovat problém, navrhnout řešení, implementovat a integrovat do celku a nakonec otestovat výstup. Tento návrh je vhodný pro nové SP týmy, protože úzce souvisí s tím, co se potřebují naučit v rámci softwarového projektu.

V rámci této diplomové práce jsem spolupracoval s několika SP týmy. Zabýval jsem se řízením vývoje, validací jejich výstupu a přidával drobné optimalizace či vylepšení v kódu. Za tuto dobu jsme navrhli a implementovali několik hlavních mikroslužeb: Configurations mikroslužbu, Connections mikroslužbu a SW Configurations mikroslužbu. Bylo vidět, že pro SP týmy to bylo zajímavější než práce na monolitním portálu. Velmi rychle se zorientovali v požadavcích a psali kvalitní kód. Myslím si, že velký vliv měla jednoduchost samotných systémů. Jsou malé a jednoduché, proto by se v nich každý mohl zorientovat během půl hodiny, na rozdíl od monolitní aplikace. Rámcově měl kód jedné mikroslužby kolem 7-9 tisíc řádků včetně všech konfiguracích souborů, což je 10 až 15 krát méně než u monolitního portálu.



Obrázek 3.11 Kontextová mapa jednotlivých poddomén



Configurations mikroslužba

Tato mikroslužba uchovává všechny informace o uživatelích, kurzech, paralelkách, proto všechny ostatní mikroslužby jsou na ní závislé. Pro přehlednost tyto vazby nezobrazuji.

Obrázek 3.12 Závislosti mezi mikroslužbami DBS portálu

Samostatně jsem navrhl a implementoval autorizační mikroslužbu, kterou popisuji v podkapitole 4.3. Autentizace a autorizace uživatelů jsou složitějšími funkcemi systému, které by měly dobře fungovat v rámci celkové architektury mikroslužeb. Navíc potřebuji zajistit rychlé ověření identity uživatelů, aby to nezpůsobovalo zbytečné zatížení systému. Proto jsem nemohl dovolit její implementaci zvlášť pro každou mikroslužbu, ale potřeboval jsem to vyřešit obecně. Nejen autorizace, ale i mnoho implementovaných funkcí jsou společné pro více mikroslužeb. Abych mohl odstranit tento nedostatek architektury mikroslužeb, rozhodl jsem se implementovat knihovnu, která by mohla shromáždit všechny společně využívané funkce a dovolila je znovu použít v každé mikroslužbě bez nutnosti duplicitní implementace. Tak vznikl návrh knihovny „DBS Utils bundle“, který popíšu v podkapitole 4.4.

3.6 Migrace dat

Navrhl jsem novou architekturu DBS portálu, která dělí systém na více malých nezávislých částí. Poslední problém, který jsem potřeboval vyřešit, byla migrace dat z monolitické aplikace do jednotlivých mikroslužeb. Data z DBS portálu je potřeba zachovat, protože tento systém umí automaticky hodnotit testy na základě předchozích vyhodnocených odpovědí. Proto potřebuji zachovat alespoň všechna zadaní, otázky a odpovědí. Každá mikroslužba má svoji vlastní databázi s úplně jinou vnitřní strukturou, a proto nemohu data z databáze původního DBS portálu tak jednoduše přesunout do nového. Musím nejdřív prozkoumat všechna data, zjistit, jaká data a kde je mám v obou systémech. Poté je nutné vytvořit datovou mapu. Ta mi řekne, který sloupeček v které tabulce se má uložit do které databáze, tabulky a sloupečku. Nakonec mohu pomocí speciálního nástroje vytáhnout data ze staré databáze, modifikovat je a uložit do nové. Tento proces se nazývá **Extrakt Transform Load (ETL)** a umožňuje vytáhnout potřebná data ze zdrojové databáze, transformovat do potřebné podoby a uložit je do cílového úložiště. Může se provádět jak pomocí specializovaných nástrojů, tak pomocí SQL funkcí a procedur konkrétního databázového stroje. Obecně se tato metoda používá pro sloučení dat z více zdrojů například pro vytvoření datového skladu. Ale v mém případě bude vhodná pro migraci dat z našeho monolitního DBS portálu do jednotlivých mikroslužeb. Implementace těchto procesů by měla podporovat vrácení změn a zachycení chyb, tj. musí správně pracovat s databázovými transakcemi a navíc zaručit idempotentnost: opakované spuštění vytvoří stejný výsledek. [60, 61] Jelikož je DBS portál obrovská aplikace s velkým množstvím databázových tabulek, navrhnu a vytvořím postup pro tuto migraci s nějakým jednoduchým příkladem. Celkovou práci dle tohoto postupu může dodělat SP tým v rámci předmětu „Softwarový projekt“.

3.6.1 Datová logická mapa

Datová logická mapa slouží pro dokumentaci a migraci dat mezi zdrojovým a cílovým systémem či úložištěm. Obsahuje informace o tabulkách, sloupečkách a jejich významu ve zdrojovém systému, které se mapují na tabulky a sloupečky v cílovém systému. To umožňuje mít podrobné informace o tom, odkud tato data pochází, a znát jejich sémantiku. To znamená, že vždy vím, který sloupeček v tabulce cílového systému odpovídá sloupečku v konkrétní tabulce zdrojového systému. To definuje, jakým způsobem bych měl zpracovat data ze zdrojového systému, aby je bylo možné uložit do cílového systému. Na základě této mapy může programátor provádět migraci bez nutnosti porozumění těmto datům. [61] Ukázku této datové mapy pro testové šablony můžete vidět na obrázku 3.13. Celkovou mapu lze nalézt v příloze na obrázku A.2.

Entita	Zdrojový systém				Cílový systém				Metadata	
	Databáze	Tabulka	Sloupec	Datový typ	Databáze	Tabulka	Sloupec	Datový typ	Byznys název	Byznys definice
TestTemplate	dbs_db	test_template	test_template_id	integer	test_templates_db	test_template	test_template_id	bigint	Identifikátor testové šablony	Identifikuje testovou šablonu v systému (Nemigrovat)
		test_template	name	varchar(50)	test_templates_db	test_template	name	varchar(50)	Název testové šablony	
		test_template	totalPoint	integer	test_templates_db	test_template	max_points	integer	Počet bodů	Maximální počet bodů, který student může získat z tohoto testu
		test_template	length	integer	test_templates_db	test_template	test_duration	integer	Trvalost studentového testu	Čas, který je odveden na vyplnění testu studentem
		test_template	total_length	integer	tests_db	test_term	term_duration	integer	Trvalost testového termínu	Čas, který je celkově odveden pro termín
		test_template	type	integer	tests_db	test_term	type	varchar(50)	Typ studentského testu	Např. zkouška, test v semestru, demo test
		test_template	user_id	integer	test_templates_db	test_template	creator_id	bigint	Autor šablony	
		test_template	active	boolean	test_templates_db	test_template	is_archivated	boolean	Archivovaná šablona	Testová šablona, kterou učitel archivoval. Nezobrazuje se v seznamu šablon.
		test_template	valid	boolean	test_templates_db	test_template	is_valid	boolean	Validní šablona	Testová šablona je validní, pokud všechny otázky jsou validní a součet bodů v otázkách je roven počtu bodů nastavenému v šabloně.
TestVariant		test_template	totalPoint	integer	tests_db	test_variant	max_points	integer	Počet bodů	Maximální počet bodů, který student může získat z tohoto testu
		test_template	total_length	integer	tests_db	test_variant	term_duration	integer	Trvalost testového termínu	Čas, který je celkově odveden pro termín
TestQuestion		test_question_to_test	test_question_id	integer	test_templates_db	static_question	question_id	bigint	Identifikátor testové otázky	Identifikuje testovou otázku v systému

Obrázek 3.13 Ukázka datové mapy pro testové šablony

3.6.2 Postup pro migraci dat

1. Vytvořit datovou logickou mapu pro každou tabulku zdrojového systému.
2. Pro každou sadu dat vytvořit ETL proces. Tento proces může být vytvořen pomocí Python skriptu, SQL skriptu nebo specializovaného nástroje, např. Pentaho.
3. Exportovat data z produkční databáze monolitního DBS portálu.
4. Vytáhnout, transformovat a uložit potřebná data do cílové databáze pomocí definovaného ETL procesu.
5. Orchestrace jednotlivých procesů. Typicky jsou tyto procesy mezi sebou propojené a závislé, proto je nutné zaručit správné pořadí provádění.

Kapitola 4
Implementace

4.1 Zavedené konvence

V každém projektu, na kterém pracuje velké množství lidí, musíme definovat nějaká pravidla, která by měl každý dodržovat. V opačném případě bude každý nový člověk na projektu produkovat jiný výstup, což negativně ovlivní základní styl, ve kterém se projekt dělal. Pro každého dalšího vývojáře, který se přípojí do vývoje, to bude matoucí a on nebude vědět, kterého stylu se musí držet. Ve výsledku to způsobuje v projektu zmatek, protože se najednou používají různé styly návrhu, různé způsoby pojmenování apod. To snižuje přehlednost kódu, což svým způsobem snižuje rychlost vývoje. Abych mohl tento problém odstranit, celkový kód vypadal vždy stejně a aby vývojáři pokaždé používali stejné způsoby návrhu, zavedu při návrhu a psaní kódu několik konvencí a doporučení.

4.1.1 GIT jmenná konvence

Git je verzovací nástroj s otevřeným zdrojovým kódem, který je používán ve většině softwarových projektů pro verzování zdrojových kódů. V našem projektu ho také použijeme a jako hlavní úložiště zdrojových kódů projektu budeme používat školní GitLab systém. Aby studenti v SP týmu vždy dodržovali správné pojmenování komitů, větví apod., potřebujeme si definovat pravidla. Tato pravidla by pak studenti měli striktně dodržovat. V rámci projektu jsem definoval následující pravidla:

- „Branches“ – pro každý úkol v systému Redmine by si měl student definovat novou větev. Větev by se měla jmenovat: <číslo vašeho úkolu v systému Redmine>_<název mikroslužby>_<váš název větví>. Pokud je úkol velký, měla by se vytvořit pro něj hlavní větev. Do ní by se měl sjednotit výstup ze všech podúkolů a následně se vytvořit „Merge request“ do hlavní „master“ větve projektu. Je vhodné vytvářet „Merge request“ i v rámci sloučení výstupu z podúkolů. Název větve by měl vysvětlovat to, co se v rámci úkolu udělalo. Příklad: 1455_connections_adding_load_endpoint.

- „Commits“ – název komitu by měl definovat to, co se v rámci něj udělalo. Je vhodné vytvářet menší komity, které se dají jednoduše zrušit, pokud bude potřeba. Název by měl mít následující podobu: [#<číslo vašeho úkolu v systému Redmine>][název služby] <váš název>. Pokud by se v rámci komitu přidaly nějaké složité konstrukce nebo by byl tento komit velký, bylo by nutné přidat i popis komitu. Příklad: [#1445][connections] Adding connection builder for creating connections in dbs portal.

- „Merge requests“ doporučuji používat pro sjednocení každé vaší větve. To umožňuje vidět veškeré provedené změny a jednoduše dělat code review. Navíc umožňuje přidávat komentáře k jednotlivým řádkům v kódu, což je vhodné pro code review.

4.1.2 Code Style

Aby se zdrojový kód držel jednoho definovaného standardu, rozhodl jsem se použít nástroje pro kontrolu stylu kódu a statickou analýzu kódu[1]. To dovolí odhalit velké množství chyb a zaručí použití jednoho způsobu psaní kódu v celém projektu.

Pro kontrolu stylu kódu jsem použil nástroj „PHP CodeSniffer“ spolu se „Slevomat Coding Standard“, které dovolují definovat jediný standard pro náš projekt. Tato pravidla jsem definoval v souboru ruleset.xml, který se nachází v kořenu projektu každé mikroslužby a dovoluje validovat a opravovat zdrojové kódy tak, aby odpovídaly definovanému standardu. Ukázka tohoto souboru je v příloze na výpisu kódu B.3. Monolitní DBS portál také používá stejný nástroj, ale v novém projektu používám novou verzi jazyka PHP, proto jsem nemohl využít existující nastavení. Bylo potřeba předefinovat pravidla tak, aby odpovídala novým konstrukcím jazyka PHP a nezpůsobovala problémy. Více o této knihovně viz podkapitola 4.6.3.

Pro statickou analýzu PHP kódu jsem použil analytický nástroj „PHPStan“. Tento nástroj umožní analyzovat kód a vždy upozorní na nějaké nebezpečné konstrukce, chybějící typování a další obdobné problémy v kódu. Více se lze o tomto nástroji dozvědět v podkapitole 4.6.4.

SP tým vytvořil v rámci své práce automatizované testování s použitím mnou definovaných pravidel. Po každém odevzdání zdrojových kódů do systému Gitlab se spustí testování, a pokud testy naleznou nějaké chyby, bude vývojář upozorněn. Jde o vhodné řešení, protože nedovolí odevzdat kód, který není v souladu s definovanými pravidly.

4.1.3 Databázová jmenná konvence

Pro pojmenovávaní databázových objektů potřebujeme také zavést pravidla, aby bylo možné dodržet jeden způsob jmenování. Proto jsem zadefinoval následující pravidla:

- Jména databázových objektů píšeme pouze malým písmem v anglickém jazyce oddělovačem slov v názvu, může být „_“ nebo „___“.
- Název tabulky by měl vysvětlovat její použití, označovat nějaký objekt z konkrétní domény. Pro vazební tabulky je vhodné vymyslet nějaké jméno představující tuto vazbu. Příklad: question_tag.
- Jména atributů by měla splňovat:
  - Primární klíč: <název tabulky>_id, například question_id.
  - Cizí klíče: <název atributu v cizí tabulce>.
  - Integritní omezení pro cizí klíče: fk_<název tabulky>___<název atributu>.
  - Ostatní atributy: jména by měla vysvětlovat, k čemu se tento atribut používá.
  - Oddělovač slov: „_“.
- Indexy: idx_<název tabulky>___<název atributu a nebo umělý název, pokud je index je nad více atributy>
- Sekvence: <název tabulky>___<název atributu>_seq.
- Pohledy: <název>_view.
- Funkce: <název>_func.

[1] Statická analýza kódu je metoda prozkoumání zdrojových kódů bez nutnosti jejich spuštění. Tato analýza dovoluje odhalit různé chyby v kódu: bugy, překlepy, neexistující třídy či objekty, špatné typování, mrtvý kód a další podobné problémy. [62]

4.1.4 REST API

Jelikož je REST rozhraní velmi důležité pro komunikaci klientů a serveru, jeho návrh se vždy doporučuje dělat s použitím tzv. „Best practices“. Je vhodné používat osvědčené postupy a metody, protože to usnadní budoucí údržbu tohoto rozhraní. Doporučuje se:

- Používat JSON formát pro zasílání a obdržení zpráv. XML se už moc nepoužívá a je složitější pro zpracování. [63] Jména jednotlivých hodnot samotné zprávy píšeme malým anglickým písmem s oddělovačem „_“.
- Používat podstatná jména místo sloves pro definice cest endpointů, protože jednotlivé metody HTTP protokolu už jsou ve formě sloves pro provádění nějakých akcí. [63]
- Pojmenovávat kolekce pomocí množného čísla. Například: GET /student-tests – správně, GET /student-test – špatně. [63]
- Používat návratové status kódy pro zpracování chyb. [63]
- Používat složení pro znázornění vazeb mezi entitami. Například GET /tests/testId/questions, který říká, že v rámci testu s identifikátorem testId potřebujeme získat všechny otázky. Nedoporučuje se vytvářet více než tři úrovňové složené cesty endpointů. [63]
- Doporučuje se používat filtrování, ražení a stránkování pro zpracování požadovaných dat. [63]
- Používat SSL protokol pro zabezpečení dat. [63]
- Podporovat verzování API rozhraní. Umožní to budoucí modifikaci endpointů, bez nutnosti úpravy všech na něm závislých klientů. Například: GET /v1/tests/testId/questions. [63]
- Definovat správnou a přesnou dokumentaci. V opačném případě klienti vůbec nebudou vědět, jakým způsobem se má toto API rozhraní využívat. [63]

4.2 Architektura

V předchozích kapitolách jsem popisoval rozdělení DBS portálu na více menších částí, ale neřešil jsem žádné technické podrobnosti této architektury. V této kapitole se více zaměřím na technické detaily a popíšu, jak probíhá komunikace mezi mikroslužbami, jaké úložiště budu používat, jakým způsobem budu jednotlivé mikroslužby spravovat apod.

Jelikož potřebuji spravovat velké množství malých aplikací, nejjednodušší způsob, jak to mohu dělat, je kontejnerizace. Pro každou mikroslužbu si vytvořím kontejner, který se dá jednoduše vytvořit, spustit, smazat či restartovat. Kontejnerizace je vylepšený způsob virtualizace aplikací. Na rozdíl od klasické virtualizace kontejnerizace sdílí více prostředků mezi běžícími systémy. Mohou se sdílet různé procesy operačního systémů, soubory, adresáře a spoustu dalších služeb. Umožňuje izolovat jednotlivé vrstvy a tím zaručuje vyšší bezpečnost aplikací. Procesy probíhající v rámci jednoho kontejneru nemohou ovlivnit jiné kontejnery, ani přistoupit k datům, které vlastní fyzicky stroj. Má vlastní síťové prostředí. Tyto aplikace pak lze dynamicky škálovat a snáz orchestrovat. Navíc velmi zlepšuje efektivitu práce vývojářů, protože je jednoduchá pro nastavení a spuštění. [64] Pro naše účely budeme používat Docker, protože je velmi jednoduchý pro používání a obsahuje všechny funkce, které potřebujeme. Více se lze o tomto nástroji dozvědět v podkapitole 4.6.7.

Každá mikroslužba je tvořena webovým serverem Nginx a Symfony aplikací. Diagram této architektury lze vidět na obrázku 4.1 Více se lze o tomto webovém serveru dozvědět v podkapitole 4.6.9. Jako základní obraz kontejneru jsem použil Nginx Unit, který je popsaný v podkapitole 4.6.10. Tento obraz obsahuje dohromady jak webový server, tak i prostředí pro provoz PHP aplikací, což je zde vhodné. Na základě něj jsem vytvořil nový obraz, kde jsem přidal neprivilegovaného uživatele, instaloval dodatečné knihovny, přidal PHP rozšíření, které jsou například

vyžadované pro práci s PostgreSQL databází a podobné. Navíc jsem také vložil dovnitř obrazu potřebná nastavení pro Nginx a Symfony aplikaci, které lze nalézt v příloze na výpisech kódů B.1 a B.5. Výsledný Dockerfile je zobrazen na výpisu kódu 4.1. Tento obraz je uložen do takzvaného „Container Registry“ ve školním systému Gitlab. Jelikož pomocí tohoto obrazu jsou vytvořeny kontejnery více mikroslužeb, mohl by nastat problém v případě jeho změny. Verzování vytvořených obrazů je podporováno Dockerem i naším úložištěm obrazů, což tento problém eliminuje.

```
FROM nginx/unit:1.26.1-php8.1


#================= Web user creation ==================
# Creating working dir, user and group and adding permissions to web user
RUN mkdir var/www/public && groupadd -r web && useradd --no-log-init -u 1000 -r -g web -g unit
      web && \
    chown -R web:web var/www/public
# Creating a home dir for web user
RUN mkdir -p /home/web && \
    chown web:web /home/web && \
    usermod -d /home/web web

#================ Instaling utils ====================
# Install unzip utility and libs needed by zip PHP extension
RUN apt-get update && apt-get install -y \
    zlib1g-dev \
    libzip-dev \
    unzip \
    vim
RUN docker-php-ext-install zip

# Instaling sudo
RUN apt-get -y install sudo

# Install Git
RUN apt-get update
RUN apt-get -y install git

#==================== PHP Extentions =================
# Instaling pdo driver for postgreSQL
RUN apt-get update && apt-get install -y libpq-dev && docker-php-ext-install pdo pdo_pgsql
      pdo_mysql

# Instaling debuger
RUN pecl install xdebug \
    && docker-php-ext-enable xdebug

# Install Composer
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=
      composer

#=================== Setting Locale ====================
RUN apt-get install -y locales
RUN localedef -i cs_CZ -f UTF-8 cs_CZ.UTF-8 \
 && rm -f /etc/localtime \
 && ln -s /usr/share/zoneinfo/Europe/Prague /etc/localtime

# ================== Coping Files =======================
WORKDIR /var/www/public
COPY ./php/xdebug.ini /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
COPY ./symfony/*.json /docker-entrypoint.d/

CMD ["unitd", "--no-daemon", "--control", "unix:/var/run/control.unit.sock"]

```

Výpis kódu 4.1 Dockerfile soubor pro vytvoření obrazu kontejneru mikroslužby

Kromě nových mikroslužeb, které jsem navrhl, máme v tomto projektu také již existující služby, které byly vytvořené jako samostatné projekty pro podporu DBS portálu. Proto je stačilo jen přesunout do samostatných docker kontejnerů. To je například systém RAT, jehož účelem je transformace požadavků z relační algebry do jazyka SQL. Také máme služby pro zpracování mnoha SQL požadavků. Tyto služby rozdělují jeden textový řetězec obsahující mnoho SQL dotazů na více samostatných textových řetězců obsahujících jeden SQL dotaz. To umožňuje postupné vyhodnocení SQL požadavků pomocí databáze. Jsou i jiné služby, které používá DBS portál, ale aktuálně nejsou pro nás zajímavé. Tyto služby se budou migrovat postupně, až se budou implementovat mikroslužby, které je používají.

Pro každou z mikroslužeb potřebuji nějaké úložiště pro uložení jejích dat. Jak už jsem popisoval v předchozích kapitolách, každá mikroslužba bude mít vlastní úložiště, tj. vlastní databázi. Tento přístup na jedné straně zamezí libovolným snahám budoucích vývojářů systému získat data napřímo a vynutí je tato data získávat správně, pomocí rozhraní. Na druhé straně dovolí velkou nezávislost samotných mikroslužeb, což umožní snadné škálovaní a možnost kdykoliv úplně změnit úložiště bez jakýchkoli zásahů do jiných služeb.

Budu používat dva databázové stroje: jeden pro uložení dat mikroslužeb – PostgreSQL a jiný pro uložení dat o přihlašovacích řetězcích (access tokens) uživatele – Redis. Uložení přihlašovacích údajů podrobně vysvětlím v podkapitole 4.3. Zvolil jsem databázový stroj PostgreSQL, protože se již využívá v současném DBS portálu a je jednoduchý pro používání. Kromě toho se v předmětu BI-DBS táto databáze vyučuje, proto je to vhodná volba. Jelikož budou mikroslužby používat PostgrsSQL databáze, tak mohu pro všechny tyto databáze použít jeden kontejner. V rámci něj mohu vytvořit pro každou mikroslužbu samostatnou databázi. Pro vytvoření kontejneru budu používat oficiální obraz PostgreSQL. Pro vývojové prostředí se mi bude hodit inicializační skript, který vytvoří veškeré databáze, uživatele a provede jiné potřebné nastavení. Příklad tohoto skriptu lze vidět na výpisu kódu 4.2. Tento inicializační skript se namapuje do potřebné složky kontejneru a spustí se během prvního spuštění tohoto kontejneru.

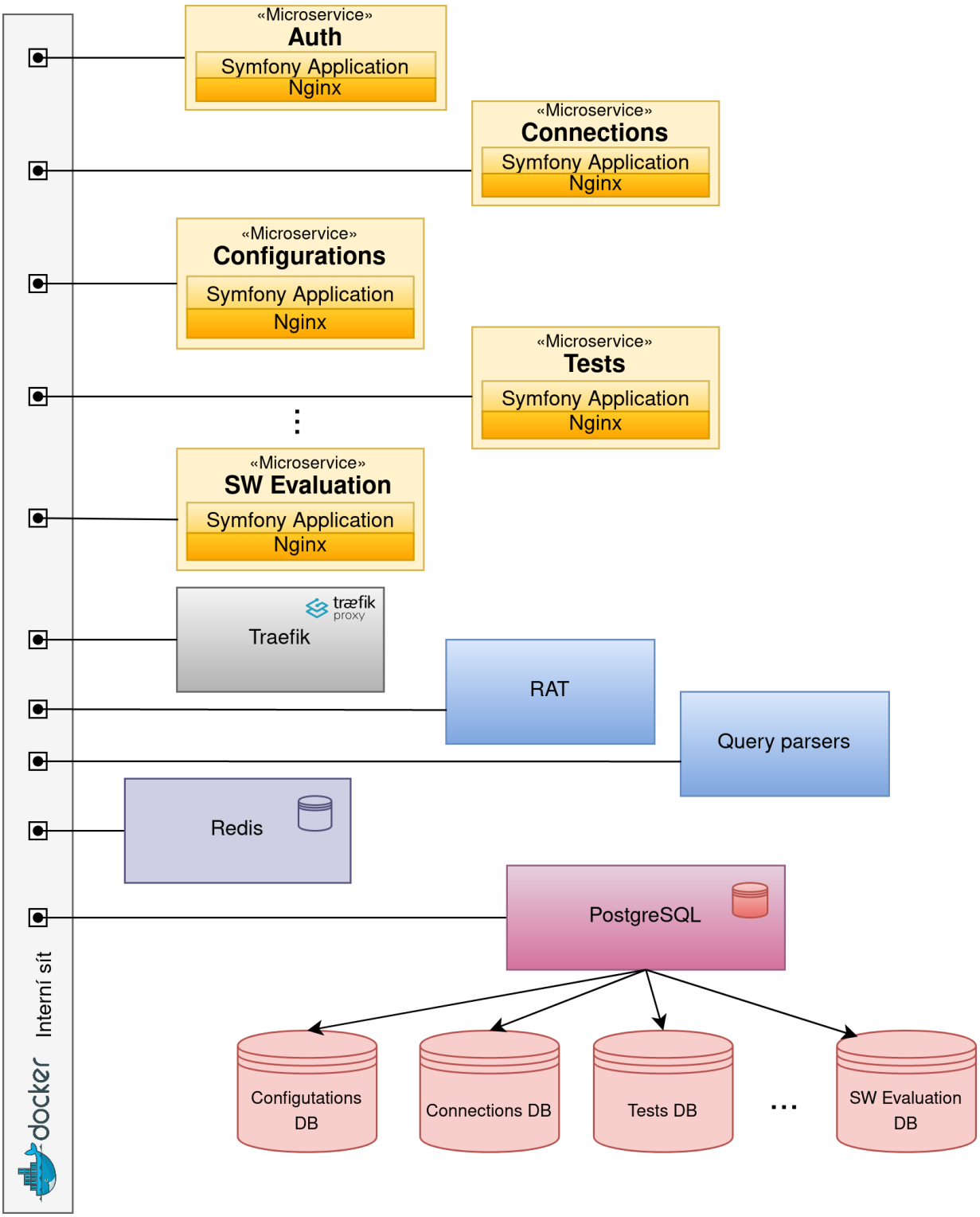
```
1  -- Vytvoření uživatele
2  CREATE USER dbs WITH PASSWORD 'dbs_pass' CREATEDB;
3  -- Vytvoření databáze pro mikroslužbu
4  CREATE DATABASE dbs_assignments  Assignmens
5      WITH
6      OWNER = dbs
7      ENCODING = 'UTF8'
8      LC_COLLATE = 'cs_CZ.utf8'
9      LC_CTYPE = 'cs_CZ.utf8'
10     TABLESPACE = pg_default
11     CONNECTION LIMIT = -1;
12
13 -- Vytvoření dalších databází
14   ...
15
```

Výpis kódu 4.2 Nastavení PostgreSQL kontejneru

Posledním kontejnerem bude služba, která se bude zabývat směrováním požadavků mezi jednotlivými mikroslužbami. Tato služba bude jediným vstupním bodem do systému a bude přebírat všechny požadavky a směrovat je dle nastavených pravidel, tj. bude představovat reverzní proxy. To dovolí v budoucnu jednoduchou výměnu existujících mikroslužeb a také zabezpečí dobrou škálovatelnost s možností vyvážení zátěže (Load Balancing). Pro tyto potřeby jsem zvolil Traefik, hlavně díky jeho jednoduchosti a snadnosti konfigurace. Jelikož se neustále mění lidé na projektu, potřebuji používat snadné nástroje, které se budou schopni rychle naučit. Více se lze o tomto nástroji dozvědět v podkapitole 4.6.8.

Traefik umožňuje vytvoření směrovací konfigurace pomocí několika různých způsobů. Zvolil jsem konfiguraci pomocí „Docker labels“[2], která je velmi jednoduchá. Kromě toho je hlavní

[2]Docker labels představuje způsob, pomocí kterého můžeme přidat nějaká metadata do konkrétního docker kontejneru. Tato data jsou ukládána jako páry(klíč, hodnota), kde můžeme zapisovat uživatelské informace. [65]



Obrázek 4.1 Architektura DBS portálu

výhodou, že nevyžaduje restartovat Traefik během přidávání nových kontejnerů, tedy systém zůstává dostupný v případě manipulací s jinými službami. Traefik je schopen dynamicky zjistit, jaké kontejnery běží a jaké mají nastavení, a dle toho začít směrovat požadavky. Například konfiguraci pro autorizační mikroslužbu je možno vidět na výpisu kódu 4.3. Na tomto nastavení lze vidět, že provádím směrování podle cesty v URL. Všechny požadavky, které budou mít ve prefixu /auth, Traefik přesměruje na autorizační mikroslužbu. Obdobně to funguje i pro ostatní mikroslužby.

```
1 LABEL "cz.cvut.fit.dbs.microservice"="Auth"
2 LABEL "traefik.enable"="true"
3 LABEL "traefik.http.routers.auth.rule"="PathPrefix(`/auth`)"
4 LABEL "traefik.http.routers.auth.entrypoints"="web"
5 LABEL "traefik.http.services.auth.loadbalancer.server.port"="9000"
6
```

Výpis kódu 4.3 Konfigurace traefik pro autorizační mikroslužbu

Traefik představuje jediný vstupní bod do našich mikroslužeb. Nemůžeme se k nim dostat napřímo, ale vždy přistupujeme pomocí tohoto proxy serveru. Všechny mikroslužby jsou připojené k interní síti typu „Bridge“. Tento druh sítě zabezpečí komunikaci pouze mezi kontejnery, které jsou připojeny k této síti. V rámci sítě je zajištěno automatické rozlišení doménových jmen mezi kontejnery. Abychom mohli volat rozhraní jiné mikroslužby, můžeme místo jména hosta uvést jméno potřebného kontejneru a port, na kterém běží tato mikroslužba. Docker si sám vyřeší správné směrování. Navíc to umožňuje izolovat služby nacházející se v různých sítích, což zabrání přístupu z jiných sítí. [66] Jediné otevřené porty, které dovolí přístup do sítě zvenku, budou porty 80 a 5432. První pro přístup k Traefik a druhý pro vzdálený přístup k PostgreSQL databázi. Diagram znázorňující tuto architekturu můžete nalézt na obrázku 4.1.

Pro přidání nové mikroslužby bude potřeba splnit jen několik kroků: vytvořit úložiště pro danou mikroslužbu, přidat ji do stejné sítě a nastavit ji správnou konfigurací pro směrování požadavků pomocí Traefik.

Abych mohl tedy zjednodušit zprovoznění vývojového prostředí pro méně zkušené lidi, vytvořil jsem docker-compose.yaml soubor. V tomto souboru jsem definoval všechny aktuálně existující mikroslužby, databáze, sítě a provedl jejich nastavení tak, aby bylo možné vytvořit vývojové prostředí během pár příkazů za několik minut. Detailní postup lze nalézt v souboru README.md v kořenové složce projektu v systému Gitlab. Ukázku tohoto postupu můžete také nalézt v příloze na obrázcích A.3 a A.4. Na ukázku souboru docker-compose.yaml se lze podrobněji podívat v příloze na výpisu kódu B.2.

4.3 Auth Mikroslužba

V této podkapitole popíšu, jakým způsobem bude vypadat autorizační mikroslužba. Definuji požadavky, vytvořím návrh a implementuji ji. Pro autorizaci uživatelů budu potřebovat, aby mikroslužba umožňovala vydávání a uložení JWT tokenů na základě dat uživatele a jeho rolí v rámci kurzu. Jak jsem už popisoval v předchozích kapitolách, bude tento token umožňovat uživateli přístup jen v rámci jednoho školního kurzu. To je nutné, aby velikost informace uvnitř přihlašovacího tokenu byla nejmenší možná. Jinak každý uživatel pracuje během semestru, až na par výjimek, pouze s jedním kurzem, proto nemá smysl zahrnovat informaci o všech kurzech do těla tokenu. Více jsem o JWT tokenu a způsobu autorizace v DBS portálu uvedl v kapitolách 3.2.4.3 a 3.2.5.

4.3.1 Definice požadavků

Softwarový požadavek specifikuje nějakou část chování systému, jeho omezení či nějakou vlastnost. Každý požadavek by měl definovat akci, aktéra (člověka, který akci provádí) a omezující kritéria. Je důležité, aby požadavky správně vymezovaly funkce systému, byly jednoznačné a přesné. Požadavky dělíme na funkční, nefunkční a požadavky na doménu. [1]

4.3.1.1 Nefunkční požadavky

- Autorizační služba by měla snížit počet požadavků na školní autorizační server a zlepšit dobu odezvy.
- Potřebujeme, aby byl počet požadavků na konfigurační mikroslužbu s účelem získání informací o uživateli minimalizován. Tedy přihlašovací token by měl obsahovat informaci o uživateli a jeho rolích v kurzu.
- Systém by měl vydávat přihlašovací tokeny s relativně krátkou dobou platností pro zajištění bezpečnosti.

4.3.1.2 Funkční požadavky

- Mikroslužba by měla umožnit vytvoření JWT tokenu. Token bude vytvořen na základě přístupového tokenu OAuth, potřebné informace o uživatelovi a jeho rolích v kurzu. Autentizace uživatele proběhne mimo naši mikroslužbu.
- Mikroslužba by měla nabízet možnost obnovení JWT tokenu po uplynutí jeho doby platnosti.
- Měla by podporovat ověření doby platnosti existujícího tokenu.
- Měla by umožňovat změnit aktuální identitu uživatele na jinou. Tato funkce bude dostupná jen pro učitele a garanta. Uživatel se nemůže přihlásit za jiného uživatele, pokud ten má stejnou roli v rámci kurzu. Administrátor se může přihlásit za libovolného uživatele.
- Mikroslužba bude umožňovat návrat původní identity uživatele v případě, že ji uživatel úspěšné měnil.

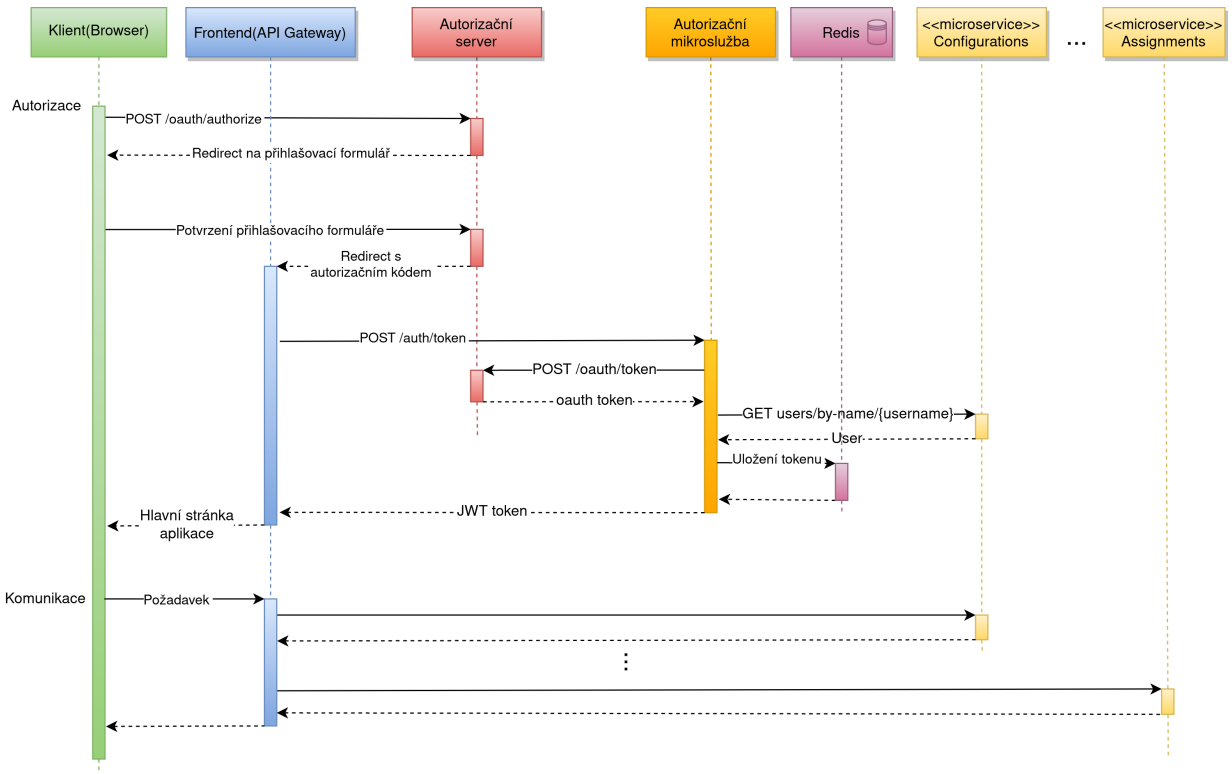
4.3.2 Návrh

Nejprve se podívám na proces generování přístupového tokenu. Jak jsem popisoval v podkapitole 3.2.5, proces autentizace se bude provádět pomocí fakultního autorizačního serveru. Autorizace se provede na základě našeho JWT tokenu, který bude vytvořen naší autorizační mikroslužbou. Klient iniciuje vytvoření přístupového tokenu dle protokolu OAuth na fakultním autorizačním serveru. Poté, co uživatel vyplní své přístupové údaje, udělí přístup naší aplikaci a získá autorizační kód, klient zavolá endpoint POST /auth/token a pošle tento kód v těle tohoto endpointu. Naše mikroslužba vygeneruje s použitím tohoto kódu přístupový token pomocí fakultního autorizačního serveru a ověří identitu uživatele. Po ověření identity se zeptá konfigurační mikroslužby na informaci o uživateli, na základě které vytvoří JWT token, který uloží do Redis databáze. Tuto databázi jsem zvolil z důvodu rychlého a jednoduchého způsobu uložení informací o vytvořeném tokenu, který je platný relativně krátkou dobu. Podrobněji si lze o Redis databázi přečíst v podkapitole 4.6.12. Po vytvoření přístupového tokenu systém vrátí klientovi odpověď ve formátu JSON, která bude obsahovat: přístupový token, obnovovací token, typ schématu a dobu expirace. Informace obsažené v této odpovědi lze vidět na výpisu kódu 4.4. Pro lepší představu tohoto procesu jsem vytvořil diagram, který je zobrazen na obrázku 4.2.

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0ayI6ImUwNjUyMzc2
    LTU0MjYtNGIyYS1iNGI3LTQ0NDZhYmI0ZmY0YSIsInVzZXIiOjUzNiwiY3JzIjoxLCJhZG0
    iOmZhbHNlLCJybHMiOlszLDJdLCJleHAiOiIyMDIyLTExLTI2VDE0OjI3OjEzKzAwOjAwIn0.
    UFFT2pyqmyzMWjXFaGMxCv8t2iT1y6iw_5NX8qr_uzw",
    "token_type": "Bearer",
    "refresh_token": "84c327da-d606-4905-b548-01c19b4800b5",
    "expire_in": 2010
}

```

Výpis kódu 4.4 Informace o vygenerovaném JWT tokenu

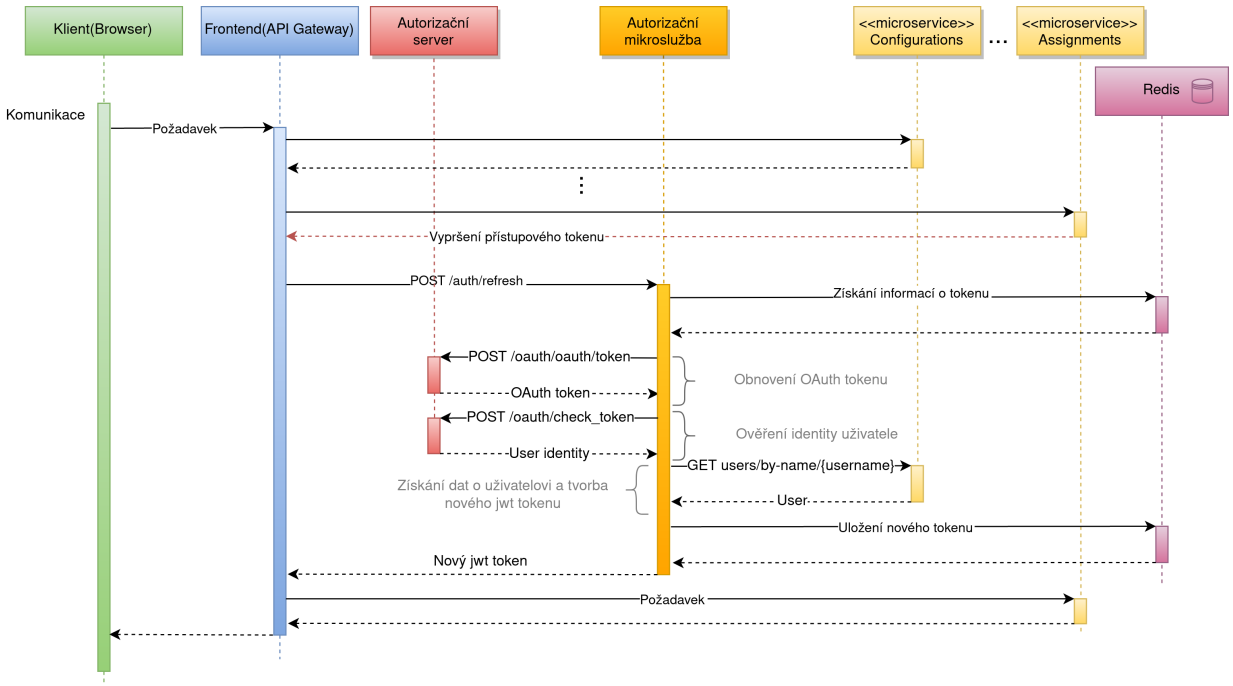


Obrázek 4.2 Proces tvorby přístupového tokenu

Po vytvoření přístupového tokenu je tento token platný po dobu 60 minut. Zvolil jsem stejnou dobu platností tokenu, jako má fakultní autorizační server. Ten vydává tokeny dle protokolu OAuth s dobou platností 60 minut. Po této době je nutné buď se znovu přihlásit, nebo obnovit tento token pomocí obnovovacího tokenu. Pokud by bylo potřeba, lze pro náš token tuto dobu platnosti změnit.

Obnovení tohoto tokenu bude probíhat pomocí obnovovacího tokenu fakultního autorizačního serveru. Klient pošle požadavek na naši mikroslužbu s cílem obnovit neplatný token. Naše mikroslužba zavolá fakultní autorizační server a požádá o obnovení tokenu. Autorizační server vygeneruje nový přístupový a obnovovací token a pošle ho naší mikroslužbě. Na základě nového tokenu vytvoří JWT token a zašle ho klientovi. Výsledkem tohoto volání bude stejná informace o přístupovém tokenu, kterou jste už viděli na výpisu kódu 4.4. Tento proces je o trochu jednodušší, protože uživatel nemusí znovu zadávat přístupové údaje. Tento proces by měl probíhat tak,

aby uživatel nebyl zklamaný častým přihlašováním během práce na projektu či při testu. Proces obnovení přístupového tokenu iniciuje služba API Gateway, kterou představuje náš frontend. Tato služba je nezbytná pro snadné získání dat z různých mikroslužeb, jejich složení, správné zobrazení a další podobné procesy. Pro lepší představu tohoto procesu jsem vytvořil diagram, který lze vidět na obrázku 4.3.

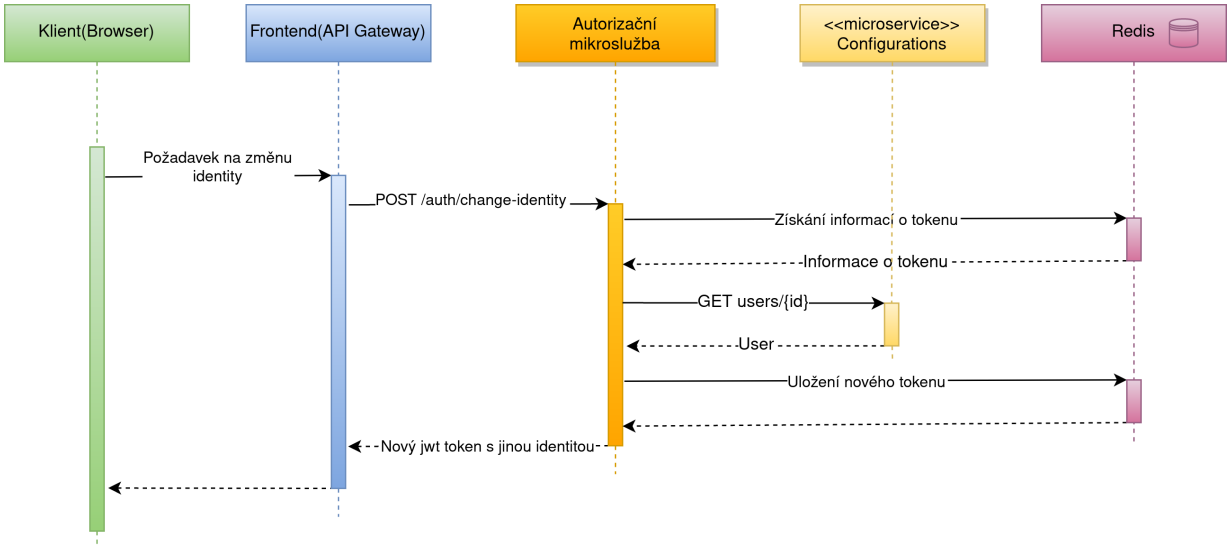


Obrázek 4.3 Proces obnovení přístupového tokenu

Pro validaci přístupového tokenu definuji endpoint: POST /auth/check-token. Tento endpoint bude v rámci požadavku přijímat náš přístupový JWT token a jako výstup posílat informace o době platnosti a další informaci o tomto tokenu. Tyto informace budou obdobné, jako už bylo možno vidět na výpisu kódu 4.4. V budoucnu se tam mohou přidat další vlastnosti, například informace o změně identity, kdo identitu změnil apod. Podrobnou dokumentaci tohoto a jiných endpointů lze nalézt na Wiki projektu v systému Redmine.

Na závěr bych chtěl definovat způsob, pomocí kterého se bude měnit identita uživatele. V DBS portálu se často stává, že se učitelé potřebují rychle podívat na práci studenta, prozkoumat problém a pomoc studentovi nějakou radou. Z tohoto důvodu potřebují mít možnost se přihlásit pomocí účtu studenta a být schopni provádět akce pod jeho identitou. Jelikož je autorizace v systému založená na přístupovém tokenu, musím vygenerovat nový JWT token, do kterého zapíšu nový identifikátor uživatele, a původní identitu si zapamatuji. Bude možné měnit identitu pouze jednou. Pokud ji uživatel bude potřebovat změnit podruhé, musí navrátit původní identitu a pak ji změnit. Tento endpoint nebude přijímat žádnou informaci v těle požadavku, ale bude obsahovat HTTP hlavičku Authorization s validním původním tokenem. Zde bude vhodná Redis databáze, do které si mohu uložit informaci o změně identity uživatele. Tato informace spolu s informací o nově vytvořeném tokenu se například uloží s klíčem: C56_U128_PU23, kde kód C značí kurz, U uživatele a PU předchozího, který změnil svou identitu. Po těchto kódech všude následuje identifikátor tohoto objektu. Pro obyčejné tokeny se klíče generují obdobně, jen bez posledního kódu PU, například: C56_U128. Dle těchto klíčů pak mohu snadno vyhledávat informace o tokenech v databázi. Endpoint: POST /auth/change-identity bude vracet stejnou informaci o tokenu jako předchozí tři.

Návrat původní identity představuje opačný proces, který vytvoří nový JWT token na základě



Obrázek 4.4 Proces změny identity uživatele

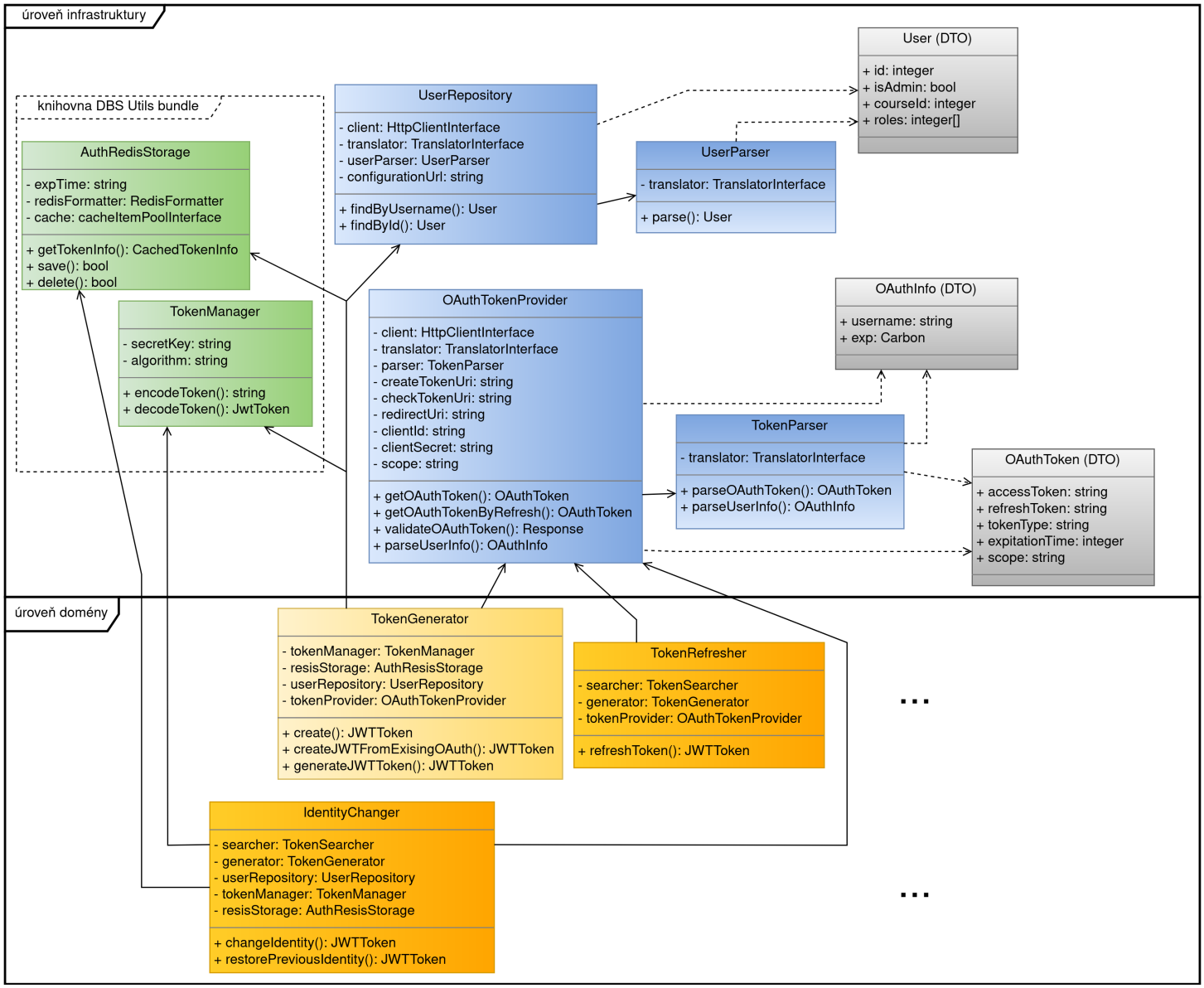
existujícího tokenu vydaného fakultním autorizačním serverem a původním uživatelem. Obdobně jako předchozí endpoint: POST /auth/restore-previous-identity bude obsahovat HTTP hlavičku Authorization a bude vracet stejnou informaci o tokenu. Podrobně se lze o těchto endpointech dozvědět pomocí dokumentace, která je definovaná s použitím specifikace Open API. Proces změny identity je zobrazen na obrázku 4.4.

4.3.3 Implementace

Tuto mikroslužbu jsem implementoval s použitím přístupů popsaných v podkapitole 3.1. Celkovou implementaci mohu rozdělit do třech úrovní. První z nich je úroveň infrastruktury. Tato úroveň je zodpovědná za komunikaci mezi mikroslužbami a fakultním autorizačním serverem. V rámci něj mám dvě služby (úložiště), jejichž účelem je získání informace o uživateli a přístupovém tokenu. Tyto informace jsou následně zpracované a uložené do DTO objektů[3]. To je důležité, protože zbytek systému je závislý pouze na těchto objektech a je mu jedno, odkud pochází. Pokud by bylo tedy potřeba změnit implementaci úrovně infrastruktury, lze to jednoduše udělat bez velkého zásahu do zbytku kódu. Na obrázku 4.5 lze vidět tuto úroveň, a jak je na ní závislá úroveň domény. Je tam také vidět, že součástí infrastruktury je DBS Utils bundle, jehož implementace je popsaná v sekci 4.4. Na tomto diagramu je tato knihovna znázorněna pouze pomocí dvou tříd, které jsou používány naší mikroslužbou. Zbytek na diagramu neuvádím, protože není důležitý.

Druhou úroveň představuje úroveň domény. Tady je uložená veškerá byznys logika aplikace. Na této úrovni mám několik služeb, které zodpovídají za hlavní funkce systému: vytvoření přístupového tokenu, jeho validaci, obnovení a také změnu identity uživatele. Každá ze služeb má svůj účel a svoji oblast zodpovědnosti, což umožňuje jejich znovupoužití. Například TokenGenerator a TokenSearcher jsou znovu použité službou TokenRefresher, která se zabývá obnovením přístupového tokenu. Tyto služby jsou malé a mají svá zaměření, která popisují procesy v rámci naší domény. Už jen z názvu lze poznat, k čemu tyto služby slouží. Tento kód pak nepotřebuje podrobné dokumentace, protože každý člověk, který rozumí procesům v rámci domény, ho bude

[3]DTO objekt je objektem, který se používá pro zapouzdření informací a jejich posílání mezi podsystémy. Typicky se používá ve víceúrovňové architektuře pro posílání dat mezi úrovněmi. Hlavní výhodou je to, že tyto objekty mohou redukovat velikost informací a obsahovat jen to, co potřebuje konkrétní služba, pro kterou je tento objekt vytvořen. Typicky nejsou modifikovatelné. [67]



Obrázek 4.5 Úroveň infrastruktury Auth mikroslužby

schopen rychle pochopit. Je to výhoda, kterou bychom chtěli mít v rámci každé mikroslužby. Diagram tříd znázorňující tuto úroveň můžete vidět na obrázku 4.6.

V rámci této mikroslužby byla většina procesů poměrně jednoduchá, proto jsem nedefinoval operační úroveň. Poslední úroveň, kterou tady mám, je úroveň uživatelského rozhraní zabývající se zpracováním požadavků a přípravou odpovědí pro klienty. Tato úroveň je představena dvěma kontroléry: ChangeIdentityController a AuthController, které také můžeme vidět na obrázku 4.6. Tyto kontroléry definují řadu endpointů, které představují funkce umožnující přihlášení uživatele a změnu jeho identity.

Abych mohl tuto mikroslužbu správně používat, potřebuji nakonfigurovat několik proměnných prostředí a nastavit některé konfigurační soubory. Jak lze vidět na diagramu, služba OAuthTokenProvider obsahuje proměnné s informacemi potřebnými pro provádění autentizace na autorizačním serveru. Tady lze vidět, že není nutné používat fakultní autorizační server, ale mohl bych využít libovolný jiný, který podporuje OAuth protokol. Jediné, co bychom musel změnit, jsou třídy UserParser a TokenParser. Tyto třídy se zabývají zpracováním odpovědí z tohoto autorizačního serveru, proto jsou specifické pro konkrétní autorizační server.

V konfiguračním souboru services.yaml potřebuji definovat nastavení třídních proměnných. Tyto proměnné nastavím pomocí proměnných prostředí, které bude možné v budoucnu změnit dle potřeby. Toto nastavení je možné vidět na výpisu kódu 4.5.

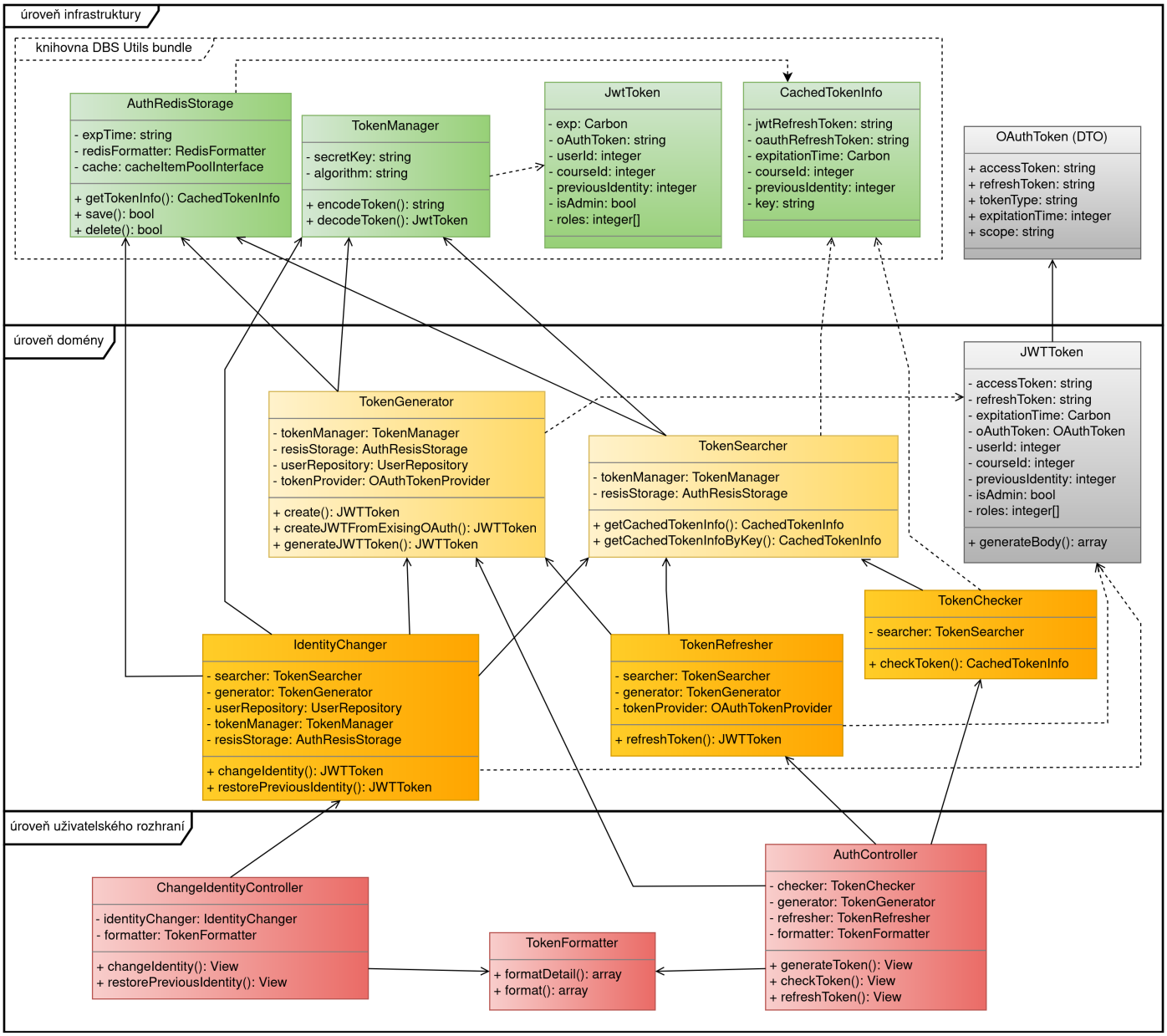
```
1  # OAuth token provider
2  App\Model\Repository\OAuthTokenProvider:
3      arguments:
4          $createTokenUri: '%env(FIT_CREATE_TOKEN_URI)%'
5          $refreshTokenUri: '%env(FIT_REFRESH_TOKEN_URI)%'
6          $checkTokenUri: '%env(FIT_CHECK_TOKEN_URI)%'
7          $redirectUri: '%env(FIT_REDIRECT_URI)%'
8          $clientId: '%env(FIT_CLIENT_ID)%'
9          $clientSecret: '%env(FIT_CLIENT_SECRET)%'
10         $scope: '%env(FIT_SCOPE)%'
11
12 App\Model\Repository\UserRepository:
13     arguments:
14         $configurationUrl: '%env(DBS_CONFIGURATION_GET_USER_URL)%'
15
```

Výpis kódu 4.5 Ukázka konfigurace mikroslužby v souboru services.yaml

V rámci autorizační mikroslužby mám následující proměnné prostředí:

- FIT_CREATE_TOKEN_URI – proměnná uchovávající URL adresu zdroje pro generování přístupového tokenu na autorizačním serveru.
- FIT_REFRESH_TOKEN_URI – proměnná uchovávající URL adresu zdroje pro obnovení přístupového tokenu na autorizačním serveru.
- FIT_CHECK_TOKEN_URI – URL adresa zdroje pro ověření přístupového tokenu.
- FIT_REDIRECT_URI – URL adresa zdroje, kam se bude provádět přesměrování po schválení formuláře s přístupovými údaje.
- FIT_CLIENT_ID – identifikátor naší aplikace pro autorizační server.
- FIT_CLIENT_SECRET – tajný klíč naší aplikace pro autorizační server.
- FIT_SCOPE – definuje oblast, ke které bude udělen přístup.
- DBS_CONFIGURATION_GET_USER_URL – URL adresa zdroje, který umožňuje získat data o uživateli z konfigurační mikroslužby.

Dokumentace této mikroslužby je vytvořená pomocí Open API specifikace a je ji možno nalézt ve školním systému Gitlab, v projektu DBS-MicroServices. Po nasazení do provozu bude tato dokumentace dostupná online jako webová stránka.



Obrázek 4.6 Úroveň uživatelského rozhraní a úroveň domény Auth mikroslužby

4.3.4 Testování

Pro každou aplikaci je nutné vědět, zda správně plní své funkce a funguje dle očekávání. Proto potřebuji zajistit, aby naše aplikace splňovala vytvořenou specifikaci. Toto mohu udělat pomocí manuálních a automatizovaných testů. Na rozdíl od monolitní architektury je zde testování velmi důležité, protože mám velkou sadu služeb, které mohou být nějakým způsobem mezi sebou závislé. V tomto případě může člověk, který nemá zkušenosti se systémem, způsobit změnu, která bude mít vliv na nějakou jinou část systému a tím může zapříčinit problémy. Proto se budu v rámci této podkapitoly zabývat testováním naší autorizační mikroslužby. Potřeboval bych zajistit základní kvalitu produktu, eliminovat způsobení škody kvůli neopatrným změnám a celkově zaručit bezpečnost budoucího rozvoje systému.

Obecně provádíme testování z několika důvodů[68, 69]:

- Chceme vyzkoušet, jak se systém chová v reálném provozu.
- Chceme ověřit, zda systém dělá to, co má, a funguje dle stanovené specifikace.
- Chceme vyzkoušet spolehlivost a dostupnost systému.
- Chceme změřit kvalitu vytvořeného programu.
- Chceme analyzovat software a nalézt neobjevené chyby.
- Chceme ověřit, zda systém dosahuje požadovaných či akceptovaných výsledků.
- Chceme mít jistotu, že nově přidaný kód neovlivní fungující části aplikace.
- Chceme snížit náklady na budoucí údržbu a problémy spojené se závažnými chybami.

Jak lze vidět, existuje mnoho různých cílů testování, a proto si musím nejprve ujasnit, co potřebuji testovat, jakým způsobem to lze testovat a co je cílem tohoto testování. [68] V tomto případě potřebuji automatizované testy, které nám do nějaké míry řeknou, že přidaná rozšíření či změny v systému nezpůsobily nějakou závažnou chybu. Vývojáři potřebují mít možnost, pomocí které je každý z nich schopen během několika minut ověřit hlavní funkce mikroslužby. Tyto testy se nazývají automatizované regresní testy. Softwarové testy lze dělit dle několika různých dimenzí. Zde jsem použil dimenzi kvality a rozdělil jsem je do následujících skupin [70]:

- Funkční testy – testují konkrétní byznys funkce systému:
  - unit testy,
  - integrační testy,
  - smoke testy,
  - regresní testy,
  - akceptační testy,
  - lokalizační testy,
  - globalizační testy,
  - testy interoperability.
- Výkonové testy – testují výkon systému, například zkoumají, jaký počet požadavků zvládne obsloužit náš systém za jednotku času.
- Testy kompatibility – ověří, zda může být systém kombinován s jiným softwarem nebo hardwarem.
- Testy použitelnosti – testují, zda budou uživatelé schopni náš systém používat.
- Testy dostupnosti a spolehlivosti – ověří, zda se systém chová stejně za libovolných okolností, například pokud nastala chyba či nějaký výpadek.
- Testy bezpečnosti – testují přístup k systému a zda není možné se dostat do uložených informací bez přihlášení do systému pomocí svého účtu.
- Testy udržitelnosti – testují, zda lze systém udržovat a rozšiřovat.

4.3.4.1 Testování Auth mikroslužby

V rámci mikroslužeb se vždy definuje veřejné rozhraní, které může být použito libovolnou z ostatních mikroslužeb. Proto je v rámci testování vhodné testovat toto rozhraní. Z tohoto důvodu budu naše testy provádět nad veřejným rozhraním autorizační mikroslužby. Jelikož mě nezajímá vnitřní implementace, budu používat black-box testy API rozhraní. Pro vytvoření samotných testů budu používat integraci frameworku Symfony s knihovnou PHPUnit, která se používá pro tvorbu testů v jazyce PHP. Instaluji do našeho projektu balíček symfony/test-pack, se kterým se instalují všechny potřebné knihovny pro tvorbu testů, a pomocí Symfony Flex vytvořím veškeré potřebné konfigurace pro zahájení testování. Také budu potřebovat nastavit proměnné testovacího prostředí v konfiguračním souboru .env.test.

Jelikož naše mikroslužba volá externí systém (FIT Auth server), potřebuju tento systém nějak nahradit. To mohu udělat pomocí vytvoření mocku[4] API externího systému. Potřebuji vytvořit několik zdrojů (endpointů), které budou simulovat provoz FIT Auth serveru. Proto si zadefinuji tyto zdroje a simulované odpovědi k nim:

- Zdroj: POST /tests/oauth/token:
  - Bude se používat pro tvorbu přístupového tokenu.
  - Odpověď ve formátu JSON je zobrazená na výpisu kódu 4.6.
- Zdroj: GET /tests/oauth/check-token?token=access_token:
  - Bude se používat pro ověření přístupového tokenu.
  - Odpověď ve formátu JSON je zobrazená na výpisu kódu 4.7.
- Zdroj: POST /tests/oauth/token:
  - Bude se používat pro obnovení přístupového tokenu.
  - Odpověď ve formátu JSON je zobrazená na výpisu kódu 4.8.
- Zdroj: GET /users/by-name/username:
  - Bude se používat pro získání informací o uživateli.
  - Odpověď ve formátu JSON je zobrazená na výpisu kódu 4.9.

Pro simulaci těchto endpointů jsem použil stránku mockapi.io, která nabízí možnost simulací několika endpointů pro jednu aplikaci zdarma, což je pro nás vyhovující. Navíc toto umožňuje jednoduše nakonfigurovat potřebné zdroje během pár kliků. Potom stačí jen správně nastavit proměnné prostředí a simulace externí aplikaci je připravená. Poté mohu začít s tvorbou testů.

```
{
  "access_token": "OauthToken-Admin",
  "token_type": "Bearer",
  "refresh_token": "cef9be9b-b10f-443c-ab16-5766cd777a33",
  "expires_in": 354545245,
  "scope": "cvut:umapi:read"
}

```

Výpis kódu 4.6 Odpověď pro POST tests/oauth/token

[4]Mock objeckt je objektem simulujícím objekt z reálného systému, ale nabízejícím pouze omezenou a předem definovanou sadu funkcí a výsledků, které jsou vyžadované v rámci konkrétního testovacího případu. Vytvoření takových objektů se nejčastěji používá pro tvorbu jednotkových testů. [71]

```
{
  "exp": 2889998780,
  "user_name": "testallroles",
  "authorities": [
    "ROLE_USER"
  ],
  "client_id": "cac560d3-55a4-4793-a147-2d03cfd2f73e",
  "scope": [
    "cvut:umapi:read"
  ]
}

```

Výpis kódu 4.7 Odpověď pro GET ests/oauth/check-token

```
{
  "access_token": "OauthToken-Admin-Refreshed",
  "token_type": "Bearer",
  "refresh_token": "cef9be9b-b10f-443c-ab16-5766cd777a33",
  "expires_in": 354545245,
  "scope": "cvut:umapi:read"
}

```

Výpis kódu 4.8 Odpověď pro POST ests/oauth/check-token

```
{
  "id": 1,
  "name": "John",
  "surname": "Doe",
  "username": "doejoh",
  "email": "john.doe@example.com",
  "language": "cs",
  "last_login": "2022-12-04T18:36:03.726Z",
  "type": "normal",
  "admin": true,
  "current_course_id": 1,
  "roles": [
    {
      "id": "2",
      "course_id": 1
    },
    ...
  ],
  ...
}

```

Výpis kódu 4.9 Odpověď pro GET /users/by-name/username

4.3.4.2 Testovací případy

Testovací případ definuje konkrétní postup, který se provádí s konkrétní softwarovou komponentou, modulem či službou. Testovací případy se vytváří jak pro automatizované, tak i pro manuální testy. Pro manuální testování se tvoří sada konkrétně definovaných kroků a očekávaných výstupů programu. Automatizované testování se definuje pomocí provádějícího skriptu, který automaticky může rozpoznat, zda test selhal, nebo proběhl úspěšné. Testovací případ je tedy dokument popisující určitý proces v rámci systému, který potřebujeme otestovat. Definuje potřebné kroky pro provedení spolu se vstupními a výstupními hodnotami. [72]

Procesy v rámci autorizační mikroslužby nejsou složité, a proto zkusím jednoduše zadefinovat několik testovacích případů, které budu implementovat. Předpokládám, že mám vytvořenou simulaci FIT Auth serveru pomocí existujících nástrojů. Dále si definuji několik testovacích případů:

- **Vytvoření přístupového tokenu**.
  - Úspěšná tvorba přístupového tokenu: v rámci testu se zavolá POST /auth/token a vytvoří přístupový token. Výsledkem bude odpověď se status kódem 200 a validní JWT token, který bude umožňovat přístup k naší aplikaci.
  - Neúspěšná tvorba přístupového tokenu: v rámci testu se zkusí zavolat POST /auth/token a vytvořit přístupový token, ale uživatel, který se snaží přihlásit do systému, chybí v konfigurační mikroslužbě. Výsledkem bude odpověď se status kódem 409, v které bude informace o tom, že uživatel chybí.
- **Obnovení přístupového tokenu**.
  - Úspěšné obnovení tokenu: v rámci testu se provede POST /auth/refresh-token a obnoví přístupový token. Výsledkem bude odpověď se status kódem 200 a nový validní JWT token, který bude umožňovat přístup k naší aplikaci.
  - Neúspěšné obnovení tokenu: v rámci testu se zkusí zavolat POST /auth/refresh-token a obnovit přístupový token, ale v rámci požadavku pošle špatný obnovovací token. Ve výsledku mikroslužba vrátí odpověď se status kódem 409 a informaci o chybě.
- **Změna identity uživatele**.
  - Úspěšná změna identity: v rámci testu se zavolá POST /auth/change-identity a změní identitu uživatele. Výsledkem bude odpověď se status kódem 200 a nový JWT token umožnující přístup pod novou identitou.
  - Neúspěšná změna identity: v rámci testu se zkusí zavolat POST /auth/change-identity a změnit identitu uživatele, ale uživatel nebude mít vyžadovanou roli. Výsledkem bude odpověď se status kódem 409, v rámci které bude informace o tom, že uživatel nemůže tuto akci provést.
- **Vrácení původní identity uživatele**.
  - Úspěšné obnovení původní identity: v rámci testu se zavolá POST /auth/restore-previous-identity a vrátí původní identitu uživatele. Výsledkem bude odpověď se status kódem 200 a nový JWT token umožnující přístup pod původní identitou uživatele.
  - Neúspěšná změna identity: v rámci testu se zkusí zavolat POST /auth/restore-previous-identity, ale uživatel předtím neměnil svou identitu. Proto systém musí vrátit odpověď se status kódem 409 a informaci o vzniklé chybě.

4.3.4.3 Výsledky testování

Tato mikroslužba byla otestovaná jak manuálními, tak i automatizovanými testy. Manuální testy jsem prováděl osobně spolu se studenty pracujícími na projektu v rámci SP týmů. Vzhledem k tomu, že je tato mikroslužba poměrně malá, nenalezl jsem tady velké množství chyb. Podařilo se ale objevit problém se zpracováním vzniklých chyb spojených s nedostupností jiných systémů, nebo s chybným vyplněním vstupních dat klientem. V případě, že se vytvoření nebo obnovení přístupového tokenu nepodařilo, klient nevěděl, kvůli čemu se to stalo. Proto jsem opravil zpracování chyb a teď aplikace vrací nejenom HTTP status kód chyby, ale posílá i podrobnou informaci o chybě, která nastala. Jako součást této mikroslužby jsem také otestoval knihovnu DBS Utils bundle, protože velká část jejích funkcí je spojená s autorizacemi uživatelů. Tato knihovna je podrobně popsaná v následující podkapitole.

Pro bezproblémový vývoj jsem připravil několik testovacích přístupových tokenů, které mohou použít vývojáři pro manuální testování systému. Každý následující token má větší práva v systému. V systému jsou definované následující role: student, učitel, garant a administrátor. Nejmenší práva v systému má role nejvíce vlevo, největší – nejvíce vpravo, což je administrátor. Pro každou z těchto rolí jsem definoval přístupový token, který je platný neomezenou dobu. Podrobně jsou tyto testovací přístupové tokeny popsány v rámci dokumentace projektu v systému Redmine.

4.4 DBS Utils bundle

Každá z mikroslužeb má vždy nějaké společné funkce s jinými mikroslužbami, což může způsobovat duplikaci zdrojového kódu. Abych mohl tento problém odstranit, potřebuji vyjmout tyto funkce ven a implementovat je jako samostatný modul. Tento modul by měl být jednoduše integrovatelný s libovolnou z našich mikroslužeb a tím redukovat zbytečnou duplikaci kódu. Z tohoto důvodu jsem se rozhodl implementovat knihovnu pro Symfony framework, která by tyto funkce sjednotila a umožnila jejich použití v libovolné z mikroslužeb.

4.4.1 Definice požadavků

4.4.1.1 Nefunkční požadavky

- Tento modul musí být knihovnou, která se dá integrovat pomocí nástroje pro správu balíčku v PHP (Composer). Více se lze o tomto nástroji dozvědět v podkapitole 4.6.2.
- Musí obsahovat veškeré společné funkce využívané mikroslužbami.
- Musí být vytvořen pro Symtony framework, tedy jako Symfony bundle.
- Měl by obsahovat podrobnou dokumentaci, vysvětlující veškeré jeho funkce.

4.4.1.2 Funkční požadavky

- Tato knihovna bude obsahovat třídu AbstractController, která dědí od třídy AbstractFOSRestController z knihovny FOS RestBundle. Tato třída bude obsahovat metody pro zpracování HTTP odpovědí:
  - Metoda pro zaslání úspěšně zpracovaného požadavku. Tato metoda přijímá data a posílá je se stavovým kódem 200.
  - Metoda pro zaslání informace o úspěšně vytvořeném objektu. Jejím úkolem je poslat zdroj pro získání tohoto objektu v HTTP hlavičce Location se stavovým kódem 204.

  - Metoda pro zaslání informací o špatném požadavku. Bude mít dvě varianty: jedna přijímá identifikátor zprávy, přeloží ji dle nastaveného jazyka uživatele a pošle chyby se stavovým kódem 400. Druhá jen pošle tyto chyby, bez dodatečné zprávy.
  - Metoda pro zaslání informace o chybě při zpracování vstupních dat v těle požadavku. Tato metoda převezme Symfony formulář, identifikuje nastalé chyby, zpracuje je a pošle spolu se stavovým kódem 400.
  - Metoda pro oznámení neexistence objektu. Jejím účelem je poslat informaci, že objekt neexistuje, spolu se stavovým kódem 404.
  - Metoda pro zaslání informace o nastalé chybě během zpracování požadavku. Tato metoda posílá informaci o problému spolu se stavovým kódem 409.
- Měla by obsahovat implementaci některých společně využívaných typů pro Symfony formuláře. Tyto formuláře se používají pro zpracování zaslaných informací od klienta v JSON formátu.
  - AbstractFormType – umožní vytvářet objekty reprezentující informace z požadavku klienta. Jednoduché rozšíření Symfony AbstractType.
  - CypherTextType – typ umožnující šifrování zaslaného textu.
  - CarbonType – typ, který dovolí používat objekty z knihovny Carbon pro datum a čas.
- Metody pro práci s Redis databází: uložení, získání a smazání informací spojených s přístupovým tokenem uživatele.
- Autorizace uživatelů. Tato část knihovny by měla implementovat:
  - DbsAuthenticator – třída umožnující autentizaci pomocí knihovny Symfony Security.
  - UserProvider – třída umožnující získání uživatele a validaci přístupového tokenu v rámci Symfony Security knihovny.
  - Definice jednotlivých rolí v DBS portálu.
  - Anotace, která dovoluje validaci uživatelských rolí v systému.
  - TokenManager zodpovědný za transformaci tokenu z řetězce na objekt a opačně. Také umožňuje jeho validace.

4.4.2 Implementace

Tuto knihovnu jsem implementoval jako sadu nezávislých funkcí, které lze integrovat do libovolné z mikroslužeb. Představuje tak zvaný Symfony bundle[5], který se jednoduše integruje do Symfony aplikací. Tuto knihovnu je možno nalézt ve školním systému Gitlab s názvem DBS-Microservices-utils-bundle.

Samotná konfigurace knihovny se provádí pomocí proměnných prostředí a některých konfiguračních souborů. Hlavní konfigurační soubor knihovny je services.xml, který definuje, jakým způsobem má probíhat DI proměnných prostředí a služeb v naší knihovně. Tento soubor si lze prohlédnout v příloze na výpisu kódu B.4. Dále potřebuji nastavit několik proměnných prostředí, která zajistí správné fungování Redis databáze a ověření vydávaných tokenů. Bohužel je třeba tyto proměnné nastavit v každé mikroslužbě, což není dobré, ale má to následující výhodu: během ověření přihlašovacího tokenu se nemusí zbytečně volat autorizační mikroslužba, nýbrž token se může validovat pomocí tohoto rozšíření. Všechny nastavené proměnné by měly být stejné napříč všemi mikroslužbami. Je potřeba nastavit následující proměnné:

[5]Symfony Bundle představuje jednoduchou knihovnu, kterou lze jednoduše integrovat se Symfony aplikacemi. Navíc můžeme tuto knihovnu používat jen pro konkrétní prostředí, například vývojové prostředí. Samotný Symfony Framework se skládá ze sady takových dohromady spojených knihoven. Je to jednoduchý způsob, jak přidat nové funkce do aplikace. [73]

- REDIS_URL – url, kde je dostupná Redis databáze.
- REDIS_PREFIX – klíč, na základě kterého se generuje prefix v Redis databázi.
- REDIS_EXPIRATION_TIME – doba, po kterou je dostupný záznam v databázi.
- TOKEN_SECRET_KEY – tajný klíč, potřebný pro vytvoření a ověření podpisu přístupových tokenů.
- TOKEN_ALGORITHM – algoritmus, který se používá v době generování a ověření přístupových tokenů.

Aby všechno fungovalo, potřebuji nastavit i několik konfiguračních souborů, které patří jiným knihovnám. První z nich je soubor cache.yaml. Tam je potřeba definovat, že používám pro cache Redis databázi. Navíc je potřeba v souboru services.yaml definovat službu redis_provider. Ukázka tohoto nastavení je na výpisu kódu 4.10.

```yaml
# cache.yaml
framework:
    cache:
        app: cache.adapter.redis
        default_redis_provider: 'redis_provider'
        prefix_seed: '%env(REDIS_PREFIX)%'


# services.yaml

  ...
# Redis provider
    redis_provider:
        class: \Redis
        factory: [ 'Symfony\Component\Cache\Adapter\RedisAdapter', 'createConnection' ]
        arguments:
            - '%env(REDIS_URL)%'
            - {persistent: 1, timeout: 30, port: 6379, retry_interval: 0, tcp_keepalive: 0}
```

Výpis kódu 4.10 Konfigurace DBS Utils bundle

Dále potřebuji nastavit hierarchii uživatelských rolí, přístupy k nim a další potřebné nastavení. Tato nastavení vyžaduje knihovna Symfony Security, na základě které jsem vytvořil DbsAuthenticator a UserProvider. Ty potřebuji nakonfigurovat v konfiguračním souboru security.yaml. Ukázka konfigurace je na výpisu kódu 4.11.

```yaml
# security.yaml

security:
    enable_authenticator_manager: true
    role_hierarchy:
        ROLE_ADMIN: [ ROLE_USER, ROLE_GUARANTOR, ROLE_STUDENT ]
        ROLE_GUARANTOR: [ ROLE_TEACHER, ROLE_LECTOR ]
        ROLE_TEACHER: ROLE_USER
        ROLE_LECTOR: ROLE_USER
        ROLE_STUDENT: ROLE_USER
    providers:
        app_user_provider:
            id: Dbs\UtilsBundle\Security\Provider\UserProvider
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            pattern: ^/
            provider: app_user_provider
            custom_authenticators:
                - Dbs\UtilsBundle\Security\Authenticator\DbsAuthenticator
            stateless: true
```

```
24
25    access_control:
26        - path: ^/auth/check-token
27          roles: ROLE_USER
```

Výpis kódu 4.11 Konfigurace DBS Utils bundle pro zabezpečení autorizace

Veškerou dokumentaci k této knihovně jsem definoval v souboru README.md v projektu této knihovny v školním systému Gitlab. V budoucnu se tato knihovna doplní o další potřebné funkce společné pro více mikroslužeb, které se objevily v průběhu implementace samotných mikroslužeb. Je to například získání objektů z jiných mikroslužeb, které se často používají, apod. Tato knihovna hodně zjednodušila práci SP týmům a umožnila jim mnohem rychleji vyvíjet potřebné funkce. Například pro přidání autorizace do nové mikroslužby je potřeba pouze instalovat toto rozšíření a přidat konfiguraci. Poté může vývojář jednoduše přidávat atribut #AllowRole(['some_role']) do potřebných endpointů a tak nastavovat přístupy k nim.

4.5 Výsledky práce a budoucí rozvoj DBS portálu

Během práce jsem se snažil co nejvíce zjednodušit budoucí vývoj DBS portálu v rámci SP projektů. Navrhl jsem několik řešení různých problémů spojených s organizací práce v rámci SP týmu. Tato řešení jsme mohli hned vyzkoušet. Některá z nich nebyla dobrá, například report o stavu projektu, který lidé nebyli ochotni provádět a který částečně duplikoval funkce systému Redmine. Jiné měly významný vliv na produktivitu práce, například zaškolení lidí pomocí nového testovacího projektu. Tento testovací projekt jim pomohl rychle se naučit základní práce s technologiemi na projektu, čímž hodně zvýšil produktivitu jejich práce. Na základě zpětné vazby od studentů jsem ho vylepšil, což bude mít v budoucnu pozitivní dopad na zaškolení dalších lidí. Také jsem vytvořil podrobnou dokumentaci objektů a procesů pro doménu DBS portálu - to by mělo zjednodušit pochopení systému. Pro novou architekturu DBS portálu jsem navrhl a připravil veškeré podklady a podrobný popis postupů, konvencí, mikroslužeb apod.

Systém se začal postupně migrovat do mikroslužeb a v rámci SP týmů byly vytvořeny mikroslužby „Configurations“, „Connections“ a „SW-Configurations“. Každý z SP týmů se zabýval vývojem jedné z mikroslužeb, což velmi snížilo nutnost porozumění velké části systému najednou. Před začátkem práce na mikroslužbě si každý člen týmu vyzkoušel implementovat zkušební projekt. V tomto projektu si student vyzkoušel základní možností frameworku Symfony, jazyku PHP, tvorbu dokumentace a práci s nástroji: Docker, GIT, Composer, Postman a další. Pomocí knihovny DBS Utils Bundle SP týmy mohly jednoduše přidat důležité funkce ke svým systémům, čímž ušetřily velké množství času, například autorizaci uživatelů, formátovaní odpovědí na požadavek a další. To mělo svoje výhody, protože SP tým byl schopný velmi rychle navrhnout dobré řešení, implementovat ho a pak řešit integraci s jinými mikroslužbami.

Osobně jsem se zabýval validací řešení SP týmů, celkovým řízením vývoje mikroslužeb, jejich integrací a testováním. V průběhu validace kódu jsem se snažil vysvětlit výhody a nevýhody použití některých konstrukcí, upozornit na závažné chyby a doporučit nějaká zlepšení kódu, což mělo pomoct studentům pochopit používání technologií na projektu. Někteří studenti z těchto SP týmů se ke konci předmětu BI-SP2 naučili perfektně používat framework Symfony a jejich výstup byl velmi kvalitní.

Současně přišli zájemci, kteří chtějí ve svých závěrečných pracích pracovat na DBS portálu. Tito studenti si zvolili část mikroslužeb, které odpovídají za vytvoření, napsání a hodnocení testů, a postavili na tom svoje závěrečné práce. Také ze strany frontendu máme několik prací spojených s implementací nového frontendu používajícího naše mikroslužby. V rámci těchto závěrečných prací by měl být dokončen vývoj mikroslužeb zodpovědných za testy a také vývoj velké části frontendu pro DBS portál. Proto budeme pro další SP týmy, které se objeví v příštím semestru, nabízet vývoj mikroslužeb odpovídajících za vypracování semestrálních prací. Každý z týmů by měl zvládnout implementovat jednu mikroslužbu a integrovat ji do celku. Pro ně jsem připravil

za prvé podrobnou dokumentaci, jak začít s vývojem, a definoval různé konvence a postupy, za druhé jsem vytvořil testový projekt, ve kterém si studenti mohou vyzkoušet technologie, naučit se nějakým základním konstrukcím jazyka PHP a seznámit se s frameworkem Symfony. Také mohou použít slovník pojmů a procesů, kde jsem definoval všechny společné termíny pro naše domény. To pomůže studentům, neznajícím doménu DBS portálu, rychleji se zorientovat a porozumět projektu. Pro zjednodušení vývoje jsem také implementoval knihovnu DBS Utils bundle, která nabízí mnoho společných funkcí. Například tato knihovna dovoluje pomocí jednoho řádku kódu přidat autorizaci k vytvořeným endpointům a nakonfigurovat přístup dle potřebných rolí uživatele.

Dále bude potřeba migrovat existující data ze současné databáze DBS portálu do jednotlivých databází mikroslužeb. Migrace dat není jednoduchou činností, proto jsem také zadefinoval postup, pomocí kterého mohou studenti v rámci SP týmů postupovat a tuto migraci provádět. Samotnou migraci mohou vytvořit pomocí specializovaného nástroje, jazyka SQL či jiného programovacího jazyka. V této práci jsem definoval datovou mapu pro migraci testových šablon do mikroslužby zodpovídající za testové šablony, kterou lze použít jako inspiraci.

Myslím, že jsem připravil veškeré podklady pro rychlé zorientování se v projektu, jeho konfiguraci a zahájení efektivního vývoje, což původně chybělo. Tato dokumentace je vytvořena pomocí „Wiki“ v systému Redmine. Ukázku hlavní stránky dokumentace projektu lze vidět v příloze na obrázku A.5. Také se na rozdíl od současného DBS portálu budou používat nové technologie a knihovny. To by mělo mít pozitivní vliv na podporu projektu a jeho rozvoj. Navíc pokud by se v budoucnu zjistilo, že se jedna z mikroslužeb dostala do stavu jako současný DBS portál, můžeme ji jednoduše upravit nebo vyměnit. Tato hlavní výhoda mikroslužeb dovoluje vylepšit libovolnou část systému bez zásahu do zbytku.

Velkou výhodou navrženého řešení je, že již není tento systém tak závislý na fakultním prostředí, ale může být jednoduše rozšířen a použit i jinou vysokou školou, kde se vyučují databázové systémy. Auth mikroslužba je implementovaná tak, že dovoluje jednoduše změnit poskytovatele uživatelských identit pomocí drobných rozšíření. Jediné, co by bylo potřeba dopracovat, je import uživatelů, kurzů a paralelek z jiného studijního systému, než je studijní systém KOS, nebo implementovat manuální tvorbu těchto objektů v konfigurační mikroslužbě.

4.6 Použité technologie a nástroje

4.6.1 Jazyk PHP

PHP (Hypertext Preprocessor) je interpretovaným programovacím jazykem, který se používá především pro tvorbu dynamických webových stránek a webových aplikací. Jazyk PHP lze také používat pro vytvoření různých skriptů, konzolových či desktop aplikací. Tento jazyk je jedním z nejpoužívanějších skriptovacích jazyků pro tvorbu webových stránek a aplikací. Díky své jednoduchosti je nejvíce využívaný pro malé a střední weby. Je také vhodný i pro velké projekty, například jsou v tomto jazyku implementovány Facebook a Wikipedie. [74]

PHP je často volen díky specializaci na webové stránky, velkému množství funkcí v základní knihovně PHP, podpoře na většině hostingových službách, strmé křivce učení, obrovskému množství existujících projektů a kódů, které lze zdarma použít, velmi svobodné licenci a dalším výhodám. Tento jazyk má také nevýhody, jako je například nekonzistentní pojmenování funkcí, neudržení kontextu aplikace (snížení výkonu), složitost přechodů mezi verzemi jazyka. Například do verze 8 chybělo striktní vyžádání typů proměnných a mnoho dalších. [74, 75]

4.6.2 Composer

Composer je nástroj používající se pro správu knihoven a jiných zdrojů v jazyku PHP. Ovládá se pomocí příkazové řádky a je multiplatformní. Lze pomocí něj definovat, které knihovny a

závislosti projekt využívá. Umožňuje jejich instalaci (řídí automatické nahrávaní tříd, tzv. class autoloading), kontrolu konfliktu verzí knihoven a jejich aktualizaci. Také v případě, že knihovna chybí, umí tyto knihovny stáhnout, například ze serverů GitHub.com či packagist.org. [76, 77]

4.6.3 PHP CodeSniffer

PHP CodeSniffer představuje dva základní skripty v jazyce PHP. Jeden z nich umožňuje vytvořit tokeny z PHP, JavaScript a CSS souboru a následně provést kontrolu kódu dle předem definovaných pravidel, standardů. Druhý umožňuje automatickou korekci zdrojových souborů dle definovaných pravidel či standardů. Uživatel může používat již existující standardy kódování, nebo si definovat vlastní, což je velká výhoda. Tato knihovna jako výchozí používá PEAR standard kódování. Tento standard můžeme jednoduše rozšířit například pomocí přidání dalších pravidel standardu Slevomat. Tam si můžeme nastavit spoustu pravidel, které bychom chtěli změnit oproti výchozímu nastavení. Tato pravidla jsou definovaná pomocí XML souboru, který se typicky jmenuje ruleset.xml a je umístěn do kořene projektu. [78, 79] Slevomat dělí tato pravidla do třech kategorií: funkční, čisticí (spojené s mrtvým, nepoužívajícím se kódem) a formátující (spojené se vzhledem kódu). [80]

4.6.4 PHPStan

PHPStan je nástroj pro statickou analýzu zdrojového kódu v jazyku PHP s otevřeným zdrojovým kódem. Tento nástroj umožňuje odhalení mnoha různých chyb bez samotného spuštění aplikace. Mohou to být například problémy spojené s typováním proměnných, mrtvým kódem, logickými chybami apod. Má několik různých úrovní přesností statické analýzy (čím vyšší je číslo, tím přísnější bude analýza). Tato nastavení se obvykle definují pomocí konfiguračního souboru ve formátu NEON. Mohou se také zadávat pomocí parametrů příkazové řádky. Tento nástroj má také PHPStan PRO verzí, která je placená a nabízí dodatečné funkce. [81]

4.6.5 Nette

Nette je framework s otevřeným zdrojovým kódem pro tvorbu webových aplikaci v jazyku PHP. Je tvořen množstvím samostatně použitelných komponent a využívá událostmi řízené programování. Nette je zaměřen na výkon, bezpečnost a znovupoužitelnost implementovaných komponent. Autorem tohoto frameworku je David Grudl, ale v současné době se o jeho vývoj stará organizace Nette Foundation. Je to český framework, proto během několika let kolem tohoto frameworku vznikla velká komunita, která pro něj vytváří různá rozšíření a knihovny [1, 82]

4.6.6 Symfony

Symfony je framework pro tvorbu webových aplikací a služeb v jazyku PHP s otevřeným zdrojovým kódem. Je kvalitně objektově navržen a založen na návrhovém vzoru MVC. Výhodou Symfony je, že je tento framework tvořen sadou samostatných znovupoužitelných komponent, které spolu tvoří celek. To je velmi užitečné, protože umožňuje použití pouze potřebných funkcí v konkrétním projektu. Navíc můžeme tyto komponenty díky jejich znovupoužitelnosti často nalézt i v jiných frameworkách či knihovnách. Z tohoto důvodu se stal tento framework velmi populárním ve světě a na jeho bázi fungují jak malé, tak i velké projekty. [83–85]

4.6.7 Docker

Docker je nástroj s otevřeným zdrojovým kódem, který zajišťuje izolaci aplikací pomocí kontejnerové virtualizace. Umožňuje jejich vytvoření, automatizované nasazení, přenositelnost a nezá-

vislost. Na rozdíl od obyčejné virtualizace sdílí kontejnery v dockeru procesy mezi sebou, čímž se zvyšuje výkon. Může být použit jak v rámci cloudu, tak i na obyčejném serveru či lokálním počítači. Je vhodný pro vývojové prostředí, protože se jednoduše konfiguruje a nevyžaduje dodatečnou konfiguraci od vývojáře, který ho potřebuje jen použít. Správce musí jen vytvořit obraz kontejneru, který může dát do nějakého úložiště obrazů, z něhož si každý bude schopen tento obraz stáhnout a vytvořit potřebný kontejner. [86, 87]

Docker compose je nástroj určený pro vytvoření a provoz vícekontejnerového prostředí. To může být nakonfigurováno pomocí obyčejného souboru ve formátu YAML. Do něho popíšeme všechny kontejnery, které budeme potřebovat pro naše prostředí. Tento nástroj představuje nadstavbu nad dockerem a používá se pro zjednodušení procesu spuštění složitě konfigurovatelného prostředí. [88]

4.6.8 Traefik

Traefik je nástroj s otevřeným zdrojovým kódem používající se pro směrování požadavků, vyvážení zátěže, proxy, nebo všechno dohromady. Traefik je nativně kompatibilní se všemi hlavními clusterovými technologiemi, jako jsou Kubernetes, Docker, Docker Swarm, AWS a další. Také umí dynamicky měnit svoji konfiguraci, není ho tedy potřeba restartovat po každé změně. Může obalit všechny vnitřní služby a zabezpečit HTTPS spojení k nim. [89, 90]

Traefik se používá k řešení problémů souvisejících s provozem a správou velkých aplikací představujících velké množství služeb. Spojení mezi těmito službami je důležitou součástí chování aplikace. Traefik nabízí vývojářům lepší přehled a spolehlivost, protože jim poskytuje vrstvu abstrakce pro řízení této komunikace. [90]

4.6.9 Nginx

Nginx je jeden z nejpoužívanějších webových serverů s otevřeným zdrojovým kódem. Je možné ho použít k různým účelům, například může sloužit jako reverzní proxy server, vyrovnávač zátěže nebo mezipaměť HTTP. Je zaměřený především na výkon a na nízké nároky na paměť. Pracuje s protokoly HTTP, HTTPS, SMTP, POP3, IMAP a SSL. Jeho autorem je Igor Sysoev a první oficiální verze vyšla v roce 2004. V současné době se jeho vývojem zabývá společnost Nginx, Inc. [91, 92]

4.6.10 Nginx Unit

Nginx Unit je webový aplikační server s otevřeným zdrojovým kódem mající tři základní funkce: je to webový server pro statická data, aplikační server umožnující spouštět kód v různých programovacích jazycích a reverzní proxy, který dovoluje směrovat příchozí požadavky dle nastavené konfigurace. Umí dynamicky spravovat svoji konfiguraci bez nutnosti zastavení všech běžících služeb pomocí API rozhraní. [93, 94]

4.6.11 PostgreSQL

PostgreSQL je pokročilý objektově relační databázový systém podnikové třídy s otevřeným zdrojovým kódem. Je vysoce stabilní a je vyvíjen a podporován více než 35 let. Na jeho vývoji se podílí velká komunita vývojářů a firem. Tento databázový stroj je navržen pro spoustu různých použití od jednotlivých strojů po velké datové sklady a webové služby s velkým množstvím aktivních uživatelů. PostgreSQL podporuje velkou část standardu SQL a nabízí mnoho moderních funkcí a rozšíření, které jsou užitečné během vývoje aplikací. [95, 96]

4.6.12 Redis

Redis představuje klíč-hodnotové úložiště s otevřeným zdrojovým kódem, které se používá jako datové úložiště v paměti pro ukládání dat, mezipaměť, zprostředkovatel zpráv nebo streamovací stroj. Všechna data tento stroj ukládá do paměti, což má velmi dobrý dopad na výkon, protože přístup k paměti je vždy rychlejší než přístup k disku. Výsledkem je velmi rychlý výkon s průměrnými operacemi čtení a zápisu, které trvají méně než milisekundu. Proto podporují mnohem více operací za vteřinu na rozdíl od datových úložišť ukládajících data na disk. Umožňuje provádění atomických operací a nabízí velké množství datových struktur, které můžeme použít pro uložení dat, například řetězce, seznamy, hashe, bitové mapy, streamy a další. Umožňuje vytváření repliky tohoto úložiště a jednoduché škálovaní. [97, 98]

Kapitola 5
Závěr

Mezi hlavní cíle této práce patřilo prozkoumání problémů spojených s vývojem a údržbou DBS portálu a také navržení nového řešení, které by mohlo nalezené problémy odstranit. Tato analýza měla za úkol zjistit příčinu problému, který s každým rokem zhoršuje celkovou udržitelnost systému a znemožňuje jeho snadný rozvoj. Na základě analýzy jsem zjistil, že současná architektura DBS portálu není vhodná pro tento způsob vývoje. Proto jsem navrhl a vytvořil novou architekturu DBS portálu. Ta má být odolnější vůči časté změně vývojářů a vývoji v malých týmech bez dobré znalosti technologií. Dalším cílem této práce bylo vytvoření dobré dokumentace projektu, která by umožňovala snadno se zorientovat na projektu a začít s vývojem mikroslužeb. Tyto cíle byly splněny.

Na začátku jsem provedl podrobnou analýzu vývoje DBS portálu v rámci SP týmů, na základě které jsem zjistil hlavní nevýhody současného řešení, jako je například velikost projektu, velká závislost komponent mezi sebou, zastaralé technologie apod. Velká závislost snižovala výhody frameworku Nette, který nabízí vytvoření samostatně použitelných komponent. Také velká provázanost tohoto systému způsobovala problémy s jeho pochopením. To mělo velmi negativní dopad na SP týmy, protože musely věnovat hodně času pochopení systému. Další problém byl spojen s tím, že lidé připojující se k projektu a nemající žádné zkušenosti s technologiemi se v něm hůře orientovali. Nebyla tedy možnost studenty správně zaškolit, aby si vyzkoušeli alespoň základní funkce frameworku a jazyka PHP. Bylo velmi složité existující testovací projekt instalovat. Nabízel totiž vyzkoušení jen některých funkcí, které ovšem nemohly plně připravit studenty na práci s tak velkým systémem.

Ve své práci jsem na základě provedené analýzy zjednodušil základní architekturu DBS portálu, modernizoval systém pomocí novějších verzí knihoven a jazyka PHP a vytvořil podrobnou dokumentaci systému. V dokumentaci jsem popsal veškeré postupy, konvence a definoval slovník objektů a procesů v rámci domény DBS portálu. Také jsem v dokumentaci definoval postup pro migraci dat do nového systému. Pro vývoj jsem vytvořil nové vývojové prostředí na základě docker kontejnerů, které budou studenti schopni instalovat během několika minut. Proces zprovoznění tohoto vývojového prostředí a tvorby nové mikroslužby jsem detailně popsal v dokumentaci projektu.

Dále jsem se rozhodl implementovat testovací projekt pro zaškolení studentů v rámci SP týmů. Jelikož se během analýzy objevil problém s neznalostí technologií, bylo potřeba zlepšit znalosti studentů před samotnou prací na projektu. Také jsem zjednodušil proces vytvoření vývojového prostředí pro tento testovací projekt pomocí dockeru. S použitím projektu se studenti budou schopni naučit základní funkce frameworku Symfony, jazyka PHP, vyzkoušet si pracovat s dockerem a REST API.

Následně jsem se v této práci zabýval návrhem a implementací mikroslužby odpovídající za autorizaci uživatelů a také knihovnou DBS Utils bundle, která obsahuje veškeré společné

funkce ze všech mikroslužeb. Na základě protokolu OAuth 2.0 jsem navrhl a implementoval způsob autorizace pomocí JWT tokenu, který obsahuje data o uživateli. Tento způsob autorizace umožňuje jednotné přihlášení a představuje obdobu OpenId Connect protokolu. Aby se v každé mikroslužbě neduplikoval kód, vytvořil jsem knihovnu obsahující všechny společné funkce. Jedna z důležitých funkcí, které jsou implementované v rámci této knihovny, je přihlášení.

Nakonec jsem se zabýval řízením vývoje mikroslužeb, kde jsem pomáhal a validoval řešení studentů, kteří se zabývali vývojem DBS portálu v rámci SP týmů. Díky této spolupráci jsem získal hodně zkušeností spojených s řízením vývojových týmů a naučil se lépe jednat s lidmi.

Příloha A

Obrázky

Verze 11/12: https://dbs.fit.cvut.cz/redmine/versions/172

Plánovaný rozsah práce:

- Komponenta na stránkování #5548 - Kuba Kuchejda
- Pročištění všude řazení, stránkování, filtrování #5493, #5581 - Kuba Kuchejda
- Příprava mockAPI, axiosAPI a dat pro templaty #5447 - Lukáš Fiala
- Propojení tvorby testové šablony a seznamu #5496 - Ondra Hampejs
- Přidání toastify k odstraňování #5580 - Ondra Hampejs
- Mixin na opakující se fetche #5441 - Martin Dvořák
- Připravení podoby parametrů, které budeme používat při filtraci, stránkování, řazení #5425 - Vojta Moravec
- Dodělání normalizace u assignments #5598 - Vojta Moravec
- Opravy dokumentace na základě diskuze s Andriim #5593 - Kuba Čermák
- Krásné zápisy z interních schůzek viz. wiki - Kuba Kuchejda
- Krásné zápisy ze schůzek s Jirkou viz. wiki - Lukáš Fiala

Vyřešeno:

```
Komponenta na stránkování #5548 - Kuba Kuchejda
Příprava mockAPI, axiosAPI a dat pro templaty #5447 - Lukáš Fiala
Přidání toastify k odstraňování #5580 - Ondra Hampejs
Dodělání normalizace u assignments #5598 - Vojta Moravec
Krásné zápisy z interních schůzek viz. wiki - Kuba Kuchejda
Krásné zápisy ze schůzek s Jirkou viz. wiki - Lukáš Fiala
```

Nestihlo se:

```
Pročištění všude řazení, stránkování, filtrování #5493, #5581 - Kuba Kuchejda
Propojení tvorby testové šablony a seznamu #5496 - Ondra Hampejs
Mixin na opakující se fetche #5441 - Martin Dvořák
Připravení podoby parametrů, které budeme používat při filtraci, stránkování, řazení #5425 - Vojta Moravec
Opravy dokumentace na základě diskuze s Andriim #5593 - Kuba Čermák
```

Plán na další sprint: Co budeme dělat dále?

- Viz. informace o verzi 13

Obrázek A.1 Ukázka reportu o stavu projektu

Entita	Zdrojový systém			Cílový systém				Metadata		Vývoj		
	Databáze	Tabulka	Sloupec	Datový typ	Databáze	Tabulka	Sloupec	Datový typ	Byznys název	Byznys definice	Implementační nástroj	Odkaz na zdrojové soubo	Poznámka pro vývoj
TestTemplate	dbs_db	test_template	test_template_id	integer	test_templates_db	test_template	test_template_id	bigint	Identifikátor testové šablony	Identifikuje testovou šablonu v systému (Nemigrovat)			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_template	name	varchar(50)	test_templates_db	test_template	name	varchar(50)	Název testové šablony				
		test_template	totalPoint	integer	test_templates_db	test_template	max_points	integer	Počet bodů	Maximální počet bodů, který student může získat z tohoto testu			
		test_template	length	integer	test_templates_db	test_template	test_duration	integer	Trvalost studentovho testu	Čas, který je odveden na vyplnění testu studentem			
		test_template	total_length	integer	tests_db	test_term	term_duration	integer	Trvalost testovho terminu	Čas, který je celkově odveden pro termin			
		test_template	type	integer	tests_db	test_term	type	varchar(50)	Typ studentovho testu	Např. zkouška, test v semestru, demo test			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_template	user_id	integer	test_templates_db	test_template	creator_id	bigint	Autor šablony				
		test_template	active	boolean	test_templates_db	test_template	is_archivated	boolean	Archivovaná šablona	Testová šablona, kterou učitel archivoval. Nezobrazuje se v seznamu šablon.			
		test_template	valid	boolean	test_templates_db	test_template	is_valid	boolean	Validní šablona	Testová šablona je validní, pokud všechny otázky jsou validní a součet bodů v otázkách je rovný počtu bodů nastavenonu v šabloně.			
TestVariant		test_template	totalPoint	integer	tests_db	test_variant	max_points	integer	Počet bodů	Maximální počet bodů, který student může získat z tohoto testu			Kopie pro tabulku varianta testu
		test_template	total_length	integer	tests_db	test_variant	term_duration	integer	Trvalost testového terminu	Čas, který je celkově odveden pro termin			
TestQuestion		test_question_to_test	test_question_id	integer	test_templates_db	static_question	question_id	bigint	Identifikátor testové otázky	Identifikuje testovou otázku v systému			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_question_to_test	test_template_id	integer	test_templates_db	static_question	test_template_id	bigint	Identifikátor testové šablony	Identifikuje testovou šablonu v systému			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_question_to_test	evaluator_user_id	integer	test_templates_db	static_question	evaluator_id	bigint	Učitel zodpovědný za hodnocení	Tento vyučující je zodpovědný za hodnocení této testové otázky			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_question_to_test	answer_order	integer	test_templates_db	static_question	index	integer	Pořadí otázek v rámci šablony	Definuje pořadí otázek, které se budou objevovat v testu.			
		test_question_to_test	point	integer	test_templates_db	static_question	max_points	integer	Počet bodů	Maximální počet bodů, které student může získat za odpověď na tuto otázku.			
TestTerm		test_created_test	course_id	integer	tests_db	test_term	course_id	bigint	Identifikátor kurzu				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_created_test	exam_id	integer	tests_db	test_term	exam_id	bigint	Identifikátor zkoušky				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_created_test	progtestevn_only	boolean	tests_db	test_term	progtest_image_require	boolean	Progtest image mode	Povolení psaní testu pouze z počítačů, kde běží progtest image			
		test_created_test	start_time	timestamp	tests_db	test_term	start	timestamp	Datum začátku psaní testů.	Od této doby studenti jsou schopni spouštět svoje vlástní testy			
		test_created_test	test_template_id	integer	tests_db	test_variant	test_template_id	bigint	Identifikátor testové šablony				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_created_test	test_created_test_id	integer	tests_db	test_variant	test_variant_id	bigint	Identifikátor testové varianty	(Nemigrovat)			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_created_test	total_length	integer	tests_db	test_term	term_duration	integer	Trvalost testového terminu	Čas, který je celkově odveden pro termin			Duplicitní
		test_created_test	user_id	integer	tests_db	test_term	creator_id	bigint	Autor terminu				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_created_test	status	varchar(10)	tests_db	test_term	status	varchar(50)	Stav terminu	Vytvořen, Připraven, Spuštěn, Uzavřen			
StudentTest		test_student_test	test_student_test_id	integer	tests_db	student_test	student_test_id	bigint	Identifikátor testu studenta				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_student_test	user_id	integer	tests_db	student_test	assigned_studen_id	bigint	Přiřazený uživatel	Je to student, který píše tento test			Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_student_test	test_room_id	integer	tests_db	student_test	room_id	bigint	Identifikátor místnosti				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_student_test	test_created_test_id	integer	tests_db	student_test	test_variant_id	bigint	Identifikátor varianty testu				Nelze použít ze staré db, je potřeba vyhledat mezi našimi identifikátory.
		test_student_test	teachers_comment	text	tests_db	student_test	teachers_comment	text	Komentář vyučujícího k testu				
		test_student_test	status	varchar(10)	tests_db	student_test	status	varchar(50)	Stav testu	Vytvořen, připraven, spuštěn, odevzdán, ohodnocen			
		test_student_test	start_time	timestamp	tests_db	student_test	started_at	timestamp	Čas začátku testu				
		test_student_test	end_time	timestamp	tests_db	student_test	ended_at	timestamp	Čas konce testu				
		test_student_test	additional_time	integer	tests_db	student_test	additional_minutes	integer	Dodatečný čas k testu	Vyučující se může rozhodnout přidat studentovi dodatečný čas na vypracování testu			
		test_student_test	oral_exam	boolean	tests_db	student_test	(neexistuje)						
		test_student_test	oral_exam_point	real	tests_db	student_test	oral_exam_points	integer	Body za ustní zkoušku				

Obrázek A.2 Ukázka datové mapy pro testové šablony a testy

Setup

1. Zkopírujte si docker-compose.yaml do kořene projektu.
2. Bude potřeba si natáhout git sub moduly, např. RAT.

   ▸ **Show instructions**
3. Přihlasit se do docker regustry: `docker login gitlab.fit.cvut.cz:5000`.
4. Spustite systémy pomocí `docker-compose up -d`.
5. Připojení k backendu: `docker-compose exec <název kontejneru> bash`
6. Spuštění přikazů na serveru `sudo -HEu web <příkaz>`, kde web je uživatel pod kterým se to spouští.
7. Instalace knihoven: `sudo -HEu web composer install`
8. Provedení migrace: `sudo -HEu web php bin/console do:mi:mi`
9. Kontrola kód stylu: `sudo -HEu web php vendor/bin/phpcs --standard=./ruleset.xml`
10. Syntax analýza pomocí phpstanu `sudo -HEu web php vendor/bin/phpstan analyse -l 9 src`
11. Spuštění unit testů: `sudo -HEu web php vendor/bin/phpunit`

Obrázek A.3 Ukázka postupu pro vytvoření vývojového prostředí

Vytvoření nové mikroslužby

1. Přejděte na větev base_microservice.
2. Zkopirujte si složku BaseMicroservice někam mimo projekt a přejmenujte ji na název nové mikroslužby. (Není vhodné mergovat, protože pro různé služby mohou vzniknout konflikty)
3. Přejděte na svojí vyvojovou větev a zkopirujte tuto složku zpatky do projektu. (Do kořenového adresáře.)
4. Nakonfigurujte `docker-compose.yml` file. Lze se inspirovat jinými kontejnery v tomto souboru. Co je potřeba nastavit:
   1. Název kontejneru.
   2. Mapování složek (volumes).
   3. Pojmenovaní hostu.
   4. Nástavit labels pro traefik. Navíc je potřeba změnit základní prefix pro path pro všechny endpointy mikroslužby. Tj. ve souboru routes.yaml změnit konfigurace kontrolerů. Dle tohoto prefixu pak bude směrovat pořadavky traefik. Např.

```
controllers:
resource: ../src/Controller/
type: annotation
prefix: /<Váš PATH>
```

5. Nakonfigurovat databázi, pokud ještě neexistuje:
   1. Upravit soubory ve složce `./docker/postgres/pg_scripts`
   2. Nastavit v .env proměnnu `DATABASE_URL` a jiné potřebné například pro fungování autorizaci, Více viz Auth mikroslužba.
6. Nástávení nginxu je v souboru `./docker/symfony/symfony.json` pokud by bylo potřeba něco měnit.
7. Dockerfile pro současný image je ve složce `./docker`. Potřebné obrazy jsou zblděné a uložené do gitlab registry.
8. Pro fungování automatického ověření kod stylu a statické analýzy kódu je potřeba přidat path do proměnny SOURCE_PATHS v souboru .gitlab-ci.yml. Proto aby to fungovalo kontejner musí mít nástaven label cz.cvut.fit.dbs.microservice.
9. Dále můžete postupovat dle bodu 5) v sekcí Setup.

Obrázek A.4 Ukázka postupu pro vytvoření nové mikroslužby

NewDBS Microservices

Od roku 2022 začíná vývoj backendu nového dbs portálu, který bude rozdělen na několik microslužeb.

- Popis microslužeb Požádavky k budoucím microslužbam.
- Slovník pojmů a procesů z domény
- Hinty k vyvoji

- Jmenná konvence v databázích Jak jménovat tabulky.
- Jmenná konvence pro REST API Jak jménovat atributy, entity, requesty, responsy.
- GIT Jmenná konvence Jak pracovat s gitem.

- Semestr B211 Popis práce předchoziho sp teamu.
- Refactoring Testového modulu DBS portálu stárá verze, ale hodně časti se dá přepoužit.

- Migrace dat do mikroslužeb

Dokumentace jednotlivých mikroslužeb:

- Authorizační mikroslužba
- Connections
- Kos imports and system configurations
- Semestral work configurations
- ...

Repozitáře:

- Současný project: https://gitlab.fit.cvut.cz/dbs/newDBS
- Nový project: https://gitlab.fit.cvut.cz/dbs/dbs-microservices
- Repo s bakalářkou o testové časti: https://gitlab.fit.cvut.cz/dbs/newDBS/merge_requests/250
- Repo s dokumentaci API testové časti: https://gitlab.fit.cvut.cz/dbs/newDBS/merge_requests/510

Bundles: ¶

- Filtry: https://gitlab.fit.cvut.cz/dbs/dbs-filters-bundle
- Utils: https://gitlab.fit.cvut.cz/dbs/dbs-microservices-utils-bundle

Wiki sp teamů vyvíjejících microslužby:

- SP-2022 Backend https://dbs.fit.cvut.cz/redmine/projects/sp-dbs-2022-backend/wiki
- SP-2022 Mix https://dbs.fit.cvut.cz/redmine/projects/sp-dbs-2022-mix/wiki

Technologie:

- **PHP8**
    1. Nové věci v php:
        - https://stitcher.io/blog/new-in-php-8
        - https://stitcher.io/blog/new-in-php-81
- **Symfony 6** dokumentace: https://symfony.com/doc/current/index.html
- **PostgreSQL 14** : https://www.postgresql.org/docs/current/index.html
- **Docker** , docker compose dokumentace: https://docs.docker.com/compose/
- **Dokumentace nginx unit** : https://unit.nginx.org/
  Image je postaven na základě nginx unit image. Detalně složka docker
- **Traefik** : https://doc.traefik.io/traefik/
- **Nginx** dokumentace: https://docs.nginx.com/

› Soubory (2)

Obrázek A.5 Ukázka dokumentace projektu v systému Redmine

Příloha B

Ukázky kódu

```
{
    "listeners": {
        "*:9000": {
            "pass": "routes/symfony",
            "client_ip": {
                "header": "X-Forwarded-For",
                "recursive": false,
                "source":
                [
                    "10.0.0.0/8",
                    "172.16.0.0/12"
                ]
            }
        }
    },

    "routes": {
        "symfony":
        [
            {
                "match": {
                    "uri":
                    [
                        "*.php",
                        "*.php/*"
                    ]
                },

                "action": {
                    "pass": "applications/symfony/direct"
                }
            },
            {
                "action": {
                    "share": "/var/www/public/public$uri",
                    "fallback": {
```

```
                        "pass": "applications/symfony/index"
                    }
                }
            }
        ]
    },


    "applications": {
        "symfony": {
            "type": "php",
            "user": "web",
            "group": "web",
            "targets": {
                "direct": {
                    "root": "/var/www/public/public/"
                },

                "index": {
                    "root": "/var/www/public/public/",
                    "script": "index.php"
                }
            }
        }
    }
}

```

Výpis kódu B.1 Nastavení Nginx serveru pro Symfony aplikací

```
version: '3.5'

services:
  auth:
    image: "gitlab.fit.cvut.cz:5000/dbs/dbs-microservices/nginx-unit-base:php8.1-1.0.2"
    hostname: "auth"
    environment:
      XDEBUG_MODE: "develop,debug"
    depends_on:
      - postgres-db
      - redis
    volumes:
      - ./Auth:/var/www/public
    networks:
      - internal
    extra_hosts: # just for dev
      - host.docker.internal:host-gateway
    labels:
      cz.cvut.fit.dbs.microservice: "Auth"
      traefik.enable: true
      traefik.http.routers.auth.rule: "PathPrefix(`/auth`)"
      traefik.http.routers.auth.entrypoints: "web"
      traefik.http.services.auth.loadbalancer.server.port: "9000"

  connections:
    image: "gitlab.fit.cvut.cz:5000/dbs/dbs-microservices/nginx-unit-connections:php8.1-1.0.1"
    hostname: "connections"
    environment:
      XDEBUG_MODE: "develop,debug"
```

```
30      depends_on:
31        - postgres-db
32        - redis
33        - auth
34        - configurations
35        - rat
36        - pgparser
37        - mssqlparser
38      volumes:
39        - ./Connections:/var/www/public
40      networks:
41        - internal
42      extra_hosts:   # just for dev
43        - host.docker.internal:host-gateway
44      labels:
45        cz.cvut.fit.dbs.microservice: "Connections"
46        traefik.enable: true
47        traefik.http.routers.connections.rule: "PathPrefix(`/connections`)"
48        traefik.http.routers.connections.entrypoints: "web"
49        traefik.http.services.connections.loadbalancer.server.port: "9000"
50
51    configurations:
52      image: "gitlab.fit.cvut.cz:5000/dbs/dbs-microservices/nginx-unit-base:php8.1-1.0.2"
53      hostname: "configurations"
54      environment:
55        XDEBUG_MODE: "develop,debug"
56      depends_on:
57        - postgres-db
58        - redis
59        - auth
60      volumes:
61        - ./Configurations:/var/www/public
62      networks:
63        - internal
64      extra_hosts: # just for dev
65        - host.docker.internal:host-gateway
66      labels:
67        cz.cvut.fit.dbs.microservice: "Configurations"
68        traefik.enable: true
69        traefik.http.routers.configurations.rule: "PathPrefix(`/configurations`)"
70        traefik.http.routers.configurations.entrypoints: "web"
71        traefik.http.services.configurations.loadbalancer.server.port: "9000"
72
73    # Database
74    postgres-db:
75      image: postgres:${POSTGRES_VERSION:-14}-alpine
76      restart: on-failure
77      volumes:
78        - ./docker/postgres/pg_scripts/docker_postgres_init.sql:/docker-entrypoint-initdb.d/
          docker_postgres_init.sql
79        - db_data:/var/lib/postgresql/data
80      environment:
81        POSTGRES_USER: root
82        POSTGRES_PASSWORD: root
83        POSTGRES_DB: root
84      ports:
85        - "5432:5432"
86      networks:
87        - internal
88
89    # Redis
90    redis:
91      image: "redis:6-alpine"
92      hostname: "redis"
93      networks:
94        - internal
```

```
    ports:
      - "6379:6379"

  rat:
    build: ./RAT
    hostname: "rat"
    networks:
      - internal

  pgparser:
    build: ./pgparser
    hostname: "pgparser"
    networks:
      - internal

  mssqlparser:
    build: ./mssqlparser
    hostname: "mssqlparser"
    networks:
      - internal

  traefik:
    image: "traefik:v2.8"
    container_name: "traefik"
    ports:
      - "80:80"
      - "443:443"
      - "8080:8080"  # (Optional) Expose Dashboard Don't do this in production!
    volumes:
      - ./docker/traefik:/etc/traefik
      - /var/run/docker.sock:/var/run/docker.sock
      - ./logs/traefik:/var/log/traefik/
    networks:
      - internal

networks:
  internal:

volumes:
  db_data:

```

Výpis kódu B.2 Příklad souboru docker-compose.yaml

```
    <?xml version="1.0"?>
<ruleset name="DBS-Microservices">
    <description>DBS-Microservices coding standards.</description>
    <config name="installed_paths" value="../../slevomat/coding-standard"/><!-- relative path from
      PHPCS source location -->
    <config name="php_version" value="80000"/> <!-- Set target version, for 7.4 set to 70400 etc. -->
    <!-- Check folders -->
    <file>./src</file>
    <arg name="colors"/> <!-- Output with colors -->
    <arg value="p"/> <!-- Display progress in report -->
    <arg value="s"/> <!-- Display sniff codes in report -->

    <!-- Include PSR-2 standard with some exceptions -->
    <rule ref="PSR2">
        <!-- exclude line length -->
        <exclude name="Generic.Files.LineLength"/>
        <!-- exclude elseif keyword -->
        <exclude name="PSR2.ControlStructures.ElseIfDeclaration"/>
        <!-- exclude abstract/final keyword before visibility declaration -->
        <exclude name="PSR2.Methods.MethodDeclaration"/>
        <!-- exclude opening parenthesis of a multi-line function call must be the last content on the line --
      >
```

```
21      <exclude name="PSR2.Methods.FunctionCallSignature.ContentAfterOpenBracket"/>
22      <exclude name="Squiz.Functions.MultiLineFunctionDeclaration.NewlineBeforeOpenBrace"/>
23      <exclude name="PSR2.Methods.FunctionCallSignature.MultipleArguments"/>
24  </rule>
25
26  <!-- Class must have declare(strict_types = 1) -->
27  <rule ref="SlevomatCodingStandard.TypeHints.DeclareStrictTypes">
28      <properties>
29          <property name="declareOnFirstLine" value="true" />
30          <property name="linesCountAfterDeclare" value="1"/>
31          <property name="spacesCountAroundEqualsSign" value="0"/>
32      </properties>
33  </rule>
34
35  <!-- ============================== Imports
        ================================== -->
36  <!-- Use statements must be sorted alphabetically -->
37  <rule ref="SlevomatCodingStandard.Namespaces.AlphabeticallySortedUses"/>
38  <!-- Detect unused use statements -->
39  <rule ref="SlevomatCodingStandard.Namespaces.UnusedUses">
40      <properties>
41          <property name="searchAnnotations" value="true"/>
42          <property name="ignoredAnnotationNames" type="array" value="@example"/>
43      </properties>
44  </rule>
45  <!-- Use statement cannot start with backslash -->
46  <rule ref="SlevomatCodingStandard.Namespaces.UseDoesNotStartWithBackslash"/>
47  <!-- Do not allow use from same namespace -->
48  <rule ref="SlevomatCodingStandard.Namespaces.UseFromSameNamespace"/>
49  <!-- Do not allow to use group imports -->
50  <rule ref="SlevomatCodingStandard.Namespaces.DisallowGroupUse"/>
51
52  <!-- ============================== Class structure
        ================================== -->
53  <rule ref="SlevomatCodingStandard.Classes.ClassStructure">
54      <properties>
55          <property name="groups" type="array">
56              <element value="uses"/>
57
58              <!-- Public constants are first but you don't care about the order of protected or private
        constants -->
59              <element value="public constants"/>
60              <element value="constants"/>
61
62              <!-- You don't care about the order among the properties. The same can be done with "
        properties" shortcut -->
63              <element value="public static properties, protected static properties, private static properties"/>
64              <element value="public properties, protected properties, private properties"/>
65
66              <!-- Constructor is first, then all public methods, then protected/private methods and magic
        methods are last -->
67              <element value="constructor"/>
68              <element value="static methods"/>
69              <element value="abstract methods"/>
70              <element value="all public methods"/>
71              <element value="methods"/>
72              <element value="magic methods"/>
73          </property>
74      </properties>
75  </rule>
76
77  <rule ref="SlevomatCodingStandard.Arrays.TrailingArrayComma"/>
78
79  <rule ref="SlevomatCodingStandard.TypeHints.ParameterTypeHint"/>
80  <rule ref="SlevomatCodingStandard.TypeHints.PropertyTypeHint"/>
81  <rule ref="SlevomatCodingStandard.TypeHints.ReturnTypeHint"/>
```

```
82    <!-- Use short type hints (integer -> int)-->
83    <rule ref="SlevomatCodingStandard.TypeHints.LongTypeHints"/>
84    <!-- Check spacing at parameter type hint -->
85    <rule ref="SlevomatCodingStandard.TypeHints.ParameterTypeHintSpacing"/>
86    <!-- Detect type hint without ? and with null default value -->
87    <rule ref="SlevomatCodingStandard.TypeHints.NullableTypeForNullDefaultValue"/>
88    <!-- Check spacing at return type hint -->
89    <rule ref="SlevomatCodingStandard.TypeHints.ReturnTypeHintSpacing"/>
90    <!-- Functions must have return type hint -->
91    <rule ref="SlevomatCodingStandard.ControlStructures.RequireNullCoalesceOperator"/>
92    <rule ref="SlevomatCodingStandard.PHP.TypeCast"/>
93    <rule ref="SlevomatCodingStandard.TypeHints.ParameterTypeHint.
        MissingTraversableTypeHintSpecification">
94        <severity>0</severity>
95    </rule>
96    <rule ref="SlevomatCodingStandard.TypeHints.ReturnTypeHint.MissingTraversableTypeHintSpecification
        ">
97        <severity>0</severity>
98    </rule>
99
100   <!-- Use short array syntax -->
101   <rule ref="Generic.Arrays.DisallowLongArraySyntax"/>
102   <!-- Detect empty if/elseif/else statements -->
103   <rule ref="Generic.CodeAnalysis.EmptyStatement"/>
104   <!-- Detects incrementer jumbling in for loops. -->
105   <rule ref="Generic.CodeAnalysis.JumbledIncrementer"/>
106   <!-- Detects for-loops that use a function call in the test expression. -->
107   <rule ref="Generic.CodeAnalysis.ForLoopWithTestFunctionCall"/>
108   <!-- Detect unconditional if statements -->
109   <rule ref="Generic.CodeAnalysis.UnconditionalIfStatement"/>
110   <!-- Detect final methods in final class -->
111   <rule ref="Generic.CodeAnalysis.UnnecessaryFinalModifier"/>
112   <!-- Do not use deprecated functions -->
113   <rule ref="Generic.PHP.DeprecatedFunctions"/>
114   <!-- Do not use forbidden functions (sizeof, delete) -->
115   <rule ref="Generic.PHP.ForbiddenFunctions"/>
116   <!-- Detect unnecessary string concat-->
117   <rule ref="Generic.Strings.UnnecessaryStringConcat"/>
118   <!-- Class should have same name as file -->
119   <rule ref="Squiz.Classes.ClassFileName"/>
120   <!-- Do not allow to use @deprecated annotations without description -->
121   <rule ref="SlevomatCodingStandard.Commenting.DeprecatedAnnotationDeclaration"/>
122
123   <!-- Always use constant visibility -->
124   <rule ref="SlevomatCodingStandard.Classes.ClassConstantVisibility"/>
125   <!-- Do not allow assignment in condition -->
126   <rule ref="SlevomatCodingStandard.ControlStructures.AssignmentInCondition"/>
127   <!-- Checks language construct used with parentheses -->
128   <rule ref="SlevomatCodingStandard.ControlStructures.LanguageConstructWithParentheses"/>
129   <!-- Do not allow yoda comparison example: (false == $resource) -->
130   <rule ref="SlevomatCodingStandard.ControlStructures.DisallowYodaComparison"/>
131   <!-- Detect dead catch -->
132   <rule ref="SlevomatCodingStandard.Exceptions.DeadCatch"/>
133   <!-- Use \Throwable instead of \Exception -->
134   <rule ref="SlevomatCodingStandard.Exceptions.ReferenceThrowableOnly"/>
135
136   <!-- ================================ Comments
        ======================================== -->
137   <!-- Do not allow comments for constructor and php storm generated comments -->
138   <rule ref="SlevomatCodingStandard.Commenting.ForbiddenComments">
139       <properties>
140           <property name="forbiddenCommentPatterns" type="array" value="~Constructor\.~"/>
141           <property name="forbiddenCommentPatterns" type="array" value="~Created by PhpStorm\.~"/>
142       </properties>
143   </rule>
144   <!-- Reports empty comments. -->
```

```
    <rule ref="SlevomatCodingStandard.Commenting.EmptyComment"/>
    <!-- Reports invalid inline phpDocs with @var. -->
    <rule ref="SlevomatCodingStandard.Commenting.EmptyComment"/>
    <!-- Requires property comments with single-line content to be written as one-liners. -->
    <rule ref="SlevomatCodingStandard.Commenting.RequireOneLinePropertyDocComment"/>
</ruleset>
```

Výpis kódu B.3 Příklad souboru ruleset.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/
    services/services-1.0.xsd">
    <services>
        <!-- ============================ -->
        <!-- =======|  Crypto  |======= -->
        <!-- ============================ -->
        <!-- SimpleCrypto -->
        <service id="dbs_utils.crypto.simple_crypto"
                class="Dbs\UtilsBundle\Crypto\SimpleCrypto"
                public="true"
        >
            <argument key="$secret">%env(APP_SECRET)%</argument>
        </service>
        <service id="Dbs\UtilsBundle\Crypto\CryptoInterface"
                alias="dbs_utils.crypto.simple_crypto"
                public="true"
        />

        <!-- ============================ -->
        <!-- =====|  Form types  |===== -->
        <!-- ============================ -->
        <!-- CarbonTransformer -->
        <service id="dbs_utils.form.fields.transformers.carbon_transformer"
                class="Dbs\UtilsBundle\Form\Fields\Transformers\CarbonTransformer"
                public="true"
        />
        <service id="Dbs\UtilsBundle\Form\Fields\Transformers\CarbonTransformer"
                alias="dbs_utils.form.fields.transformers.carbon_transformer"
                public="true"
        />
        <!-- CarbonType -->
        <service id="Dbs\UtilsBundle\Form\Fields\CarbonType"
                class="Dbs\UtilsBundle\Form\Fields\CarbonType"
                public="true"
        >
            <argument type="service" id="dbs_utils.form.fields.transformers.carbon_transformer"/>
        </service>
        <!-- CypherTextTransformer -->
        <service id="dbs_utils.form.fields.transformers.cypher_text_transformer"
                class="Dbs\UtilsBundle\Form\Fields\Transformers\CypherTextTransformer"
                public="true"
        >
            <argument type="service" id="dbs_utils.crypto.simple_crypto"/>
        </service>
        <service id="Dbs\UtilsBundle\Form\Fields\Transformers\CypherTextTransformer"
                alias="dbs_utils.form.fields.transformers.cypher_text_transformer"
                public="true"
        />
        <!-- CypherTextType -->
        <service id="Dbs\UtilsBundle\Form\Fields\CypherTextType"
                class="Dbs\UtilsBundle\Form\Fields\CypherTextType"
                public="true"
        >
            <argument type="service" id="dbs_utils.form.fields.transformers.cypher_text_transformer"/>
```

```
57        </service>
58
59
60        <!-- ============================ -->
61        <!-- ========| Redis |========= -->
62        <!-- ============================ -->
63        <!-- RedisFormatter -->
64        <service id="Dbs\UtilsBundle\Redis\Formatter\RedisFormatter"
65                 class="Dbs\UtilsBundle\Redis\Formatter\RedisFormatter"
66                 public="true"
67        />
68        <!-- RedisStorage -->
69        <service id="Dbs\UtilsBundle\Redis\AuthRedisStorage"
70                 class="Dbs\UtilsBundle\Redis\AuthRedisStorage"
71                 public="true"
72        >
73            <argument type="service" id="Symfony\Contracts\Cache\CacheInterface"/>
74            <argument type="service" id="Dbs\UtilsBundle\Redis\Formatter\RedisFormatter"/>
75            <argument key="$expTime">%env(REDIS_EXPIRATION_TIME)%</argument>
76        </service>
77
78
79        <!-- ============================ -->
80        <!-- ======| Security |====== -->
81        <!-- ============================ -->
82        <!-- DBS Authenticator -->
83        <service id="Dbs\UtilsBundle\Security\Authenticator\DbsAuthenticator"
84                 class="Dbs\UtilsBundle\Security\Authenticator\DbsAuthenticator"
85                 public="true"
86        />
87        <!-- User provider -->
88        <service id="Dbs\UtilsBundle\Security\Provider\UserProvider"
89                 class="Dbs\UtilsBundle\Security\Provider\UserProvider"
90                 public="true"
91        >
92            <argument type="service" id="Dbs\UtilsBundle\Security\TokenManager\TokenManager"/>
93        </service>
94        <!-- Token manager -->
95        <service id="Dbs\UtilsBundle\Security\TokenManager\TokenManager"
96                 class="Dbs\UtilsBundle\Security\TokenManager\TokenManager"
97                 public="true"
98        >
99            <argument key="$key">%env(TOKEN_SECRET_KEY)%</argument>
100           <argument key="$alg">%env(TOKEN_ALGORITHM)%</argument>
101       </service>
102   </services>
103 </container>
104
```

Výpis kódu B.4 Nastavení DBS Utils knihovny

```
1 zend_extension=xdebug
2
3 [xdebug]
4 xdebug.mode=develop,debug
5 xdebug.client_host=172.18.0.1
6 xdebug.start_with_request=yes
7
```

Výpis kódu B.5 Nastavení XDebug pro vývojové prostředí

Bibliografie

1. PLYSKACH, Andrii. *Refaktoring testové části backendu portálu dbs.fit.cvut.cz*. Praha, 2020. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
2. ERBEN, Marek. *Automatické generování a oprava otázek na normalizaci databáze pro předmět BI-DBS*. Praha, 2017. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
3. MACHALA, Filip. *Automatická oprava zjednodušeného relačního zápisu*. Praha, 2017. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
4. GLAZAR, Filip. *Systém pro podporu BI-DBS - semestrální práce*. Praha, 2016. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
5. PEJŠA, Petr. *Systém pro podporu testování v BI-DBS*. Praha, 2016. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
6. KOVÁŘ, Pavel. *Automatizované testování webového portálu dbs.fit.cvut.cz*. Praha, 2017. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
7. FEDOR, Bc. Tomáš. *ER diagrams web component II*. Praha, 2017. Dipl. pr. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
8. KUBIŠ, Martin. *Překladač z relační algebry do SQL*. Praha, 2016. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
9. MALEC, Oldřich. *Řízení projektu a infrastruktury portálu pro podporu výuky předmětu BI-DBS*. Praha, 2017. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
10. SÝKORA, Jan. *Podpora automatizované kontroly semestrální práce z předmětu Databázové systémy*. Praha, 2015. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
11. HANZL, Martin. *dbs.fit.cvut.cz - Refaktoring testů I*. Praha, 2021. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.

12. JORDÁN, Pavel. *dbs.fit.cvut.cz - Refaktoring testů II.* Praha, 2022. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
13. CHALUPA, Tomáš. *Frontend manuální korektury otázek v portálu dbs.fit.cvut.cz.* Praha, 2022. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří HUNKA.
14. Representational State Transfer. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-11-09]. Dostupné z: `https://cs.wikipedia.org/wiki/Representational_State_Transfer`.
15. Mikroslužby. In: [online]. https://cs.wikipedia.org, [b.r.] [cit. 2022-12-15]. Dostupné z: `https://cs.wikipedia.org/wiki/Mikroslu%C5%BEby`.
16. What are Microservices? In: [online]. https://aws.amazon.com, [b.r.] [cit. 2022-12-15]. Dostupné z: `https://aws.amazon.com/microservices/`.
17. Slack (software). In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-11-09]. Dostupné z: `https://cs.wikipedia.org/wiki/Slack_(software)`.
18. Redmine. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-11-09]. Dostupné z: `https://cs.wikipedia.org/wiki/Redmine`.
19. What is Domain Driven Design (DDD)? In: *https://stackoverflow.com* [online]. [B.r.] [cit. 2022-10-09]. Dostupné z: `https://stackoverflow.com/questions/1222392/what-is-domain-driven-design-ddd/1222488#1222488`.
20. What is Domain Driven Design and why is it so important? In: *https://www.hexacta.com* [online]. [B.r.] [cit. 2022-10-09]. Dostupné z: `https://www.hexacta.com/what-is-domain-driven-design-and-why-is-it-so-important/`.
21. Developing the ubiquitous language. In: *https://thedomaindrivendesign.io* [online]. [B.r.] [cit. 2022-10-19]. Dostupné z: `https://thedomaindrivendesign.io/developing-the-ubiquitous-language/`.
22. What Is Legacy Code: 8 Tips For Working Effectively With Legacy Code. In: *https://www.perforce.com* [online]. [B.r.] [cit. 2022-10-09]. Dostupné z: `https://www.perforce.com/blog/qac/8-tips-working-legacy-code`.
23. ERIC EVANS, Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2004. ISBN 0321125215.
24. Creational Design Patterns. In: *https://refactoring.guru* [online]. [B.r.] [cit. 2022-10-15]. Dostupné z: `https://refactoring.guru/design-patterns/creational-patterns`.
25. Pattern: Decompose by business capability. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-29]. Dostupné z: `https://microservices.io/patterns/decomposition/decompose-by-business-capability.html`.
26. Pattern: Decompose by subdomain. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-29]. Dostupné z: `https://microservices.io/patterns/decomposition/decompose-by-subdomain.html`.
27. Self-contained service. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-29]. Dostupné z: `https://microservices.io/patterns/decomposition/self-contained-service.html`.
28. Service per team. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-29]. Dostupné z: `https://microservices.io/patterns/decomposition/service-per-team.html`.
29. Pattern: Database per service. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-29]. Dostupné z: `https://microservices.io/patterns/data/database-per-service.html`.

30. Database-per-service pattern. In: *https://docs.aws.amazon.com* [online]. [B.r.] [cit. 2022-10-30]. Dostupné z: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/database-per-service.html.
31. Pattern: Shared database. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-30]. Dostupné z: https://microservices.io/patterns/data/shared-database.html.
32. Model distribuovaných transakcí Saga. In: *https://learn.microsoft.com* [online]. [B.r.] [cit. 2022-10-30]. Dostupné z: https://learn.microsoft.com/cs-cz/azure/architecture/reference-architectures/saga/saga.
33. Pattern: Saga. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-30]. Dostupné z: https://microservices.io/patterns/data/saga.html.
34. API Composition Pattern in Microservices. In: *https://www.linkedin.com* [online]. [B.r.] [cit. 2022-10-31]. Dostupné z: https://www.linkedin.com/pulse/api-composition-pattern-microservices-arpit-bhayani.
35. Pattern: API Composition. In: *https://microservices.io* [online]. [B.r.] [cit. 2022-10-31]. Dostupné z: https://microservices.io/patterns/data/api-composition.html.
36. Zámek (informatika). In: *https://medium.com/design-microservices-architecture-with-patterns* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c.
37. Co znamená zkratka CRUD?Co znamená zkratka CRUD? In: *https://it-slovnik.cz* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://it-slovnik.cz/pojem/crud.
38. Zámek (informatika). In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://cs.wikipedia.org/wiki/Z%C3%A1mek_(informatika).
39. Message broker. In: *https://en.wikipedia.org* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://en.wikipedia.org/wiki/Message_broker.
40. Reverzní proxy - Reverse proxy. In: *https://wikijii.com* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://wikijii.com/wiki/Reverse_proxy.
41. Why use an API gateway? In: *https://www.redhat.com* [online]. [B.r.] [cit. 2022-11-02]. Dostupné z: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do.
42. Backend for frontend (BFF) pattern— why do you need to know it? In: *https://medium.com* [online]. [B.r.] [cit. 2022-11-03]. Dostupné z: https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0.
43. Autentizace. In: *https://www.strafelda.cz* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://www.strafelda.cz/autentizace.
44. 5 fundamental strategies for REST API authentication. In: *https://www.techtarget.com* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://www.techtarget.com/searchapparchitecture/tip/5-fundamental-strategies-for-REST-API-authentication.
45. Basic access authentication. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://cs.wikipedia.org/wiki/Basic_access_authentication.
46. What is a Digital Signature? In: *https://www.techtarget.com* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://www.techtarget.com/searchsecurity/definition/digital-signature.
47. Introduction to JSON Web Tokens. In: *https://jwt.io* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://jwt.io/introduction.
48. JSON Web Token. In: *https://en.wikipedia.org* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: https://en.wikipedia.org/wiki/JSON_Web_Token.

49. How to Sign and Validate JSON Web Tokens – JWT Tutorial. In: *https://www.freecodecamp.org* [online]. [B.r.] [cit. 2022-11-04]. Dostupné z: `https://www.freecodecamp.org/news/how-to-sign-and-validate-json-web-tokens/`.
50. The OAuth 2.0 Authorization Framework. In: *https://www.rfc-editor.org* [online]. [B.r.] [cit. 2022-11-12]. Dostupné z: `https://www.rfc-editor.org/rfc/rfc6749`.
51. OAuth 2 Simplified. In: *https://aaronparecki.com/* [online]. [B.r.] [cit. 2022-11-12]. Dostupné z: `https://aaronparecki.com/oauth-2-simplified/`.
52. OpenID Connect (OIDC). In: *https://www.pingidentity.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: `https://www.pingidentity.com/en/resources/identity-fundamentals/authentication-authorization-standards/openid-connect.html`.
53. OpenID Connect explained. In: *https://connect2id.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: `https://connect2id.com/learn/openid-connect`.
54. Shibboleth. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-05]. Dostupné z: `https://cs.wikipedia.org/wiki/Shibboleth`.
55. Shibboleth - identifikujte se jen jednou. In: *https://www.lupa.cz* [online]. [B.r.] [cit. 2022-12-05]. Dostupné z: `https://www.lupa.cz/clanky/shibboleth/`.
56. Shibboleth SP 3.0 - Install&Configure. In: *https://wiki.cvut.cz* [online]. [B.r.] [cit. 2022-12-05]. Dostupné z: `https://wiki.cvut.cz/pages/viewpage.action?pageId=40077405`.
57. SAML vs OAuth. In: *https://auth0.com* [online]. [B.r.] [cit. 2022-12-05]. Dostupné z: `https://auth0.com/intro-to-iam/saml-vs-oauth`.
58. Autorizační server. In: *https://rozvoj.fit.cvut.cz* [online]. [B.r.] [cit. 2022-11-07]. Dostupné z: `https://rozvoj.fit.cvut.cz/Main/oauth2`.
59. Architektura mikroslužeb je zárukou efektivního vývoje. Na co si dát pozor? In: *https://www.master.cz* [online]. [B.r.] [cit. 2022-10-26]. Dostupné z: `https://www.master.cz/blog/architektura-mikrosluzeb-efektivni-vyvoj-na-co-si-dat-pozor/`.
60. KOTLÁŘ, Robert; KOPECKÝ, Martin. *Prezentace: ETL nástroje* [online]. [B.r.] [cit. 2022-11-14]. Dostupné z: `https://courses.fit.cvut.cz/MI-EDW/lectures/04_prednaska.pdf`.
61. KOTLÁŘ, Robert; KOPECKÝ, Martin. *Prezentace: Podnikové datové sklady* [online]. [B.r.] [cit. 2022-11-14]. Dostupné z: `https://courses.fit.cvut.cz/MI-EDW/lectures/03_prednaska.pdf`.
62. Statická analýza PHP kodu. In: *https://www.kutac.cz* [online]. [B.r.] [cit. 2022-11-15]. Dostupné z: `https://www.kutac.cz/weby-a-vse-okolo/staticka-analyza-php-kodu`.
63. REST API Best Practices – REST Endpoint Design Examples. In: *https://www.freecodecamp.org* [online]. [B.r.] [cit. 2022-11-16]. Dostupné z: `https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/`.
64. Kontejnerizace. In: *https://studuj.digital* [online]. [B.r.] [cit. 2022-11-16]. Dostupné z: `https://studuj.digital/2020/10/13/kontejnerizace/`.
65. Docker – LABEL Instruction. In: *https://www.geeksforgeeks.org* [online]. [B.r.] [cit. 2022-11-19]. Dostupné z: `https://www.geeksforgeeks.org/docker-label-instruction/`.
66. Use bridge networks. In: *https://docs.docker.com* [online]. [B.r.] [cit. 2022-11-19]. Dostupné z: `https://docs.docker.com/network/bridge/`.
67. What is a Data Transfer Object (DTO)? In: *https://stackoverflow.com* [online]. [B.r.] [cit. 2022-11-27]. Dostupné z: `https://stackoverflow.com/questions/1051182/what-is-a-data-transfer-object-dto`.

68. ZOUBEK, Bohumír. *Prezentace: Software testing* [online]. [B.r.] [cit. 2022-12-03]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a4m33sep/prednasky/05_testing.pdf.
69. software testing. In: *https://www.techtarget.com* [online]. [B.r.] [cit. 2022-12-03]. Dostupné z: https://www.techtarget.com/whatis/definition/software-testing.
70. Typy testování software (třídění testů). In: *https://kitner.cz* [online]. [B.r.] [cit. 2022-12-03]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/.
71. Mock object. In: *https://en.wikipedia.org* [online]. [B.r.] [cit. 2022-12-03]. Dostupné z: https://en.wikipedia.org/wiki/Mock_object.
72. Test Case – Testovací případ. In: *http://testovanisoftwaru.cz* [online]. [B.r.] [cit. 2022-12-03]. Dostupné z: http://testovanisoftwaru.cz/dokumentace-v-testovani/test-case/.
73. The Bundle System. In: *https://symfony.com* [online]. [B.r.] [cit. 2022-11-19]. Dostupné z: https://symfony.com/doc/current/bundles.html.
74. PHP. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-06]. Dostupné z: https://cs.wikipedia.org/wiki/PHP.
75. Výhody a nevýhody PHP. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-06]. Dostupné z: https://www.zizka.ch/pages/programming/php/vyhody_a_nevyhody_php.html.
76. Composer. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://cs.wikipedia.org/wiki/Composer.
77. Co je to Composer? In: *https://it-slovnik.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://it-slovnik.cz/pojem/composer.
78. PHP CodeSniffer - kód který nesmrdí. In: *https://www.kutac.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://www.kutac.cz/weby-a-vse-okolo/php-codesniffer-kod-ktery-nesmrdi.
79. PHP CodeSniffer. In: *https://github.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://github.com/squizlabs/PHP%5C_CodeSniffer.
80. Slevomat Coding Standard. In: *https://github.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://github.com/slevomat/coding-standard.
81. PHPStan - PHP Static Analysis Tool. In: *https://github.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://github.com/phpstan/phpstan.
82. Nette Framework. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://cs.wikipedia.org/wiki/Nette_Framework.
83. Symfony. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://cs.wikipedia.org/wiki/Symfony.
84. Lekce 1 - Úvod do Symfony frameworku pro PHP. In: *https://www.itnetwork.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://www.itnetwork.cz/php/symfony/zaklady/uvod-do-symfony-frameworku-pro-php.
85. What is Symfony. In: *https://symfony.com/* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://symfony.com/what-is-symfony.
86. Co je to Docker a k čemu je dobrý. In: *https://vshosting.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://vshosting.cz/blog/co-je-to-docker-a-k-cemu-je-dobry.
87. Proč používat Docker. In: *https://zdrojak.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://zdrojak.cz/clanky/proc-pouzivat-docker/.

88. Docker Compose. In: *https://docs.docker.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://docs.docker.com/compose/.

89. Traefik Proxy Documentation - Traefik. In: *https://doc.traefik.io* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://doc.traefik.io/traefik/.

90. What is Traefik & How to Learn Traefik? In: *https://www.devopsschool.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://www.devopsschool.com/blog/what-is-traefik-how-to-learn-traefik/.

91. Správa linuxového serveru: Webový server Nginx. In: *https://www.linuxexpres.cz* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://www.linuxexpres.cz/praxe/sprava-linuxoveho-serveru-webovy-server-nginx.

92. Nginx. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://cs.wikipedia.org/wiki/Nginx.

93. NGINX Unit. In: *https://wtit.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://wtit.com/nginx-unit/.

94. About. In: *https://unit.nginx.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://unit.nginx.org/#about.

95. What is PostgreSQL? In: *https://www.postgresqltutorial.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://www.postgresqltutorial.com/postgresql-getting-started/what-is-postgresql/.

96. PostgreSQL. In: *https://cs.wikipedia.org* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://cs.wikipedia.org/wiki/PostgreSQL.

97. Introduction to Redis. In: *https://redis.io* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://redis.io/docs/about/.

98. What is Redis? In: *https://aws.amazon.com* [online]. [B.r.] [cit. 2022-12-07]. Dostupné z: https://aws.amazon.com/redis/.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy nového DBS portálu
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `documentation`
  - `auth_openapi.yml` ..... dokumentace k autorizační mikroslužbě
- `text`
  - `DP_Plyskach_Andrii_2023.pdf` ..... text práce ve formátu PDF
- `images` ..... složka s obrázky a diagramy