FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání diplomové práce

Název:	Aplikace Akrmat – verze 4.0 – backend
Student:	Bc. Jan Štefánik
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 31. srpna 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Pokyny pro vypracování

Cílem práce je technologický a funkční re-design backend aplikace Akrmat využívané na FIT k přípravě a generování akreditačních materiálů.
1. Seznamte se se stávající aplikací Akrmat, její funkcionalitou a technologií, ve které je implementovaná.
2. Společně s vedoucím práce shromážděte, analyzujte a formalizujte požadavky na novou verzi aplikace, zahrňte též požadavky uvedené níže.
3. Diskutujte moderní, perspektivní technologie a architektury určené pro realizaci backend aplikací, zejména datové úložiště a Api, které bude backend poskytovat.
4. Zohledněte požadavky z bodu 2 a na základě rešerše z bodu 3 zvolte vhodnou architekturu a technologie pro implementaci backend nové verze aplikace Akrmat.
5. Backend implementujte, řádně zdokumentujte a otestujte.
6. Proveďte transformaci dat z předchozí verze systému.

Požadavky:
- pro autentizaci a autorizaci využijte technologie OAuth, aplikaci navažte na role poskytované školním UserMap Api,
- pro doplnění publikací garantů předmětů využijte možnosti importu ze školních systémů UserMap a V3S a případně zvažte další API poskytující publikační citace
- pro aktualizaci nových verzí předmětů použijte KOS Api,
- definujte strukturu úložiště dat a využijte vhodnou metodu validace vkládaných dat (například JSON Schema),
- vyžijte školní DevOps prostředí (gitlab),
- generovaná dokumentace Api.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 31. srpna 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace Akrmat – verze 4.0 – backend

Bc. Jan Štefánik

Katedra softwarového inzenýrství

Vedoucí práce: Ing. Michal Valenta, Ph.D.

27. dubna 2023

Poděkování

Chtěl bych poděkovat panu Ing. Michalu Valentovi, Ph.D. jakožto svému vedoucímu práce za podporu a vstřícný přístup během realizace této práce. Dále bych také chtěl poděkovat panu Ing. Tomáši Chvostovi za poskytnutí konzultace ohledně migrace dat.

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.

Prague dne 27. dubna 2023 . . . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Štefánik, Jan. *Aplikace Akrmat – verze 4.0 – backend*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato diplomová práce se zabývá přepsáním backendu předchozí verze fakultního systému Akrmat s využitím modernějších technologií. V rámci práce jsou sesbírány požadavky na novou verzi, je analyzováno předchozí řešení včetně identifikace jeho nedostatků. Na základě těchto informací je proveden návrh a implementace. Výsledkem je nový funkční backend systému Akrmat, který je připraven na propojení s novou verzí frontendu tohoto systému, která má být do budoucna vytvořena.

Klíčová slova Akrmat, Kotlin, OAuth, akreditace, REST, MongoDB

Abstract

The goal of this diploma thesis is to re-write the backend of a faculty system called Akrmat with the use of more modern and up-to-date technologies. Requirements are gathered and previous version is analyzed including the identification of its flaws. New system is designed and implemented based on this information. The result is a new functional backend of Akrmat, which is ready to be connected to the new frontend of this system, which is planned to be developed.

Keywords Akrmat, Kotlin, OAuth, akreditace, REST, MongoDB

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tato diplomová práce se zabývá přepsáním backendu systému Akrmat, který byl v předešlých letech zpracován v rámci bakalářských a diplomových prací. Systém Akrmat slouží k správě informací týkajících se studijních akreditací. Součástmi těchto akreditací jsou informace o studijních plánech, specializacích, předmětech, učitelích a jejich publikacích. Na základě těchto informací je systém schopný generovat akreditační spisy v TEXovém formátu.

Práce je zaměřena především na analýzu předchozích verzí backendu tohoto systému, návrhu na jejich vylepšení a samotnou implementaci při zachování klíčových funkcionalit systému. Důraz je kladen zejména na udržitelnou a rozšiřitelnou architekturu systému a její implementaci.

Požadavek na přepsání systému vznikl z důvodu použití starších či již vůbec nepodporovaných technologií. Poslední verze používá databázi Sedna, jejíž vývoj byl ukončen v roce 2011. Používá také jazyk PHP, který, přestože je stále velmi používaný, již málokdy bývá volbou pro nové projekty.

Text je členěn do několika na sebe navazujících kapitol. Po definování cíle práce jsou také jasně definovány funkční a nefunkční požadavky na nový systém. Následně je provedena analýza předchozích verzí systému, shrnutí a poukázání na nedostatky, které je možno zlepšit. Poté je představen návrh nového systému, ve kterém je popsána volba databáze, technologií, architektury a integrace s dalšími systémy ČVUT FIT. Na základě této analýzy je provedena a popsána samotná imlementace ve zvolených technologiích. Na závěr je popsáno testování této implementace.

Kapitola 1

Cíl práce

Cílem práce je přepsat starou verzi backendu Akrmatu v modernějších technologiích, rozšířit či vylepšit její funkcionality a přemigrovat data ze staré databáze Sedna.

Poslední používaná verze je systém vytvořený v diplomových pracích Ing. Michala Kabelky [2] a Ing. Luboše Růžičky [1]. Byl zde ještě pokus o rozšíření tohoto systému v diplomové práci Ing. Jakuba Štefana [3], ale ta nikdy nebyla nasazená a ve zbytku práce se budu odkazovat primárně k poslední používané verzi.

Pod zachováním funkcionalit je myšlena schopnost spravovat data o akreditačních spisech a na jejich základě generovat tyto spisy ve formátu TEX.

Rozšíření funkcionalit je v podstatě zmíněno v samotném zadání, ale pro shrnutí se jedná o následující:

- Úprava datového modelu tak, aby odpovídal stávajícím požadavkům.
- Využití FIT OAuth systému a Usermap rolí pro integraci existujících uživatelů.
- Využití KOS API pro import relevantních entit týkajících se studijních plánů, předmětů a specializací.

Kapitola 2

Teorie

Tato kapitola se zabývá teoretickým seznámením s technologiemi a principy, které byly v práci použity, či byly pro použití zvažovány.

2.1 Representational State Transfer

Representational State Transfer, neboli REST, je způsob architektury API, který představil Roy Fielding ve své dizertační práci [6]. Dnes je to standardizovaný a populární způsob, jakým navrhovat API. API, které tento způsob používají, se nazývají RESTful. REST je nezávislý na protokolu, ale prakticky je téměř pokaždé implementován s využitím HTTP.

Základní principy REST architektury [4, 5] jsou:

- *Client-server architektura* – Klient i server jsou na sobě nezávislí. Je definováno uniformní rozhraní, které říká, jakým způsobem bude komunikace probíhat. Klient i server tak mohou být vyvíjeni nezávisle na sobě a v případě potřeby je možné provést jejich výměnu bez toho, aby to ovlivnilo chování druhé strany.

- *Vrstvená architektura* – Klient neví o tom, zda komunikuje přímo se serverem, nebo zda komunikuje skrze prostředníky jako proxy nebo load balancer.

- *Statelessness* – Server je bezestavový, neudržuje si žádné informace o klientech. Veškeré informace potřebné k úspěšnému provedení operace musí být obsaženo v požadavku od klienta.

- *Uniform interface* – API má definováno uniformní rozhraní, které poskytuje konečnou množinu operací pro manipulaci se zdroji. Každý zdroj má

svůj unikátní identifikátor a může mít více způsobů svých reprezentací (například různé datové formáty).

2.1.1 HTTP a REST

Jak již bylo zmíněno, protokol HTTP je nejčastější volbou pro implementaci RESTful API. Automaticky splňuje požadavky na client-server architekturu a bezestavovost, jelikož samotný protokol je takto navržený.

Pro vytvoření uniformního rozhraní používá HTTP metody, které odpovídají CRUD (create, read, update, delete) operacím [7].

- *GET* – Operace pro čtení dat.
- *POST* – Operace pro vytváření nových dat.
- *PUT* – Operace pro aktualizaci dat (celé entity). V případě neexistující entity vytváří novou.
- *DELETE* – Operace pro mazání dat.

HTTP metody mají definované, zda jsou bezpečné [8] a idempotentní [9]. Bezpečná metoda je taková, která pouze získává data, nemění je. Idempotentní metoda je taková, která má vždy stejný výsledek i při opakovaném provedení.

Metoda	Bezpečná	Idempotentní
GET	✓	✓
HEAD	✓	✓
OPTIONS	✓	✓
TRACE	✓	✓
PUT	✗	✓
DELETE	✗	✓
POST	✗	✗
PATCH	✗	✗
CONNECT	✗	✗

Tabulka 2.1: Bezpečnost a idempotentnost HTTP metod

Pro vytváření uniformního rozhraní jsou entity definovány pomocí podstatných jmen v množném čísle v kombinaci s těmito HTTP metodami. Příklad správného rozhraní může být *GET /bookings*, *DELETE /bookings/{id}*. Naopak porušení těchto konvencí je třeba *GET /getBookings*, *POST /fetchBooking*.

2.2 SQL a NoSQL databáze

SQL a NoSQL označuje 2 hlavní přístupy k návrhu databází. SQL je tradiční relační přístup, NoSQL je v podstatě všechno ostatní. NoSQL databáze se snaží řešit problémy, kde relační přístup není dostatečný a na základě specifického problému přináší jiný přístup s vlastními výhodami a nevýhodami.

2.2.1 SQL databáze

Relační databáze jsou založeny na principu relací. Data ukládají do tabulek, které se skládají ze sloupců a řádek. Sloupce definují typ informace a hodnoty, kterých mohou nabývat. Řádky definují jednotlivé záznamy, které jsou definovány unikátním klíčem. Mezi tabulkami mohou existovat vazby, na základě těchto klíčů [10]. Relační databáze jsou transakční, což znamená, že podporují *ACID* vlastnosti [11].

- *Atomicity* – Všechny operace v transakci se buď úspěšně provedou, nebo se neprovedou vůbec.
- *Consistency* – Všechny operace převádí databázi z jednoho validního stavu do dalšího validního stavu.
- *Isolation* – Každá transakce probíhá izolovaně, navzájem se neovlivňují.
- *Durability* – Výsledek úspěšně dokončené transakce je dlouhodobě uložený.

Relační databáze bývají typicky také normalizované, pokud to specifický use case nevyžaduje jinak. Normalizací se zbavujeme anomálií, které by vznikaly třeba u aktualizace dat [16].

2.2.2 NoSQL databáze

Jak bylo zmíněno, NoSQL databáze se snaží řešit různé problémy, kdy relační přístup není dostačující. Nelze však vyřešit všechny problémy zároveň. Toto vychází z CAP teorému [12, 13]. Ten definuje 3 vlastnosti a říká, že lze dosáhnout pouze 2 z nich zároveň.

- *Consistency* – Každé zápis a čtení je proveden atomicky. Po zápisu každý vidí stejná data.
- *Availability* – Na každý požadavek se vrátí (nechybová) odpověď.
- *Partition tolerance* – Systém zůstává funkční i při ztrátě zpráv mezi uzly.

Zatímco relační databáze využívají *ACID* vlastnosti, mnoho NoSQL databází implementuje vlastnosti *BASE*. *ACID* vlastnosti upřednostňují konzistenci před dostupností, *BASE* toto děla opačně [15, 14].

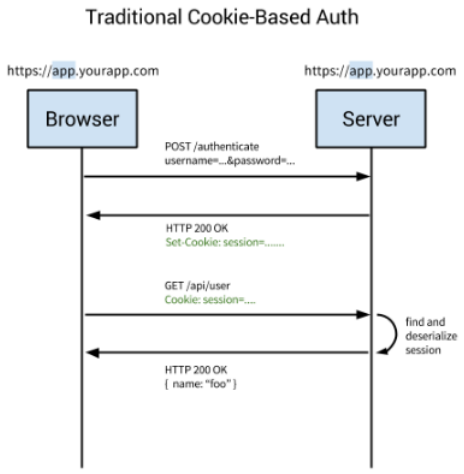
- *Basically Available* – Systém je vždy dostupný i při výpadku jeho částí.
- *Soft state* – Stav systému se může měnit bez externích požadavků.
- *Eventual consistency* – Systém se eventuálně dostane do konzistentního stavu; nemusí to však být okamžitě.

2.3 Autentifikace a autorizace

Autentifikace je proces ověření identity daného uživatele. Autorizace je proces ověření práv. Existují 2 základní principy, kterými tohoto jde docílit – pomocí session a pomocí tokenu.

2.3.1 Session-based ověření

Princip ověřování na základě session je starší, avšak stále používaný přístup. Tento přístup je stateful. V kombinaci s použitím HTTP protokolu, který je stateless, tedy vyplývá, že informace o session musí být udržována na straně serveru [17].



Obrázek 2.1: Session-based ověření [18]

Diagram na Obrázku 2.1 zobrazuje kroky, které jsou v tomto typu ověření prováděny. Klient se do systému přihlásí pomocí jména a hesla. Při úspěšném přihlášení si server informace o session uloží a klientu v odpovědi vrátí identifikátor session. Klient tento identifikátor uloží jako cookie a s každým dalším požadavkem jej posílá serveru. Aby server tuto session ověřil, musí tento identifikátor najít ve své databázi. Session může zaniknout po nějaké časově době, nebo když klient pošle požadavek na odhlášení.

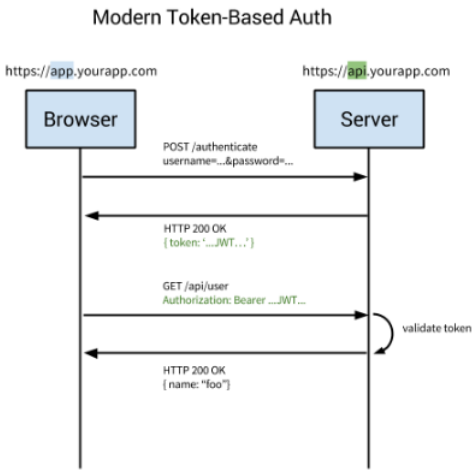
2.3.2 Token-based ověření

Ověřování na základě tokenů může být provedeno více způsoby dle zvoleného typu. Nejpopulárnějšími jsou OAuth, OpenID a JWT.

2.3.2.1 JWT

JSON Web Token, zkráceně JWT, je poměrně nový způsob, kterým jde implementovat autentifikace a autorizace. Jedná se o standard, který je popsán v RFC 7519 [20].

Tento token se skládá ze 3 částí – hlavička, payload a podpis. Hlavička obsahuje metadata, o jaký typ tokenu se jedná, jaký algoritmus byl použit pro podpis apod. Payload obsahuje deklarace, což jsou například informace o identitě nebo o rolích. Podpis je hash vytvořený z hlavičky, payloadu a tajné informace / privátního klíče (podle algoritmu). Tento podpis se používá k ověření toho, zda zpráva nebyla změněna. Pro podpis je možné využít různých algoritmů jako třeba HMAC či RSA. Výsledná struktura tokenu je *base64(header).base64(payload).base64(signature)* [21].



Obrázek 2.2: JWT ověření [18]

Diagram na Obrázku 2.2 znázorňuje kroky, které se provedou při ověřování tokenu. Po úspěšném ověření přihlašovacích údajů je klientovi vrácen JWT token, který si klient uloží. Všechny následující požadavky budou tento token přidávat v *Authorization* hlavičce. Server poté ověří, zda je JWT ve správném formátu a zda se shoduje podpis.

Výhodou oproti klasické session je, že jsou tzv. self-contained, obsahují veškeré informace potřebné k autorizaci a autentifikaci. Nemusí se volat databáze nebo autentifikační server. Jejich zprocesování je tedy rychlejší a zároveň jednodušší na implementaci [19].

2.3.2.2 OAuth 2.0

OAuth 2.0 je autorizační protokol definovaný v RFC 6749 [23]. Jedná se o delegační protokol, což znamená, že umožňuje uživateli dát oprávnění aplikacím k tomu, aby mohli přistupovat k informacím z jiných aplikací bez toho, aby jim uživatel musel sdělovat své přihlašovací údaje [22].

OAuth funguje na principu *access tokens*, které obsahují informace o autorizaci uživatele. Na rozdíl od JWT nemají standardem definovaný formát.

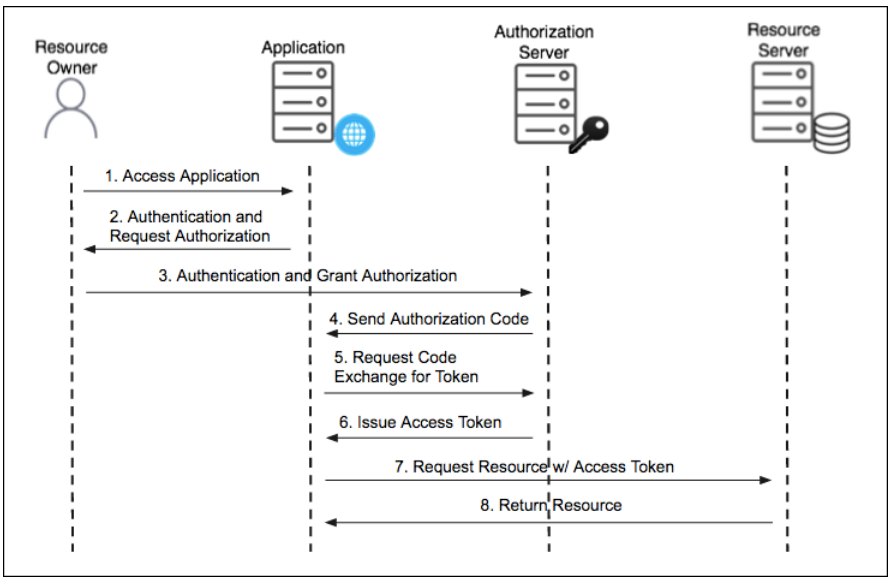
OAuth definuje několik základních rolí [24] ve svém systému:

- Resource owner - Uživatel, kterému patří nějaký zdroj či informace, ke kterým může dát přístup
- Client - Aplikace, která chce přistoupit ke zdroji uživatele
- Authorization server - Server, který ověřuje uživatele (resource owner) a na základě jeho souhlasu vydává access tokeny. Také obsahuje endpoint pro ověřování tokenů.
- Resource server - Server, který obsahuje uživatelovy resources, ke kterým chce client přistoupit.

OAuth dále definuje také koncept *scopes*. Tyto scopes definují, k jakým resources bude klient mít přístup a jaké s nimi bude moci provádět operace. Příklad scope může být například, že klient si bude moct číst data z resource serveru. Pro zápis dat může existovat separátní scope. Standard opět nedefinuje formát scopes [25].

Poslední důležitý koncept jsou *granty*. Grant je způsob, jakým může být provedena výměna za access token. OAuth grantů je celkem 7: 5 používaných, zbylé 2 již využívané nejsou. Zde budou popsány 3 nejpoužívanější - *authorization code*, *client credentials* a *refresh token*. Během těchto výměn se používá

client id, což je identifikátor klienta a *client secret*, což je tajná hodnota známá jenom klientovi a autorizačnímu serveru [26].



Obrázek 2.3: Authorization code grant [28]

Na diagramu v Obrázku 2.3 je zobrazen průběh *authorization code* grantu. Aplikace přesměruje resource ownera na autorizační server. Na straně autorizačního serveru se uživatel přihlásí pomocí svých údajů (jméno, heslo) a dá souhlas k udělení grantů. Aplikace od autorizačního serveru obdrží *authorization code*, který následně vymění za access token. S tímto access tokenem poté může přistupovat k resource serveru [27].

Refresh token grant se používá, aby uživatel nemusel znovu procházet procesem přihlašování u autorizačního serveru, když access tokenu vyprší platnost. Pokud je tento grant na autorizačním serveru implementovaný, klient obdrží v odpovědi od autorizačního serveru také refresh token. Tento refresh token poté může být vyměněn za nový access token bez nutnosti interakce uživatele [27].

Client credentials je poslední grant, který zde bude zmíněn. Tento grant nezahrnuje interakci uživatele a používá se pro komunikaci mezi aplikacemi. Výměna vyžaduje pouze client id a client secret [27].

2.4 Softwarová architektura

Softwarová architektura nemá jednoznačnou definici, ale dalo by se říci, že to je nejvyšší úroveň abstrakce, která popisuje, jak mezi sebou jednotlivé systémy či jednotlivé komponenty systému komunikují a jak jsou strukturovány.

Jedná se o důležitý krok ve vývoji softwaru, jelikož správně navržený systém umožňuje snadné a flexibilní provádění změn. Dobrá architektura také vede k principu *low coupling, high cohesion*. Tento princip říká, že jednotlivé části systému by měly být zodpovědné pouze za jednu specifickou věc (může se jednat například o generování JWT). Zároveň by mezi nimi mělo být co nejméně závislostí [29].

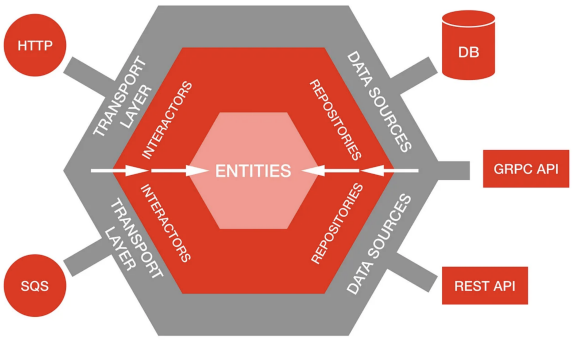
2.4.1 Hexagonální architektura

Hexagonální architektura staví na principu domény a adaptérů. Adaptéry jsou části systému, které jsou zodpovědné za interakci s externími částmi aplikace. To může být API aplikace, komunikace s databází, komunikace s API třetích stran, zpracování zpráv z Kafka apod.

Doména naopak pouze vykonává operace, které reprezentují business logiku aplikace. To často je i pouhé volání správných operací z jednotlivých adaptérů. S implementacemi adaptérů však doména nekomunikuje na přímo, nýbrž přes dodatečnou vrstvu abstrakce. Struktura projektu s touto architekturou je znázorněna na Obrázku 2.4.

V této architektuře mají doména i jednotlivé adaptéry svůj vlastní datový model, mezi kterými je prováděno mapování. Doména je tedy oddělená od implementačních detailů a neřeší věci jako například skrz jaký typ API komunikuje či odkud bere vstupní data. Tímto dobře dodržuje již zmíněný princip *low coupling, high cohesion* [30, 31].

Její nevýhodou je, že může přidávat další komplexitu způsobenou touto nepřímou komunikací. Také poměrně zvětšuje velikost kódu díky oddělení adaptérů a domény.



Obrázek 2.4: Hexagonální architektura [30]

Kapitola 3

Požadavky

Tato kapitola se zaměřuje na sběr a popis funkčních a nefunkčních požadavků na novou verzi systému. Požadavky byly definované na základě konzultací s vedoucím práce.

3.1 Funkční požadavky

- Aplikace bude spravovat 5 entit – studijní program, studijní obor, předmět, učitel a publikace.
  - V předchozí verzi byl studijní obor součástí programu. Studijní obor bude samostatná entita.
- Pro všechny entity bude vytvořeno API pro manipulaci s nimi (CRUD operace).
  - Studijní programy, studijní obory a předměty bude možné importovat z KOS API.
  - Publikace bude možné importovat z V3S systému.
  - Učitelé budou vytvářeni manuálně přes API.
  - Pro studijní program, obor, předmět a učitele bude možné vygenerovat TEX reprezentaci entity.
- Aplikace bude implementovat autentifikaci a autorizaci integrovanou s FIT OAuth a Usermap API.
  - Každý přístup k API bude vyžadovat autentifikaci uživatele s potřebnými rolemi.
  - Na základě rolí z Usermap budou jednotlivé operace zpřístupněny pouze oprávněným uživatelům.

- Aplikace bude ukládat metadata u entit. Metadata budou dostupná z API pouze admin uživatelům. Budou obsahovat následující informace:
  - autor dokumentu,
  - čas vytvoření,
  - autor poslední změny,
  - čas poslední změny,
  - informace, zda je dokument uzamčen,
  - verze dokumentu.
- Aplikace bude schopná zamykat dokumenty, čímž indikuje, že s dokumentem nikdo kromě autora nemůže interagovat s výjimkou operací pro čtení.
  - Zámek nebude omezen časem, ale administrátor bude moci zámek odebrat.

3.2 Nefunkční požadavky

- Na základě analýzy budou zvoleny vhodné technologie – programovací jazyk, případné frameworky a databáze.
- Během vývoje bude použito fakultní GitLab prostředí a Continuous Integration.
- Data budou validována na aplikační straně i na straně databáze.
- API bude obsahovat automaticky generovanou dokumentaci.
- Aplikace bude pokryta jednotkovými a integračními testy.
- Data ze starého systému budou přemigrována do nového.

Kapitola 4

Analýza

Tato kapitola se zabývá analýzou předchozích verzí Ing. Michala Kabelky [2], Ing. Luboše Růžičky [1] a Ing. Jakuba Štefana [3]. Jsou zde popsány některé části starého systému a je poukázáno na některé nedostatky. Část této kapitoly vychází z Markdown dokumentů z přiložených zdrojů v adresáři *akrmat-v4-backend/analysis*, které vznikly v průběhu konzultací před návrhem a implementací práce.

V této kapitole je odkazováno k textům těchto 3 předchozích prací a také ke zdrojovému kódu poslední nasazené verze, který mi byl zpřístupněn. Verze tohoto posledního systému neobsahuje historii změn, ale úpravy v ní prováděny byly. Nemusí tedy nutně reprezentovat stav, v jakém systém byl vytvořen na konci těchto diplomových prací, reprezentuje však současný stav implementovaného systému.

4.1 Předchozí verze Akrmatu, historie

Zde bude něco málo uvedeno k předchozím verzím systému Ing. Michala Kabelky, Ing. Luboše Růžičky a Ing. Jakuba Štefana.

4.2 Role ve starém systému

Poslední nasazená verze systému implementovala následující role zobrazené na Obrázku 4.1.

Činnost \ Role	Nepřihlášený	Čtenář	Autor	Editor	Admin
Čtení záznamů	×	✓	✓	✓	✓
Vytváření záznamů	×	×	✓	✓	✓
Editace vlastních záznamů	×	×	✓	✓	✓
Editace cizích záznamů	×	×	×	✓	✓
Generování studijního plánu	×	✓	✓	✓	✓
Generování akreditace	×	✓	✓	✓	✓
Zálohování databáze	×	×	×	✓	✓

Obrázek 4.1: Nastavení rolí v původním systému [32]

Role Editor a Admin jsou zde redundantní, jelikož plní stejný účel. Toto kritizoval i Ing. Štefan ve své práci, který navrhnul následující vytvoření rolí zobrazené na Obrázku 4.2.

Akce	Reader	Author	Admin
Čtení záznamů	Ano	Ano	Ano
Vytváření záznamů	Ne	Ano	Ano
Editace vlastních záznamů	Ne	Ano	Ano
Editace cizích záznamů	Ne	Ne	Ano
Import z VVVS API	Ne	Ano	Ano
Import z KOS API	Ne	Ne	Ano
Generování studijního plánu	Ne	Ne	Ano
Generování akreditačního spisu	Ne	Ne	Ano

Obrázek 4.2: Nastavení rolí v práci Ing. Štefana [3]

Tento návrh rolí odstraňuje redundanci mezi adminem a editorem a také upravuje oprávnění ke generování akreditačního spisu. V nastavení rolí problém nevidím a velmi podobná verze popisu rolí je popsána v Sekci 5.

4.3 API

4.3.1 Nedostatky

API předchozí verze má několik nedostatků. Prvním je, že je popisováno jako RESTful API, ale využívá PHP SESSION. Její využití není samo o sobě špatně, ale nedá se poté už mluvit o RESTful API, jelikož je stateful. To pravděpodobně vychází z části, kde Ing. Kabelka ve své práci v Sekci 2.3.1 [2] popisuje RESTful API jako takové, které dodržuje principy RESTu a je stateful a RESTless API jako takové, které dodržuje principy RESTu a je stateless. Toto není pravda. RESTful API je vždy stateless, RESTless API je pak jednoduše takové, které principy RESTu nedodržuje.

Druhým nedostatkem je, že prohazuje sémantiku POST a PUT metod. POST metoda slouží pro vytváření zdrojů, PUT slouží pro aktualizaci (může však vytvářet neexistující zdroje). Toto chování je i dokumentováno komentáři v zdrojovém kódu.

```
/*
 * zmena predmetu
 */
function post($request, $param) { ... }
```

Výpis kódu 1: Sémantika POST v předchozí verzi

Poslední věcí je používání českých názvů v API. Standardní konvencí je používání anglických jmen.

4.3.2 Chybějící funkcionality

Je zde také pár věcí, které nejsou vyloženě nedostatky nebo chyby, ale místa, jejichž funkcionality by šlo zlepšit či rozšířit. Jsou zde 2 hlavní věci, které v této verzi chybí:

- Import dat – Všechny entity jsou vytvářeny přes toto CRUD API. Systém ukládá data, která jsou dostupná z API jiných systému jako je například KOS a tento proces je možné automatizovat.
- Integrace uživatelů FIT – Systém používá své vlastní uživatele. Je zde možnost navázat uživatele na již existující v FIT OAuth systému.

4.4 Zhodnocení kvality kódu

Na základě analýzy kódu poslední nasazené verze bylo odhaleno několik značných nedostatků jak z architektonického hlediska, tak z implementačního.

Z architektonického hlediska lze kritizovat například to, že zde není žádné rozumné oddělení zodpovědností. Controller třídy rovnou vykonávají business logiku, ukládají data do databáze a tyto entity z databáze rovnou také vrací v odpovědi.

Z hlediska implementace je zde stejný problém s jmennými konvencemi jako u API. Některé části kódu jsou v češtině, některé jsou anglicky. Datový model je celý česky. Metody controllers, kvůli nedostatečné architektuře, obsahují velké množství opakující se logiky a konstanty přímo hard-coded v zdrojovém kódu. V následujícím Výpisu kódu 2 je uveden zkrácený příklad z POST metody pro předmět.

```
function post($request, $param) {
    $response = new Response($request);
    // more code here - shortened for the purposes of the example
    $document = new programDocument("$param.xml");
    try {
        $document->load();
    } catch (Exception $exc) {
        $errXML = new ERRORstatusXML("Problém při načítání programu",
                    Response::BADREQUEST, $exc);
        $response->body = $errXML->getContent();
        $response->code = $errXML->getHttpCode();
        return $response;
    }

    if ($document->getZamceno() != "" &&
        $document->getZamceno() != $_SESSION['user']['username']) {
            $exc = new Exception("program zamknul uživatel " .
                    $document->getZamceno(), 0, null);
            $errXML = new ERRORstatusXML("Problém při ukládání programu",
                    Response::FORBIDDEN, $exc);
            $response->body = $errXML->getContent();
            $response->code = $errXML->getHttpCode();
            return $response;
    }
    $document->setZmeneno(date("Y-m-d") . "T" . date("H:i:s") . "Z");
    $document->setZmenil($_SESSION['user']['username']);
    $document->setContent($request->data);
    // more code here - shortened for the purposes of the example
     try {
        $document->update();
    } catch (Exception $exc) { ... }
}
```

Výpis kódu 2: Příklad kódu starého systému

Tato zkrácená část metody ukazuje některé chyby, které byly popsané výše. V metodě z controlleru se řeší přímo přístup k databázi, kontroluje se zámek dokumentu a autentizace, mění se metadata dokumentu. Jednoduše provádí celou business logiku. Obsahuje také zmíněné hard-coded konstanty, které se v ostatních metodách tohoto i ostatních controllerů opakují. Veškerý exception handling se také řeší přímo v metodách. Lepší přístup je pomocí separátního handleru, který by chyby zpracovával.

Kapitola 5

Návrh

Následující sekce se zabývá návrhem nového systému. Stejně jako Kapitola 4 i tato sekce zčásti z vychází z Markdown dokumentů z přiložených zdrojů v adresáři *akrmat-v4-backend/analysis*.

5.1 Databáze

V následujících sekcích je popsána volba a návrh databáze pro nový systém.

5.1.1 Volba databáze

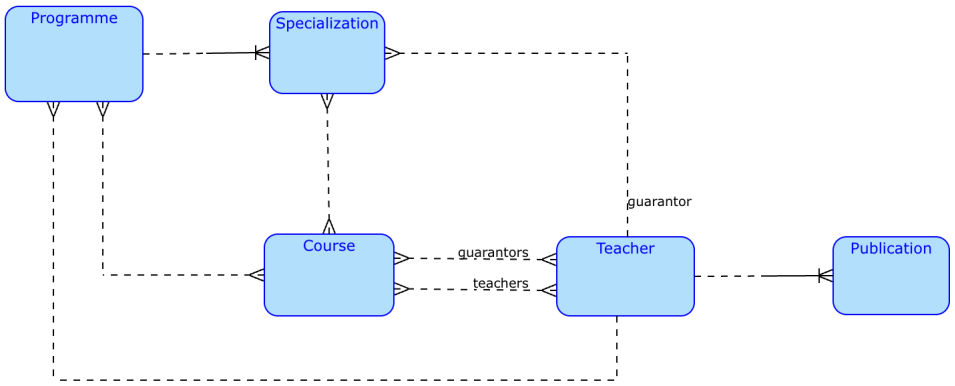
V průběhu návrhu byly 2 hlavní kandidáti, mezi kterými bylo rozhodováno, PostgreSQL a MongoDB.

První verze návrhu využívala klasický relační přístup, který nabízí PostgreSQL a to z důvodu, že chceme ukládat několik entit, které mezi sebou mají jasně dané vazby.

Na základě konzultací s vedoucím práce však vyvstaly obavy z komplikací v případě potřeby změn či aktualizací schématu. V tuto dobu již byly zvolené technologie pro implementaci, které jsou popsané v Sekci 6.1. Bylo v plánu využít Hibernate, což je ORM framework. Ten dodává rozumný způsob, jakým tyto změny provádět, ale je potřeba s ním mít zkušenosti a praxi, jinak může naopak přinést problémy. Vedoucí práce však řekl, že by radši viděl přístup, kdy se do PostgreSQL uloží data jako binární JSON (JSONB) a nebo přístup s MongoDB. Na konzultacích bylo také zmíněno, že je žádoucí využít MongoDB v praxi v některém fakultním systému. Nakonec bylo rozhodnuto pro MongoDB. Není zde ani problém využití databázových referencí Monga, které jsou pomalejší než joins v relačních databázích. Objem dat v starém systému se pohybuje pouze v řádu stovek a nepředpokládá se řádové navýšení.

5.1.2 Datový model

Jelikož byla zvolená MongoDB, v datovém modelu byl použit denormalizovaný přístup vnořených dokumentů, kde to bylo možné. U některých míst to možné nebylo a musela být použitá reference do dokumentů.



Obrázek 5.1: Datový model

Na Obrázku 5.1 je zobrazen konceptuální model entit v aplikaci. V datovém modelu jsou pomocí slabých entit znázorněné vnořené dokumenty. Ostatní relace jsou databázové reference. První vnořeným dokumentem jsou publikace učitelů, u kterých se předpokládá, že daná publikace patří pouze jednomu učiteli. V případě více autorů stejné publikace by musela být tato publikace uložena vícekrát denormalizovaným způsobem. Druhý vnořený dokument je studijní specializace. Zde by bylo možné namítnout, že se to nezbavuje problému z funkčních požadavků a není to skutečně samostatná entita. Toto bylo diskutováno s vedoucím práce a splňuje to potřeby aplikace, jelikož všechna data o specializaci (např. její předměty) jsou uložená v této entitě. V předchozí implementaci tyto data o specializaci byla umístěna do několika různých XML tagů, což je ten problém, kterého se bylo potřeba zbavit.

5.2 Role v novém systému

Jak bylo zmíněno v Kapitole 4, role v novém systému se budou podobat těm z práce Ing. Štefana. Je zde však pár změn, které vycházejí z požadavků a z konzultací.

Pro nový backend systému byly definovány role, které lze ze systémů ČVUT FIT použít. Jedna byla přepoužitá a dvě nové byly vytvořeny.

- *B-18000-SUMA-ZAMESTNANEC* – Jedná se o roli čtenáře, který si může data ze systému zobrazovat.
- *T-AKRMAT-18000-EDITOR* – Jedná se o roli editora, který si v systému může vytvářet vlastní záznamy.
- *T-AKRMAT-18000-ADMINISTRATOR* – Jedná se o roli administrátora, který má v systému největší práva.

Ve zbytku práce budou tyto role označovány pouze jako READER, EDITOR a ADMIN.

V Tabulce 5.1 jsou zobrazeny akce a nastavení rolí, které v systému figurují.

Operace	EDITOR	READER	ADMIN
Čtení záznamů	✓	✓	✓
Vytváření záznamů	✗	✓	✓
Editace vlastních záznamů	✗	✓	✓
Editace vlastních zamčených záznamů	✗	✓	✓
Editace cizích záznamů	✗	✗	✓
Editace cizích zamčených záznamů	✗	✗	✗
Import entit	✗	✓	✓
Generování akreditačního spisu	✗	✗	✓
Změna zámku svého záznamu	✗	✓	✓
Změna zámku cizího záznamu	✗	✗	✓
Vytvoření vazby mezi záznamy	✗	✓	✓

Tabulka 5.1: Role v nové verzi systému

5.3 Autentifikace a autorizace

V následujících podsekcích je popsán návrh autentifikace a autorizace pro uživatele a pro externí systémy.

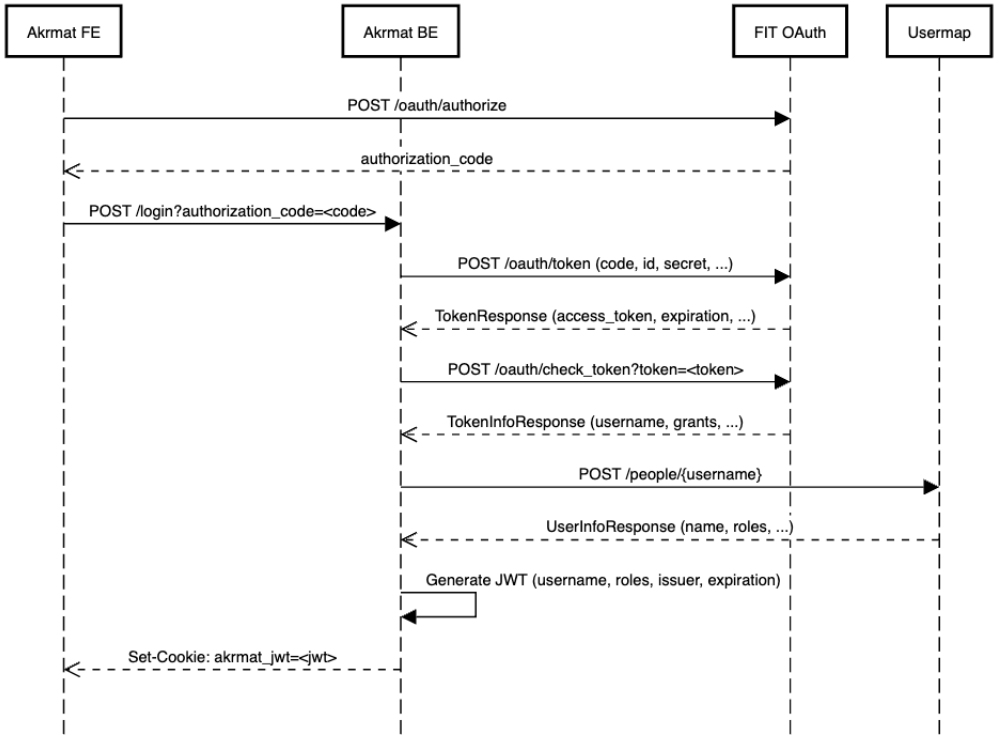
5.3.1 Autentifikace uživatelů

Je potřeba propojit integrovat existující uživatelé systémů FIT a jejich role do tohoto systému. K tomu lze využít FIT OAuth a Usermap API. Zde bude využit *authorization_code* grant.

Autentifikaci a autorizaci uživatelů bylo rozhodnuto implementovat stateless způsobem s využitím JWT. Pro prvotní přihlášení se využije OAuth serveru FITu, a i když OAuth je koncipován jako autorizační protokol, lze ho v tomto případě použít i pro autentifikační účely. Ze získaného OAuth access tokenu lze získat username uživatele a ten lze následně použít pro získání rolí z Usermap API. Z těchto informací, username a role, se následně vytvoří JWT, který se klientovi (frontend) vrátí jako cookie. Tato cookie bude mít několik vlastností:

- *HttpOnly* – Zabraňuje cookie číst client-side skriptům. To zabraňuje XSS útokům.
- *SameSite: Strict* – Zabraňuje cross-site požadavkům. To zabraňuje CSRF útokům.
- *Secure* - Přenos cookie bude povolen pouze přes bezpečné (HTTPS) spojení.

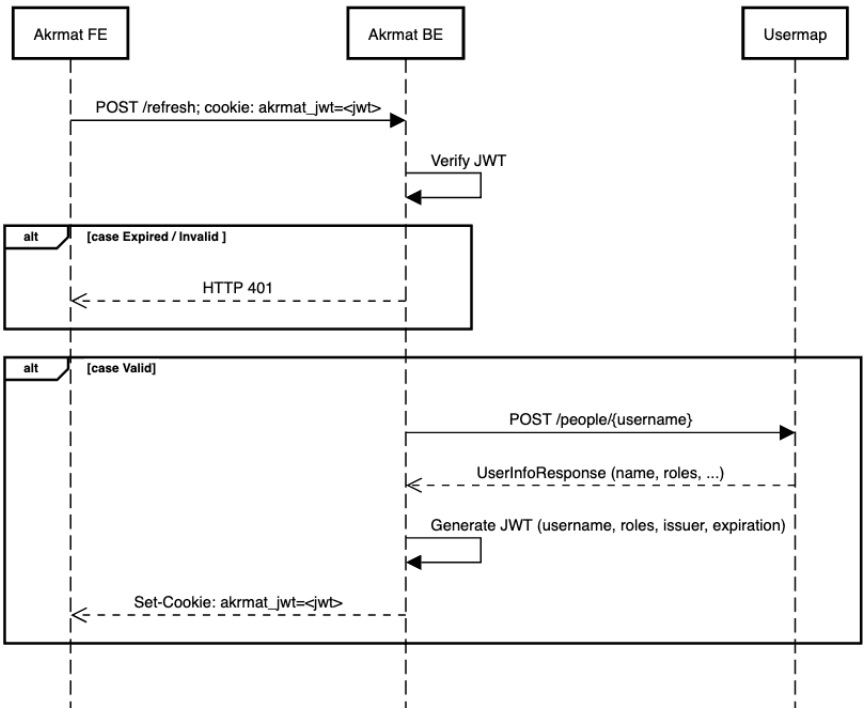
Klient tuto cookie tedy nemůže číst, ale browser jí bude automaticky posílat s každým požadavkem. Aby klient věděl o době expirace, vrátí se v těle odpovědi také čas expirace tokenu. Tento proces je zobrazen v sekvenčním diagramu na Obrázku 5.2.



Obrázek 5.2: Výměna za JWT

Na diagramu je zobrazena pouze úspěšná cesta. V případě chyby komunikace tato chyba bude samozřejmě zpropagována klientovi.

V původním návrhu byl přiložen také endpoint *POST /refresh* pro obnovení tokenu. Funkcionalita tohoto endpointu je zobrazena v diagramu na Obrázku 5.3. Po dalším zvažování však bylo rozhodnuto, že tento endpoint implementován nebude. Důvodem je, že pokud by byl implementován, ztratila by se „short-lived“ vlastnost JWT tokenů. Ta znamená, že tokeny by měly mít rozumně krátkou platnost. S tímto endpointem by je bylo možné nekonečně obnovovat. Klient tedy po vypršení JWT bude muset provést opětovné přihlášení. Platnost tokenu bude nastavena na výchozí hodnotu 4 hodiny jako kompromis mezi uživatelskou přívětivostí a bezpečností. Hodnota expirace JWT bude externě konfigurovatelná.



Obrázek 5.3: Obnovení JWT

Pro získání dat z Usermap API je potřeba mít scopes *urn:ctu:oauth:umapi.read* a *cvut:umapi:read*.

5.3.2 Import dat z externích systémů

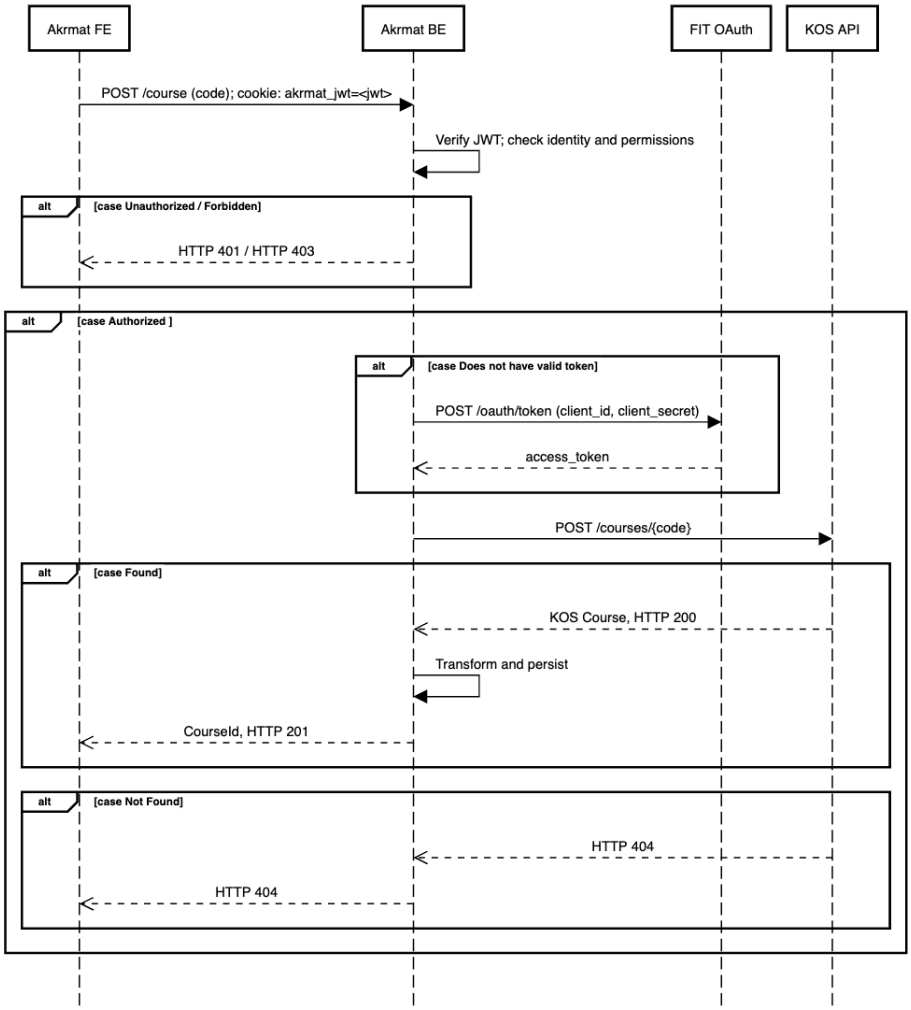
Původní požadavek byl na import dat z KOS API a z V3S API. Bohužel k V3S API nebyla dostupná ani dokumentace ani přístup, tudíž v této sekci bude popsána pouze komunikace s KOS API.

KOS API umožňuje získat 3 z 5 entit, které Akrmat spravuje. Jedná se o studijní plán, specializaci a předmět. Jsou dostupné na následujících endpointech [33]:

- GET /programmes, GET /programmes/{code}
- GET /studyPlans, GET /studyPlans/{code}
- GET /courses, GET /courses/{code}

KOS API vyžaduje autorizaci v podobě OAuth tokenu s příslušným scope *cvut:kosapi:read*. Pro tento typ komunikace bude použit grant *client creden-*

tials, jelikož se jedná o mezistrojovou komunikaci. V případě, že backend nemá uložený validní access token, musí si ho od FIT OAuth serveru vyžádat. V opačném případě může rovnou provést požadavek na KOS API. Pokud je daná entita nalezena, backend si jí transformuje do svého formátu, uloží a na frontend vrátí informaci o vytvoření. V případě, že nalezena nebyla, chybu zpropaguje na frontend. Tento proces je zobrazen na následujícím diagramu v Obrázku 5.4



Obrázek 5.4: Import z KOS API

Kapitola 6

Implementace

V této kapitole je popsána implementace systému na základě představené analýzy.

6.1 Výběr technologií

Volba programovacího jazyka a frameworků byla diskutována během pravidelných konzultací s vedoucím práce. Ze strany vedoucího zde nebyly žádné omezení či požadavky, bylo tudíž rozhodnuto použít Kotlin v kombinaci se Spring Boot. Pro spravování závislostí je využit Maven.

Důvod výběru byl i zčásti subjektivní z důvodu pohodlnosti implementace, ale jsou zde také objektivní vlastnosti, které z něj dělají dobrou volbu. Zkráceně by se Kotlin dal popsat jako „lepší Java“. Některé z těchto důvodů jsou popsány v následujících odstavcích [34].

Interoperabilita s Javou Kotlin je navržený tak, aby byl plně kompatibilní s Javou. To umožňuje v Kotlinu využívat existující a rozsáhlý JVM ekosystém. Jedná se například o různé optimalizace na úrovní byte code typu JIT compiling či využití různých existujících Java knihoven.

Statické typování Kotlin je staticky typovaný, což znamená, že ověřování typů je ověřováno během kompilace. Toto je dobrá vlastnost u projektů, které mají být dlouhodobě udržovány a má na nich pracovat více lidí. To je situace i tohoto projektu, předpokládá se, že bude vyvíjen a udržován i nadále mimo tuto práci. Tato vlastnost výrazně ulehčuje čtení a porozumění zdrojovému kódu, jelikož typy používané v programu jsou explicitně uvedeny.

Null safety Kotlin nedovoluje přiřazování *null* hodnot, pokud to není explicitně pro danou proměnnou povoleno. Má také speciální syntaxi pro *nullable*

proměnné, díky které se lze často vyhnout *if* podmínkám při práci s nimi. Jedná se o velmi dobrou vlastnost jazyka, jelikož *NullPointerException*, která vzniká při přístupu k proměnné s hodnotou *null*, je jedna z nejčastějších chyb v Javě.

```
val nullableExample: Int? = null
// compilation error
val nonNullableExample: Int = null

// example of syntax sugar to work with nullable properties
val pontentionallyNull: MyObject?
// later in the program
pontetionallyNull?.let { println(pontetionallyNullable.property) }
```

Výpis kódu 3: Null safety v Kotlinu

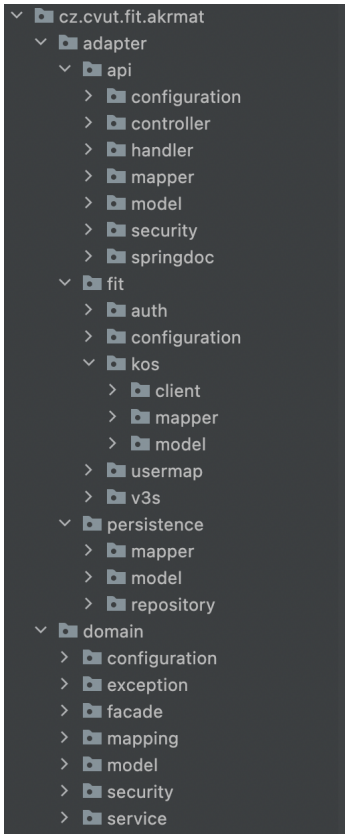
Spring Boot je jeden z nejpopulárnějších frameworků pro Javu a Kotlin. Spring byl ve svém základu vytvořen jako framework pro *dependency injection*, dnes však pod sebou obsahuje mnoho různých funkcionalit. Spring Boot usnadňuje použití Springu tím, že automaticky konfiguruje potřebné části Spring frameworku. Kromě základní dependency injection nabízené Springem bylo použito několik dalších částí.

- *Spring web* – Web server a API.
- *Spring data* – Přístup k databázi, mapování tříd do databáze a provádění dotazů.
- *Spring security* – Nastavení autentifikace a autorizace.

Maven je jeden z dvou (druhý je Gradle) hlavních populárních nástrojů pro spravování závislostí a sestavování projektů. Definuje soubor *pom.xml*, který XML formátem popisuje závislosti, kroky ke kompilaci a případné další dodatečné pluginy. Pomocí něj lze projekt kompilovat, zabalit do JAR souboru, spustit jeho testy atd.

6.2 Struktura projektu

Struktura projektu vychází z principů hexagonální architektury, která byla popsaná v Sekci 2.4.1. Je zobrazena na Obrázku 6.1.



Obrázek 6.1: Struktura projektu

Každý adaptér i doména má svůj vlastní model. Mezi modelem každého adaptéru a modelem domény existuje mapování. Každý adaptér interaguje pouze s doménou, žádný adaptér nekomunikuje s jiným na přímo. To sice přidává množství kódu v podobě tříd reprezentujících různé objekty, ale přináší to určité výhody. Odděluje vazby mezi adaptéry a doménou, což je využití principu *low coupling, high cohesion*. V případě změny v adaptéru (např. změna formátu dat z KOS API) či v případě potřeby jeho výměny se tak změna dotkne pouze daného adaptéru a na zbytek aplikace nemá vliv.

Pro mapování mezi modely se používají *extension functions*, což je vlastnost jazyka Kotlin.

```
// ImportedCourse is an adapter object, Course is a domain object
fun ImportedCourse.toDomain(): Course {
    // necessary mapping done here
    return Course(...)
}
// in a class responsible for importing data from KOS
override fun importCourse(
    subjectCode: String,
    semesterCode: String?
): Course =
    kosClient.fetchCourse(subjectCode, semesterCode).toDomain()
```

Výpis kódu 4: Mapování mezi adaptéry a doménou

Dále jsou zde třídy rozděleny do několika adresářů, které obsahují třídy podle jejich zodpovědnosti – client, repository service, facade.

Client Třídy typu client jsou součásti adaptérů. Tyto třídy reprezentují konkrétní implementaci pro operace s externími systémy. Jedná se například o třídy, které zprostředkovávají komunikaci s KOS API, FIT OAuth nebo Usermap API.

Repository Tyto třídy jsou abstrakcí nad datovou vrstvou, jsou opět součástí adaptérů. Využívají konceptu Spring Repository, které dodávají metody pro provádění dotazů v databázi bez nutnosti jejich psaní pro konkrétní implementaci.

Service Tyto třídy jsou již součástí domény. Obsahují v sobě závislosti na třídy z client a repository (abstraktně přes interface) a dodávají metody, které je možné využít ve zbytku domény pro vytvoření potřebné business logiky.

Facade Poslední důležitou částí jsou facades. Jedná se o doménové třídy, které reprezentují business logiku. Poskytují tedy operace jako je provedení importu předmětu, změna zámku entity atd. Závislosti mají pouze na třídách typu service. Nejsou tedy vázané na konkrétní implementaci databázové vrstvy či externích systémů pro import.

6.3 Autentifikace, autorizace

Pro získání hodnot jako je client ID a client secret je potřeba založit dvě aplikace v Apps Manageru [39]. První je aplikace typu *service account*, který slouží pro mezisystémovou autentifikaci a je pro potřeby backendu samotného. Druhá je typu *web applicaiton*, která bude použita pro ověřování uživatelů. V tomto systému pro správu projektů je také potřeba povolit potřebné scopes zmíněné v Sekci 5.3.1.

6.3.1 Mezisystémová autentifikace

Spring umožňuje nastavit konfiguraci komunikace s OAuth serverem přímo v jeho *WebClientu*, což je třída, která umožňuje provádět HTTP požadavky.

```
// part required by WebClient builder
private fun clientRegistration(): ReactiveClientRegistrationRepository {
    val registration = ClientRegistration
        .withRegistrationId(clientRegistrationId)
        .tokenUri("$authAddress/oauth/token")
        .clientId(clientId)
        .clientSecret(clientSecret)
        .authorizationGrantType(CLIENT_CREDENTIALS)
        .build()
    return InMemoryReactiveClientRegistrationRepository(registration)
}
```

Výpis kódu 5: Client credentials WebClient

Postačí tedy nastavit správné parametry, jako client ID, client secret, adresu autorizačního serveru atd. Tyto hodnoty jsou konfigurovatelné. Výměnu za access token a jeho ukládání do vypršení platnosti poté řeší třídy Springu. Tento WebClient lze poté využít k provádění požadavků např. do KOS API.

```
// shortened for example purposes
override fun fetchProgramme(programmeCode: String):
  ImportedProgramme =
    webClient.get()
        .uri("/programmes/$programmeCode")
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError, this::handleErrors)
```

Výpis kódu 6: Dotaz do KOS API

6.3.2 Autentizace uživatelů

Pro generování JWT tokenů byla použita Auth0 knihovnu [35]. Pro *authorization code* grant již WebClient není předem nakonfigurovaný, jelikož autorizační kód dostává na vstupu od uživatele. Flow pro prvotní přihlášení včetně následovného vygenerování JWT lze vidět ve Výpisu kódu 7.

```
override fun generateJwt(authorizationCode: String): JwtTokenInfo {
    val accessToken = OAuthClient.exchangeAuthorizationCode(authorizationCode)
                .accessToken
    val username = checkToken(accessToken).username
    val userDetails = usermapClient.getUser(username)
    return JwtTokenInfo(
        jwt = createJwt(username, roles = userDetails.roles),
        expirationSeconds = jwtExpirationSeconds
    )
}

private fun createJwt(username: String, roles: List<String>): String {
        val expiration = Instant.now()
            .plus(jwtExpirationSeconds, ChronoUnit.SECONDS)
        return JWT.create()
            .withIssuer(ISSUER)
            .withClaim(USERNAME, username)
            .withClaim(ROLES, roles)
            .withExpiresAt(expiration)
            .sign(algorithm)
    }
```

Výpis kódu 7: Generování JWT

Těchto metod se využívá při přihlašování uživatele a také v třídě *TokenAuthFilter*, která dědí z *OncePerRequestFilter*. Jedná se o třídy, které vykonávají metody před přeposláním požadavku do *controller* tříd. V tomto filtru se tento JWT ověřuje a nastavuje se z něj *SecurityContext* Springu, který je platný pouze pro daný požadavek (požadavky se navzájem neovlivňují). Pokud JWT ve filtru není úspěšně ověřen, je vrácena odpověď HTTP 401.

Tohoto *SecurityContextu* lze následně využít k ověřování práv uživatele na základě jeho rolí, což lze vidět ve Výpisu kódu 8. V případě, že uživatel nemá správné role, chyba je zpropagována do API a je vrácena odpověď HTTP 403.

```
const val READER = "B-18000-SUMA-ZAMESTNANEC"
const val EDITOR = "T-AKRMAT-18000-EDITOR"
const val ADMIN = "T-AKRMAT-18000-ADMINISTRATOR"
...
@PreAuthorize("hasAnyRole('$READER', '$EDITOR', '$ADMIN')")
override fun findById(id: String): Course
```

Výpis kódu 8: Ověřování rolí

6.4 Generování šablon

Generování šablon bylo implementováno poměrně přímočarým způsobem. Aplikace má adresář *resources/forms*, kde jsou uložené šablony s placeholder hodnotami. Pro tyto placeholder hodnoty je v programu definováno, za jaké proměnné se pro konkrétní entitu mají nahradit.

```
// example from mapping file
placeholders["CORRECTION_OVERLAP"] = course.correlationOverlap.orEmpty()
// form generation
override fun generateForm(
    path: String,
    placeholderMappings: Map<String, String>
): String {
    val template = getResourceAsText(path)
    var output = template
    for ((placeholder, value) in placeholderMappings) {
        output = output.replaceFirst(placeholder, value)
    }
    val outputStream = ByteArrayOutputStream()
    OutputStreamWriter(outputStream).use { it.write(output) }
    return outputStream.toString()
}
```

Výpis kódu 9: Generování šablon

6.5 Databáze

6.5.1 Dotazy

Pro provádění dotazů nad MongoDB je využito Spring *repositories*. Jedná se o funkcionalitu Springu, která abstrahuje dotazování od konkrétní implementace databáze. Pomocí přirozených slov a jmen atributů lze psát metody, které budou automaticky převedeny do dotazů pro konkrétní databázi. Pokud bychom chtěli najít například předmět podle zkratky a kódu semestru, stačí vytvořit metodu v příslušném repository s názvem *findByAbbreviationAndSemesterCode(abbrev, code)*. Příklad tohoto repository je zobrazen ve Výpisu kódu 10.

```
@Repository
interface CourseRepository : MongoRepository<Course, String>
```

Výpis kódu 10: Spring repository

6.5.2 Integrita dat

MongoDB nepodporuje integritní omezení jako klasické relační databáze. Byl zde jeden případ, kde byla potřeba jedno omezení tohoto typu implementovat. Většina entit má nějaký parametr, který může nabývat duplicitních hodnot, avšak tyto entity je potřeba nějak odlišit, čehož je dosažené kombinací s verzí entity v metadatech. Jedná se například o kód programu - může existovat několik programů se stejným kódem, ale ty jsou od sebe odlišené verzí. Pro dosažení této integrity na databázové úrovni jsou využity složené indexy s unikátností nastavenou na *true*.

Pro ověřování struktury dokumentů lze využít JSON schema. Bohužel Spring ani Kotlin nenabízí způsob, jakým by šlo tento proces automatizovat a tudíž jediný způsob je prozatím vytvořit toto schéma manuálně a do databáze ho vložit. Vzhledem k tomu, že projekt je v prvotní fázi a struktura dat se může měnit, v implementaci toto schéma přiloženo není.

6.6 API

API je implementováno v souladu s REST principy. Je stateless, používá standardních jmenných konvencí a smysluplně využívá status kódů pro výsledky požadavků (např. HTTP 201 = entita vytvořena).

6.6.1 Content-Types

Jelikož o entity je možné žádat v různých formátech (JSON, XML, TEX, HTML), API tyto požadavky musí zpracovávat. Pro tyto účely se využívá HTTP hlaviček *Accept* a *Content-Type*. Hlavičku Accept posílá klient pro vyžádání konkrétního datového typu a na jejím základě je nastavena hlavička odpovědi Content-Type a je vrácena příslušná reprezentace zdroje. Ve výchozím případě (hlavička Accept není poslána) se vrací JSON, v případě nepodporovaného datového formátu je vrácena chyba.

6.6.2 Zpracování chyb

Pro zpracování chyb používá API vlastní *exception handler*. Jedná se o třídu, jejíž zodpovědností je odchytávat výjimka aplikace a transformovat je na vhodnou chybovou hlášku s příslušným status kódem.

```
@ExceptionHandler(ContentNotSupported::class)
fun handleContentNotSupported(ex: ContentNotSupported):
ResponseEntity<Any> =
    ResponseEntity(ex.message, HttpStatus.BAD_REQUEST)
```

Výpis kódu 11: Zpracování výjimek v API

API také validuje potřebné hodnoty na vstupech. V případě chybějících požadovaných hodnot se klientovi vrátí HTTP 400 s popisem, které hodnoty jsou vyžadovány.

6.7 Migrace dat

Pro migraci dat byla dodána data ze starého systému v podobě XML souborů. Tyto XML soubory jsou tří typů podle entit, které reprezentují (učitelé s publikacemi, programy se specializacemi, předměty).

Pro každou z těchto entit byl vytvořen nový endpoint v API (ve stylu *POST /courses/migration*), který na vstupu akceptuje entitu ve formátu ze starého systému. Tento formát má poté definované mapování do současných doménových objektů. Tyto transformované objekty poté šlo již jednoduše uložit do nové databáze. Díky vhodně navržené architektuře systému byla jediná práce s migrací napsání objektů reprezentujících starý formát a jejich následné mapování do doménových tříd.

Pro migraci všech entit byl použit Python skript, který bere tyto XML data a pro každý soubor zavolá odpovídající endpoint API, čímž se přemigrují všechna data. Tento skript včetně dat lze najít v příloze v adresáři *akrmat-v4-backend/scripts*.

```
@JacksonXmlRootElement(localName = "predmet")
data class LegacyCourse(
    @JacksonXmlProperty(localName = "metadata")
    val metadata: LegacyMetadata,
    @JacksonXmlProperty(localName = "content")
    val content: LegacyCourseContent
)
```

Výpis kódu 12: Formát pro XML starého systému

6.8 Git, Continous Integration

6.8.1 Verzování

Pro verzování aplikace byl použit program Git v kombinaci se školním GitLab prostředím. Pro formát commit zpráv bylo využito *conventional commits* formátu [36].

```
feat: Implement publications facade
fix: Add includeMetadata flag to Publication mapping
test: Add IT tests for publications
```

Výpis kódu 13: Příklad conventional commits formátu

6.8.2 Continous Integration

Pro aplikaci bylo využito také CI GitLabu. Každý push do remote repozitáře spustí pipeline, která aplikaci zkompiluje a spustí testy. Tato pipeline musí skončit úspěchem, aby mohla být zamergována do *main* větve. Ve Výpisu kódu 14 je zobrazena část definice této pipeline.

```yaml
# shortened for example purposes
stages:
  - compile
  - test

compile:
  image: maven:3-jdk-11
  stage: compile
  script: "mvn compile"

unit-test:
  image: maven:3-jdk-11
  stage: test
  script: "mvn test"

integration-test:
  image: maven:3-jdk-11
  stage: test
  script: "mvn verify -DskipUTs"
```

Výpis kódu 14: GitLab CI

6.9 Docker

Pro aplikaci je připraven jak *Dockerfile* tak *docker-compose.yml* soubor. Dockerfile slouží k vytvoření Docker image pomocí kterého lze aplikaci jednoduše spustit jako Docker kontejner. Compose yaml soubor obsahuje definici pro spuštění tohoto kontejneru a také MongoDB kontejneru.

Jelikož v době psaní práce nebylo domluveno, jak a kde přesně aplikace bude běžet v běžném provozu, tyto soubory byly předběžně vytvořeny, pokud by bylo žádoucí ji pouštět jako kontejner. Zároveň to také může usnadnit vývoj pro frontend, jelikož kdokoliv, kdo ho bude vyvíjet, si aplikaci může sám jednoduše spustit jako tento kontejner bez nutnosti instalace a konfigurace Kotlinu a dalších potřebných částí.

6.10 Konfigurace aplikace

Jelikož aplikace je vytvořená v kombinaci se Spring frameworkem, ke konfiguraci je použito standardního způsobu pomocí *application.yml* souborů. Hodnoty, které lze nastavit jsou například adresy externích systémů (OAuth, KOS API) či client ID a client secret pro OAuth flow. Pro kompletní seznam hodnot se lze odkázat na soubor *src/main/resources/application.yml* v příloze práce.

6.10.1 Profily

Aplikace obsahuje několik Spring profilů. Tyto profily umožňují modifikovat chování aplikace a také definovat, který *yaml* soubor se použije pro konfigurační hodnoty. Spring použije *yaml* soubor dle jména profilu ve formátu *application-<profile-name>.yml*. Detaily použití konfiguračních souborů lze najít v dokumentaci [37]. Profily v aplikaci jsou následující:

- *default* - Výchozí profil. S tímto profilem je také předpokládaný provoz aplikace pro běžné používání.
- *dev* - Profil určený pro použití během vývoje. Používá *yaml* file s hodnotami, které se během vývoje mohou měnit.
- *disable-auth* - Profil, který je také určený pro potřeby vývoje. Vypíná autentifikaci a autorizaci, aplikace tedy nevyžaduje platný JWT token a nekontroluje role ze *SecurityContext*.

6.10.2 Nastavení konfiguračních hodnot

Pro běžný provoz aplikace je předpokládáno využití *environmental variables* pro nastavení potřebných hodnot. Standardní způsob je využití *.env* souboru, který nastavuje jejich hodnoty.

6.11 Dokumentace

Aplikace je dokumentována 2 způsoby. První je dokumentace API, druhá je dokumentace aplikace jako celku.

6.11.1 Dokumentace API

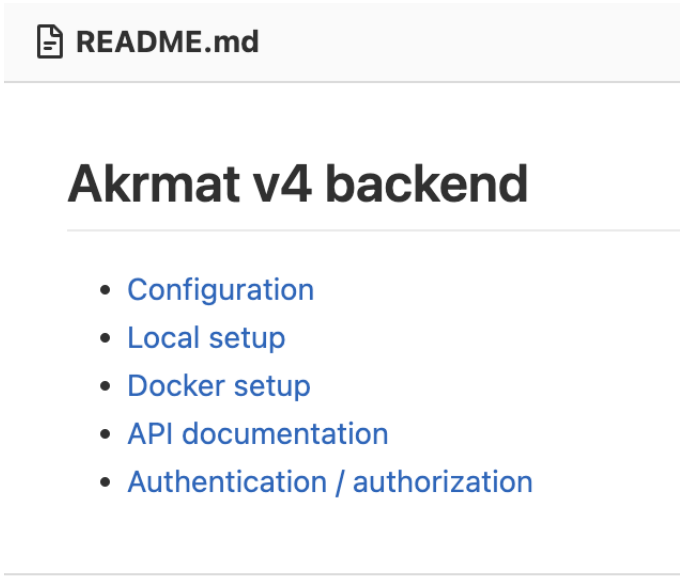
Pro dokumentaci API se používá SpringDoc [38]. Tato knihovna dokáže automaticky generovat dokumentaci API na základě *controller* tříd dle formátu OpenAPI a jeho následného zobrazení ve Swagger UI. Výhoda tohoto způsobu je, že ušetřuje ruční psaní této dokumentace, která by byla v tomto případě poměrně zdlouhavá, a také okamžitě promítá změny, které jsou v *controller* třídách provedeny. Nevýhodou je, že v některých případech vygeneruje pouze nekompletní verzi dokumentace, což však lze opravit konfigurací knihovny a příslušnými anotacemi. Příklad je uveden ve Výpisu kódu 15.

```
@Operation(
    summary = "Import Course", responses = [ApiResponse(
        description = "Created",
        responseCode = "201"
    ), ApiResponse(
        description = "Not found",
        responseCode = "404"
    ), ApiResponse(
        description = "Issue communicating with external service",
        responseCode = "500"
    )]
)
@PostMapping("/courses")
@ResponseStatus(HttpStatus.CREATED)
fun importCourse(...)
```

Výpis kódu 15: Konfigurace SpringDoc

6.11.2 Dokumentace aplikace

Pro další potřeby dokumentace je ve zdrojovém kódu také přiloženo několik Markdown souborů. Tyto soubory popisují různé části aplikace, její konfiguraci či spuštění. Tyto soubory lze najít v adresáři *akrmat-v4-backend/documentation*.



Obrázek 6.2: Dokumentace aplikace

6.12 Plány do budoucna

Do budoucna se předpokládá údržba a dodatečný vývoj samotného backendu, tak i frontendu, který s tímto backendem bude interagovat.

Z hlediska backendu samotného je zde několik věcí, které by bylo možné vylepšit. Do API by mohlo být přidáno stránkování, což bylo i diskutováno s vedoucím práce, ale prozatím bylo rozhodnuto, že potřeba není. Při větším nárůstu entit v databázi by toto mohla být část, která zabere velkou část dotazu kvůli přenosu většího objemu dat. I přestože aplikace je pokrytá testy, je zde prostor připsat další, které budou detailněji kontrolovat chování aplikace. Další věcí je zamyšlení se nad generováním formulářů. Současné řešení není špatné, ale šlo by určitě zlepšit. Hlavním nedostatkem je, že zde není žádná kontrola toho, jaké placeholdery v šablonách jsou a jakými se nahrazují. Poslední je import publikací z V3S či jiného alternativního zdroje, což v této práci nebylo implementováno z důvodu nedostupnosti tohoto zdroje. Jakmile bude dostupný, bude možné provést analýzu a implementaci do aktuálního systému.

Až bude implementovaný i frontend, určitě by bylo vhodné provést i uživatelské testování systému jako celku.

Kapitola 7

Testování

Aplikace je pokryta jednotkovými a integračními testy. Jelikož rozsah zadání pokrývá pouze vytvoření backendu, další typ testů jako například uživatelské testování prozatím nedává smysl. Kromě pokrytí automatizovanými testy byl také progres prezentován na pravidelných konzultacích s vedoucím práce, který také navíc měl vždy dostupnou poslední verzi vývoje v GitLab repozitáři.

Pro potřeby testování bylo využito knihovny JUnit 5 [40]. Pro integrační testy bylo využito knihovny TestContainers [41], která spouští Mongo databázi pro integrační testy. Pro mockování bylo využito knihovny MockK [42].

Jednotkové testy, jak již z jména vyplývá, testují pouze izolované části kódu. V rámci jednoho testu by se měla testovat pouze jedna metoda z jedné třídy. Pokud má nějaké závislosti, tyto závislosti by měly být dodány jako *mocks*. Příklad je uveden v následujícím Výpisu kódu 16.

```
@Test
fun `Generate teacher tex form`() {
    val teacher = TestEntities.teacherDomain()
    val teacherPlaceholders = TeacherTexMapping
        .getDefaultFormPlaceholders(teacher)
    val path = FULL_TEACHER_TEX_PATH

    val generatedForm = formGeneratorService
        .generateForm(path, teacherPlaceholders)

    val expectedResult = readResponse("teacher_form", EXTENSION_TEX)
    assertEquals(generatedForm, generatedForm)
}
```

Výpis kódu 16: Jednotkový test

Integrační testy testují aplikaci jako celek. Jediná část, kterou tyto testy mockují jsou systémy třetích stran. V případě této aplikace se jedná například o KOS API. Toto lze vidět v příkladu integračního testu ve Výpisu kódu 17.

```
// from tests configuration class
@MockkBean
lateinit var kosClient: KosClient
// from Course integration test class
@Test
fun `Import course with valid input`() {
    val kosApiCourse = TestEntities.importedCourse()
    every { kosClient.fetchCourse(any(), any()) } returns kosApiCourse

    val response = mvc.perform(
        MockMvcRequestBuilders
            .post("/courses")
            .contentType(MediaType.APPLICATION_JSON)
            .content(readRequest("import_course"))
            .cookie(allRolesCookie)
    ).andExpect(MockMvcResultMatchers.status().isCreated)
        .andReturn().response.contentAsString

    verify(exactly = 1) { kosClient.fetchCourse(any(), any()) }
    assertTrue(response.isNotEmpty())
    val importedCourse = courseService.findById(response)
    assertEquals(kosApiCourse.content.code, importedCourse.abbreviation)
    assertEquals(kosApiCourse.content.credits, importedCourse.creditAmount)
    courseService.delete(importedCourse.id!!)
}
```

Výpis kódu 17: Integrační test

Kapitola 8

Závěr

V rámci této diplomové práce byly zanalyzovány předchozí verze systému Akrmat a na základě této analýzy byl proveden návrh nového systému včetně vylepšení a doplnění funkcionalit. Tento návrh byl implementován v jazyce Kotlin s využitím MongoDB databáze.

Nová implementace backendu Akrmatu zachovává funkcionality starého systému a zároveň obsahuje vylepšení jako import dat z KOS API, požadované rozdělení entit v datovém modelu a integraci s již existujícími uživateli systému FIT pomocí fakultního OAuthu. Hlavní části systému jsou pokryté automatizovanými jednotkovými a integračními testy. Systém obsahuje automaticky generovanou dokumentaci API a také dokumentaci v podobě Markdown souborů, která popisuje konfigurační položky potřebné pro správné nastavení a spuštění projektu.

Výsledkem práce je funkční implementace backendu tohoto systému, která je připravena na integraci s novým frontendem, který je zatím ve fázi plánování a analýzy.

Literatura

[1] Růžička, L.: Aplikace pro správu akreditačních materiálů - část anotace předmětů. Diplomová práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2012.

[2] Kabelka, M.: Aplikace pro správu akreditačních materiálů - část odborné životopisy učitelů. Diplomová práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2012.

[3] Štefan, J.: Systém Akrmat verze 3. Diplomová práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2012.

[4] Vitvar, T. [online] Representational State Transfer [cit. 2023-03-26]. Dostupné z: https://mdw.vitvar.com/lecture5.html

[5] REST – All You Have To Know About Representational State Transfer. [online] Plesk © 2020 [cit 2023-03-26]. Dostupné z: https://www.plesk.com/blog/various/rest-representational-state-transfer

[6] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. Dizertační práce, University of California, Irvine.

[7] What is REST (REpresentational State Transfer)? [online]. TechTarget © 2021 [cit 2023-03-26]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/REST-REpresentational-State-Transfer

[8] Idempotent - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. [online] Mozilla © 1998–2023 [cit 2023-03-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Idempotent

[9] Safe (HTTP Methods) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. [online] Mozilla © 1998–2023 [cit 2023-03-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTT

[10] What Is SQL Database? - IT Glossary | SolarWinds. [online] SolarWinds Worldwide © 2023 [cit 2023-03-26]. Dostupné z: https://www.solarwinds.com/resources/it-glossary/sql-database

[11] ACID Properties In DBMS Explained | MongoDB | MongoDB. [online] MongoDB © 2023 [cit 2023-03-26]. Dostupné z: https://www.mongodb.com/basics/acid-transactions

[12] What is the CAP theorem? | IBM. [online] IBM © 2023 [cit 2023-03-26]. Dostupné z: https://www.ibm.com/topics/cap-theorem

[13] What is CAP Theorem? Definition FAQs | ScyllaDB. [online] ScyllaDB © 2023 [cit 2023-03-26]. Dostupné z: https://www.scylladb.com/glossary/cap-theorem/

[14] The basics of NoSQL databases—and why we need them. [online] Free Code Camp © 2019 [cit 2023-03-26]. Dostupné z: https://www.freecodecamp.org/news/nosql-databases-5f6639ed9574/

[15] ACID vs. BASE: Comparison of Database Transaction Models. [online] phoenixNAP © 2022 [cit 2023-03-26]. Dostupné z: https://phoenixnap.com/kb/acid-vs-base

[16] What is Normalization in DBMS (SQL)? 1NF, 2NF, 3NF Example. [online] Guru99 © 2023 [cit 2023-03-26]. Dostupné z: https://www.guru99.com/database-normalization.html

[17] How Do You Authenticate, Mate? [online]. Medium © 2023 [cit 2023-03-26]. Dostupné z: https://betterprogramming.pub/how-do-you-authenticate-mate-f2b70904cc3a

[18] Cookies vs. Tokens: The Definitive Guide - DZone. [online] DZone © 2016 [cit 2023-03-26]. Dostupné z: https://dzone.com/articles/cookies-vs-tokens-the-definitive-guide

[19] JSON Web Token Introduction - jwt.io. [online] Okta © 2023 [cit 2023-03-26]. Dostupné z: https://jwt.io/introduction

[20] M. Jones and J. Bradley and N. Sakimura, JSON Web Token (JWT) RFC 7519. Dostupné z: https://www.rfc-editor.org/rfc/rfc7519

[21] JSON Web Token Structure. [online] Auth0 © 2013-2023 [cit 2023-03-26]. Dostupné z: https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-structure

[22] OAuth 2.0 - OAuth. [online] OAuth © 2023 [cit 2023-03-26]. Dostupné z: https://oauth.net/

[23] D. Hardt, The OAuth 2.0 Authorization Framework RFC 6479. Dostupné z: https://www.rfc-editor.org/rfc/rfc6749

[24] What is OAuth 2.0 and what does it do for you? - Auth0. [online] Okta © 2023 [cit 2023-03-26]. Dostupné z: https://auth0.com/intro-to-iam/what-is-oauth-2

[25] OAuth 2.0 Scopes. [online] OAuth © 2023 [cit 2023-03-26]. Dostupné z: https://oauth.net/2/scope/ https://oauth.net/2/scope/

[26] The Client ID and Secret - OAuth 2.0 Simplified. [online] Okta © 2023 [cit 2023-03-26]. Dostupné z: https://www.oauth.com/oauth2-servers/client-registration/client-id-secret/

[27] OAuth Grant Types. [online] OAuth © 2023 [cit 2023-03-26]. Dostupné z: https://oauth.net/2/grant-types/ https://oauth.net/2/grant-types/

[28] OAuth 2.0 Grant Types. [online] VMWare © 2023 [cit 2023-03-26]. Dostupné z: https://docs.vmware.com/en/Single-Sign-On-for-VMware-Tanzu-Application-Service/1.14/sso/GUID-grant-types.html

[29] Software Architecture Guide. [online] Martin Fowler © 2023 [cit 2023-03-26]. Dostupné z: https://martinfowler.com/architecture/

[30] Ready for changes with Hexagonal Architecture. [online] Netflix © 2023 [cit 2023-03-26]. Dostupné z: https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

[31] Hexagonal Architecture, there are always two sides to every story. [online] Medium © 2023 [cit 2023-03-26]. Dostupné z: https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c

[32] Jambor, J.: IS pro tvorbu a generování akreditačních materiálů. Diplomová práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2013.

[33] Resources - KOSapi - Projekt KOSapi. [online] ČVUT © 2023 [cit 2023-03-26]. Dostupné z: https://kosapi.fit.cvut.cz/projects/kosapi/wiki/Resources

[34] Kotlin Docs | Kotlin Documentation. [online] JetBrains © 2023 [cit 2023-03-26]. Dostupné z: https://kotlinlang.org/docs/home.html

[35] auth0/java-jwt: Java implementation of JSON Web Token (JWT). [online] Auth0 © 2023 [cit 2023-03-26]. Dostupné z: https://github.com/auth0/java-jwt

[36] Conventional Commits . [online] Conventional Commits © 2023 [cit 2023-03-26]. Dostupné z: https://www.conventionalcommits.org/en/v1.0.0/

[37] 24. Externalized Configuration. [online] VMware © 2023 [cit 2023-03-26]. Dostupné z: https://docs.spring.io/spring-boot/docs/2.1.9.RELEASE/reference/html/boot-features-external-config.html

[38] OpenAPI 3 Library for spring-boot. [online] Badr NASS LAHSEN © 2023 [cit 2023-03-26]. Dostupné z: https://springdoc.org/

[39] Apps Manager BETA. [online] České vysoké učení technické v Praze © 2014 [cit 2023-03-26]. Dostupné z: https://auth.fit.cvut.cz/manager/index.jsf

[40] JUnit 5. [online] The JUnit Team © 2023 [cit 2023-03-26]. Dostupné z: https://junit.org/junit5/

[41] Testcontainers for Java. [online] Richard North © 2023 [cit 2023-03-26]. Dostupné z: https://www.testcontainers.org/

[42] MockK | mocking library for Kotlin. [online] MockK © 2023 [cit 2023-03-26]. Dostupné z: https://mockk.io/

Příloha **A**

Obsah příloh

- `akrmat-v4-backend`
  - `documentation` ..... dokumentace v Markdown formátu
  - `analysis` ..... soubory z fáze analýzy v Markdown formátu
  - `scripts` ..... migrační skript a data pro migraci
  - `src` ..... zdrojové kódy v Kotlinu
- `dthesis-text`
  - `src` ..... text práce v LaTeX formátu
  - `pdfs`
    - `stefaja7_akrmat_thesis.pdf` ..... text práce PDF formátu