FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

DMS pro evidenci dokumentace k SW

Jakub Pavlíček

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

DMS pro evidenci dokumentace k SW

Jakub Pavlíček

Vedoucí práce
Ing. Martin Dostal, Ph.D.

PLZEŇ 2024

Citace v seznamu literatury:
PAVLÍČEK, Jakub. *DMS pro evidenci dokumentace k SW*. Plzeň, 2024. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Martin Dostal, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub PAVLÍČEK**
Osobní číslo:	**A21B0234P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**DMS pro evidenci dokumentace k SW**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prostudujte problematiku DMS (Document Management System) i s ohledem na způsoby detekce možných duplicit. Dále prostudujte systémy pro práci s REST API (např. Swagger).
2. Analyzujte požadavky na REST API z pohledu dostupnosti a bezpečnosti. Dále analyzujte možnosti různých technologií, platforem a nástrojů.
3. Na základě provedené analýzy navrhněte vlastní REST API pro úložiště souborů. Při návrhu klaďte důraz na celkové zabezpečení aplikace a podporu několika vybraných SŘBD pro uložení metadat o souborech (např. verzování).
4. Navržené REST API implementujte ve zvoleném programovacím jazyce a dbejte na důslednou programátorskou dokumentaci.
5. Výslednou implementaci otestujte sadou netriviálních testů. Zhodnoťte dosažené výsledky.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Martin Dostal, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **2. října 2023**
Termín odevzdání bakalářské práce: **2. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2023

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Plzni dne 1. května 2024

. . . . . .

Jakub Pavlíček

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací systému pro správu dokumentů, který umožňuje efektivní řízení dokumentace. Pro komunikaci s tímto systémem je využito REST API. Práce se detailně zabývá různými aspekty ukládání a verzování dat a zkoumá možnosti detekce duplicitních souborů, což je klíčový prvek pro udržení integrity a struktury uložených informací. Dále popisuje REST API z hlediska jeho dostupnosti a bezpečnosti, což je zásadní pro správnou komunikaci a manipulaci s daty v DMS. Práce rovněž představuje široké spektrum technologií, platforem a nástrojů určených pro práci s REST API. Po důkladné analýze bylo zvoleno vhodné úložiště a na základě toho bylo navrženo a vytvořeno REST API, které umožňuje efektivní komunikaci s tímto úložištěm. Výsledná aplikace je následně podrobena řádnému testování, aby byla ověřena její funkčnost a spolehlivost.

Abstract

This bachelor thesis deals with the design and implementation of a Document Management System that enables efficient documentation management. REST API is used for communication with this system. The thesis extensively examines various aspects of data storage and versioning and explores options for detecting duplicate files, which is a key element in maintaining the integrity and structure of stored information. Furthermore, it describes REST API in terms of its availability and security, which is crucial for proper communication and manipulation of data in DMS. The thesis also introduces a wide range of technologies, platforms, and tools designed for working with REST API. After thorough analysis, a suitable storage solution was chosen, and based on that, REST API was designed and implemented to enable efficient communication with this storage. The resulting application then undergoes rigorous testing to verify its functionality and reliability.

Klíčová slova

Document Management System • úložiště • dokument • revize • REST API • JSON Web Token

Poděkování

Chtěl bych poděkovat Ing. Martinu Dostalovi, Ph.D. za odborné vedení, čas, trpělivost a cenné rady k práci. Děkuji také Ing. Martinu Šlapovi za odborné vedení praktické části, čas, trpělivost, cenné rady a zkušenosti, které jsem získal v průběhu tvorby této práce.

Obsah

Úvod 1

Při vývoji softwaru vzniká mnoho dokumentů, které je potřeba někde uchovávat. Jednou z nabízených možností je ukládání dokumentů v kartotékách. Toto řešení však nemusí být žádoucí v momentě, kdy už skladujeme velké množství dokumentů, protože velikost kartotéky není neomezená. Práce s dokumenty ve fyzické podobě, jako je například tisknutí či skartování dokumentů, je navíc časově náročná a neekologická. Tento čas mohou zaměstnanci, kteří pracují s jednotlivými dokumenty, využít k důležitějším věcem. Při použití diskového úložiště, ať už lokálního či cloudového, se nemusíme tolik starat o počet uchovávaných dokumentů, protože velikost tohoto úložiště mnohonásobně převyšuje velikost kartotéky. Tím ušetříme i fyzický prostor, který byl dříve nutný pro kartotéky.

Hlavním důvodem pro používání DMS, využívajícího diskové úložiště namísto fyzického, je úspora času a místa. Důvodem pro vytvoření nového DMS je fakt, že již existující DMS jsou příliš komplexní a jejich funkcionalita přesahuje pouhé ukládání či získávání dokumentů. V některých případech také nepodporují verzování dokumentů a pro ukládání dokumentů často využívají cloudová úložiště. Vytvořené DMS bude zaměřeno na jednoduché ukládání, verzování a získávání dokumentů s využitím lokálního úložiště.

Cílem práce je návrh a tvorba jednoduchého DMS, se kterým se bude komunikovat prostřednictvím REST API. Nejprve bude probrána problematika DMS, způsoby ukládání a verzování dokumentů i s ohledem na detekci jejich duplicit. Dále bude věnována pozornost různým systémům pro práci s REST API a bude provedena analýza požadavků týkajících se dostupnosti a bezpečnosti REST API, včetně možností propojení se s ním pomocí různých technologií, platforem a nástrojů. Na základě provedené analýzy bude vytvořena aplikace typu Software as a Service (SaaS) pro centrální uložení dokumentů vzniklých při vývoji aplikací. Přístup k úložišti dokumentů bude umožněn přes REST API. Aplikace bude umožňovat úsporné a verzované ukládání dokumentů s možností archivace a automatickým detekováním duplicit. Bude zajištěna celková bezpečnost aplikace a kompatibilita s několika vybranými systémy řízení báze dat (SŘBD). Správnost implementace bude ověřena manuálními testy spolu s automatickými jednotkovými a integračními testy.

2 Document Management System - moderní přístupy a řešení

V této kapitole bude představen Document Management System, jakožto systém pro správu dokumentů. Z důvodu potřeby ukládání dokumentů zde bude ukázáno, jaké jsou moderní přístupy k ukládání dat a jaké jsou jejich odlišnosti, výhody a nevýhody. Budou zde probrány i již existující řešení velkých firem, jako je například Microsoft či Google. Jelikož při vývoji softwaru často vznikají nové verze či duplicity dokumentů, budou zde ukázány i různé přístupy k verzování a techniky pro detekci duplicit.

Document Management System (DMS) je systém pro elektronickou správu dokumentů. Poskytuje centrální úložiště pro ukládání, správu a přístup k dokumentům. S tímto systémem je možné organizovat dokumenty bez nutnosti jejich uchovávání ve fyzické podobě [1]. Tento systém je již běžně používaný v mnoha organizacích. Důvodem je to, že jeho používání vede ke zvýšení produktivity zaměstnanců, protože urychluje operace, jako je například vyhledávání a mazání dokumentů.

Funkce DMS:

- ukládání dokumentů – měla by být zajištěna možnost bezpečného ukládání dokumentů,
- stahování a zobrazení dokumentů – uložené dokumenty by měly jít stáhnout či zobrazit,
- historie úprav dokumentů – logy, kdo a kdy dokument upravil,
- automatické mazání starých dokumentů – mazání starších dokumentů za účelem uvolnění místa v úložišti,

- podmíněný přístup k dokumentům – k určitým dokumentům se dostanou pouze uživatelé, kteří mají oprávnění,
- vyhledávání dokumentů – mělo by jít vyhledávat dokumenty, ať už podle názvu či podle obsahu,
- kategorizace dokumentů – dokumenty by mělo jít zařadit do kategorií pomocí tagů či metadat.

2.1 Způsoby ukládání dat

Vzniklé dokumenty je třeba někam uložit a právě proto v této podkapitole bude představeno několik způsobů, které lze využít pro ukládání dokumentů. Bude ukázáno ukládání dat pomocí:

- lokálního úložiště,
- cloudového úložiště,
- databáze,
- hybridního přístupu.

U jednotlivých způsobů ukládání bude řečeno, jaké jsou jejich výhody a nevýhody a u cloudového úložiště budou ukázány i již existující řešení, jako je Microsoft Azure Storage či Google Cloud Storage.

2.1.1 Lokální úložiště

Při využití lokálního úložiště je zapotřebí disponovat vlastními servery, což znamená, že je s tím spjatá i jejich údržba a automatické zálohování. Je důležité myslet i na celkovou velikost úložiště, protože se může stát, že místo na disku bude vyčerpáno a nebude pak kam ukládat další dokumenty. V případě vyčerpání místa na disku je pro přidání nového místa vyžadováno větší úsilí, než jen párkrát kliknout myší, jako by tomu bylo v případě využití cloudového úložiště. Každé paměťové médium je náchylné k poruchám, a proto je nutné data zálohovat a to nejlépe na více různých záložních systémech [2]. Oproti cloudovému řešení zde nezávisí na internetovém připojení, protože i v případě selhání připojení lze pořád přistupovat ke všem dokumentům. Z výše uvedených důvodů je zřejmé, že toto řešení je vhodnější spíše pro větší organizace z důvodu vyšších technických nároků. Dalším důvodem je pak plná kontrola nad systémem.

Výhody:

- nezávislost na internetovém připojení,
- rychlý přístup,
- kontrola nad systémem.

Nevýhody:

- velké počáteční náklady,
- nutnost zálohování.

2.1.2 Cloudové úložiště

Při využití cloudového úložiště se data neukládají lokálně, ale ukládají se na vzdálené servery poskytovatelů cloudových služeb [3]. Přístup k uloženým datům je zajištěn prostřednictvím Internetu, což umožňuje přistupovat k datům odkudkoliv. Ačkoliv si poskytovatelé cloudových služeb mohou účtovat nějaký poplatek za jejich využívání, tak alespoň zaručují kvalitní a spolehlivé služby. Zpravidla však platí, že organizace si platí jen za místo v úložišti, které doopravdy využívají, což umožňuje organizacím šetřit peníze.

Výhody:

- malé náklady,
- dostupnost odkudkoliv,
- škálovatelnost,
- data jsou automaticky zálohována.

Nevýhody:

- závislost na poskytovateli služeb,
- závislost na internetovém připojení.

2.1.2.1 Existující řešení

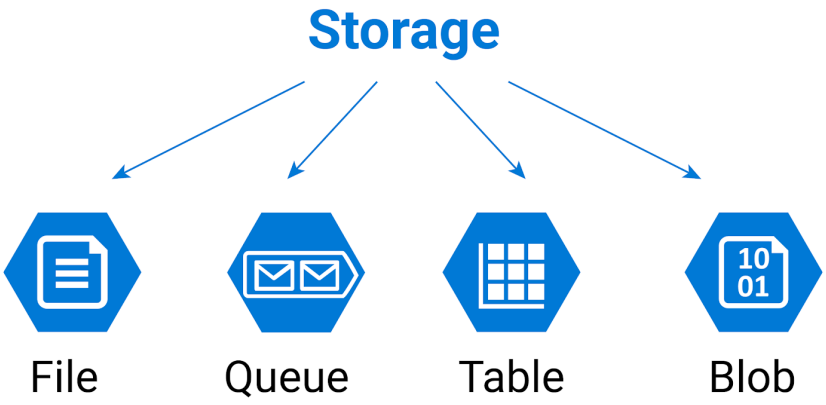
Tato podkapitola slouží k představení již existujících cloudových úložišť velkých společností, jako je Microsoft a Google. Budou zde představeny úložiště Microsoft Azure Storage a Google Cloud Platform a budou zde uvedeny i jejich hlavní výhody.

Microsoft Azure Storage

Azure Storage je moderní cloudové úložiště od společnosti Microsoft pro ukládání dat. Nabízí vysoce dostupné, škálovatelné, odolné a bezpečné úložiště pro různá data, jako jsou např. soubory, fronty zpráv, tabulky či bloby (viz obr. 2.1). Zpřístupňuje data skrze HTTP a HTTPS prostřednictvím REST API, takže je možné se k datům dostat odkudkoliv. Azure Storage nabízí klientské knihovny s podporou několika programovacích jazyků, což umožňuje vývojářům rychleji vyvíjet aplikace či služby [4]. Pomocí Azure Storage Exploreru je možné interagovat s Azure Storage prostřednictvím uživatelského rozhraní.

Výhody:

- odolnost – v případě poruchy hardwaru nedojde ke ztrátě dat,
- bezpečnost – uložená data jsou šifrována,
- škálovatelnost – masivně škálovatelný, aby vyhověl požadavkům na úložiště a výkon současných aplikací,
- dostupnost – odkudkoliv prostřednictvím Internetu,
- správa – Azure se stará o údržbu hardwaru, aktualizace a kritické problémy.



Obrázek 2.1: Ukázka ukládání různých dat v Azure Storage [5]

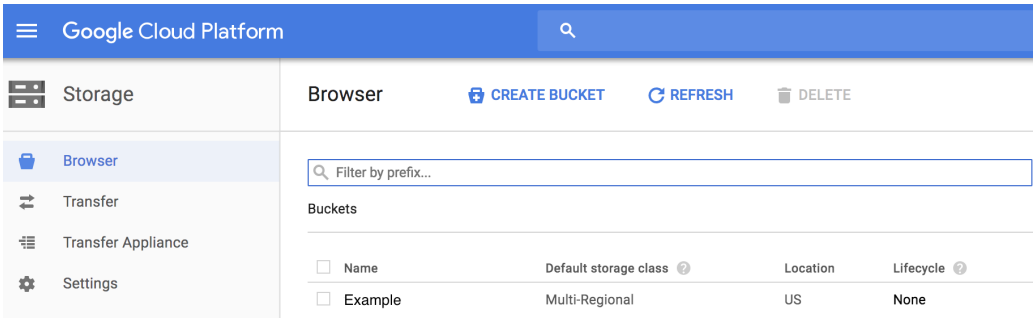
Google Cloud Storage

Cloud Storage je služba od společnosti Google pro ukládání objektů bez starostí o pevné disky, kopie dat a podobné věci [6]. Objekt je neměnná část dat, skládající se ze souboru v libovolném formátu. Při ukládání se objekty uloží v tzv. bucketu

(kbelíku). Každý bucket je spojen s projektem a díky tomu je možné projekty spojit pod jednu organizaci. Při práci s Cloud Storage je nejdříve nutno vytvořit projekt a až následně je v něm možné vytvářet buckety. Nad buckety je pak možné provádět operace, jako je například nahrání či stažení objektu. Cloud Storage dovoluje přidávat přístupová práva, ať už pro jednotlivce či pro celou skupinu uživatelů. Data jsou zpřístupněna prostřednictvím několika technologií, jako je REST API či Google Cloud Console, což je uživatelské rozhraní, které lze vidět na obr. 2.2.

Výhody:

- flexibilita – organizace si mohou vybrat, jak chtějí ukládat a přistupovat k datům,
- bezpečnost – uložená data jsou šifrována a je možné k nim přiřadit přístupová práva,
- škálovatelnost – lze přidávat a ubírat místo na úložišti,
- dostupnost – odkudkoliv prostřednictvím Internetu,
- verzování – možnost povolení verzování objektů.



Obrázek 2.2: Ukázka Google Cloud Console

2.1.3 Databáze

Ukládání dat do databáze je základním prvkem u většiny informačních systémů, protože umožňuje ukládání, manipulaci a vyhledávání dat. Informace jsou sdružovány na jednom místě, což je klíčové pro jejich následnou analýzu.

Při ukládání souborů do databáze je možné použít buď ukládání BLOBů (Binary Large Object), nebo ukládání cest k souborům na disku. V případě ukládání BLOBu jsou všechna data souboru uložena do příslušného sloupce tabulky v databázi. Na

druhou stranu, při ukládání cest k souborům je do daného sloupce uložen pouze řetězec představující cestu k souboru na disku.

Existuje několik typů databází, z nichž nejčastěji používanými jsou relační (SQL) a nerelační (NoSQL) databáze [7]. Relační databáze ukládají data strukturovaně a typicky představují objekty reálného světa. Nerelační databáze naopak nabízejí širokou škálu datových modelů, včetně dokumentového, klíč-hodnota a grafových modelů.

Výhody:

- centralizované ukládání dat,
- strukturovanost dat,
- integrita dat,
- vyhledávání dat,
- snadné zálohování a obnova dat.

Nevýhody:

- nevhodné pro ukládání objemných dat,
- omezená flexibilita schématu u relační databáze.

2.1.4 Hybridní přístup

Hybridním přístupem je myšleno to, že určitá část dat je uložena v databázi a zbytek dat je uložen buď v lokálním či cloudovém úložišti. Obvykle se tento přístup volí v momentě, kdy chceme ukládat objemná data (např. dokumenty) s tím, že metadata budou uložena v databázi a zbytek (BLOBy) se uloží do požadovaného úložiště [8].

Výhody:

- flexibilita – možné vybrat, kam budou jednotlivá data uložena,
- bezpečnost – citlivá data uložena v privátním úložišti a méně citlivá data uložena v databázi,
- zlepšená škálovatelnost – možnost využívání výhod různých typů úložišť pro různé části dat.

Nevýhody:

- data nejsou uložena na jednom místě,
- náročnější správa a monitoring.

2.2 Způsoby verzování dat

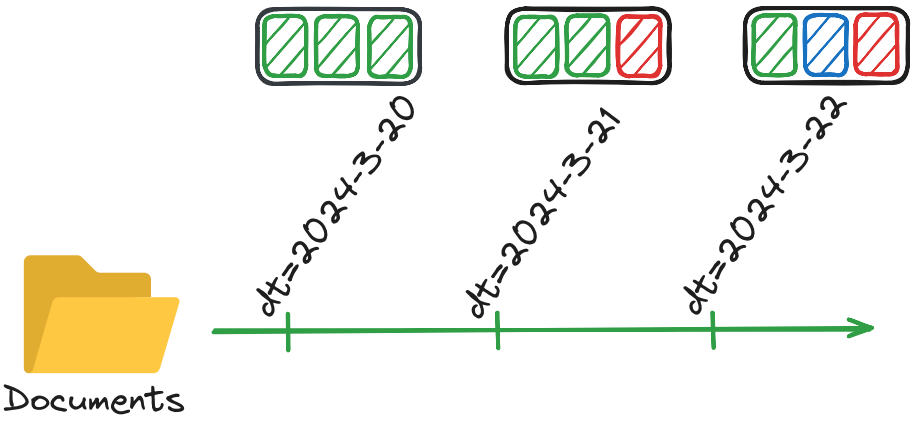
Při vývoji softwaru je běžné, že se mění podmínky či požadavky na daný software a vývojáři na ně reagují ať už změnou kódu či dokumentace. Proto je žádoucí, aby byl zajištěn nějaký verzovací systém pro ulehčení vývoje. Způsoby, kterými lze verzování dat zajistit, jsou následující:

- úplná duplikace dat,
- metadata valid_from/to,
- Data Version Control nástroje.

Tato podkapitola slouží jako seznámení se s těmito způsoby.

2.2.1 Úplná duplikace dat

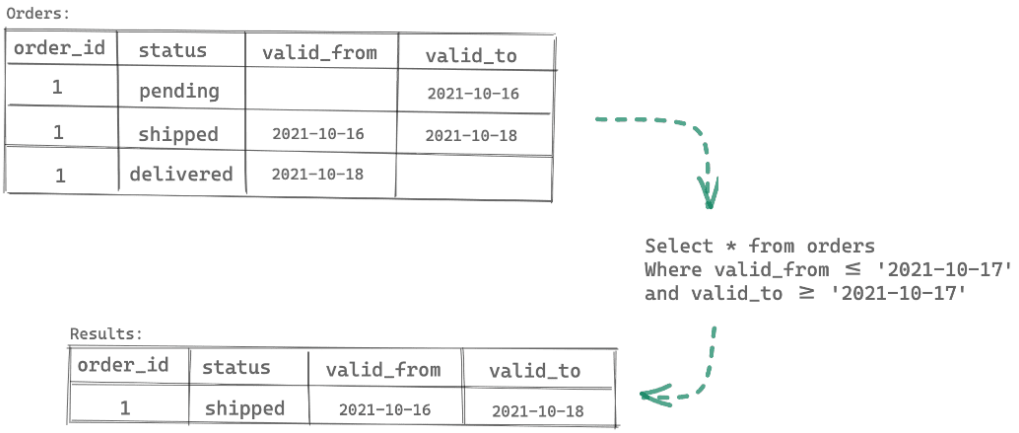
V případě použití tohoto způsobu se nová verze dat uloží pokaždé na nové místo, které je označeno časovým razítkem, kdy byla verze vytvořena. Tento způsob však využívá velké množství místa, protože pokud se v nové verzi změnila jen část dat, tak i přes to byla vytvořena úplně nová sada dat a stará data byla ponechána v podobě, ve které byla před vytvořením nové verze. Dotazy nad verzovanými daty jsou však náchylné k chybám, protože je nutné ručně zadávat správná časová razítka, aby se uživatel dostal k požadovaným verzím dat. Z těchto důvodů je tato metoda vhodnější spíše pro menší sady dat, kde nové verze vznikají méně často [9]. Na obr. 2.3 jsou zeleným obdélníkem vyznačeny části dat, které se v jednotlivých verzích nemění a které jsou duplikovány napříč jednotlivými verzemi.



Obrázek 2.3: Ukázka úplné duplikace dat

2.2.2 Metadata valid_from/to

Tento způsob řeší problém velké spotřeby místa u úplné duplikace dat tím, že místo vytváření kopie používá v databázi metadata (sloupce) valid_from a valid_to, které říkají, od kdy a do kdy jsou data validní. Pokud dojde k vytvoření nové verze dat, existující záznamy nebudou nahrazeny, ale pouze budou přidány nové záznamy. Pro všechny záznamy, které by byly nahrazeny, budou aktualizovány položky valid_to na aktuální čas [9]. Při psaní dotazů nad verzovanými daty je však nutné využívat filtry na metadata valid_from a valid_to, jako je ukázáno na obr. 2.4.



Obrázek 2.4: Ukázka použití metadat valid_from/to [9]

2.2.3 Data Version Control

V tomto případě uživatelé nepřebírají odpovědnost za verzování dat, ale spoléhají se na tzv. Data Version Control nástroje, které jsou k tomu přímo určené. Jedním z nejvíce používaných verzovacích nástrojů je Git, jenže Git je navržen pro malá data. V případě obsáhlejších dat je nutné použít jiné nástroje. Proto byly vytvořeny nástroje jako je lakeFS, DVC a Git LFS, které umožňují práci i s obsáhlejšími soubory a daty [9].

Tyto nástroje řeší:

- minimalizaci potřebného místa pro verzování dat,
- zpřístupnění operací pro práci s verzemi dat,
- stejnou funkcionalitu pro strukturovaná a nestrukturovaná data,
- zálohu a obnovu dat.

2.3 Detekce duplicitních souborů

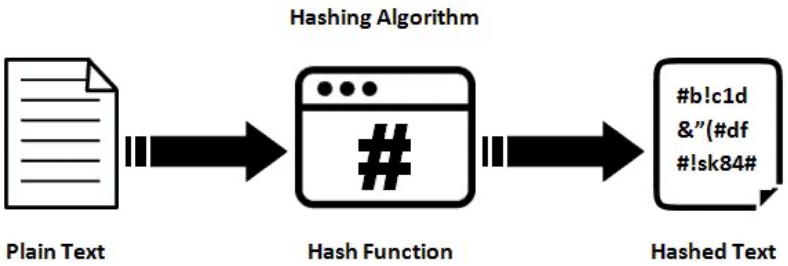
Efektivní správa dat je klíčovým prvkem pro optimalizaci využití úložného prostoru a minimalizaci redundance, což přispívá k celkové úspornosti a stabilitě informačních systémů. V dnešní době moderních systémů pro ukládání souborů a složitých algoritmů na porovnávání souborů se může zdát těžké detekovat duplicitní soubory, ale existuje řada způsobů, jak toho docílit. Mezi nejčastěji používané přístupy k detekci duplicitních souborů patří metody založené na:

- hashování,
- porovnávání obsahů,
- porovnávání metadat.

Volba vhodné metody pak závisí na typu a velikosti jednotlivých souborů, přičemž každá z nich má své vlastní výhody a omezení vzhledem k charakteristikám souborů a požadavkům na jejich detekci. V této podkapitole budou postupně představeny výše zmíněné metody.

2.3.1 Hashovací algoritmy

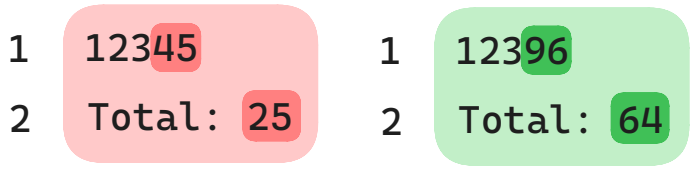
Tato metoda je založená na generování hashe pomocí matematické funkce pro každý soubor, který je porovnáván. Pro identifikaci duplicitních souborů je provedeno porovnání hashů jednotlivých souborů. Pokud jsou hashe stejné, pak je soubor považován za duplicitní [10]. Zde je důležité zohlednit velikost souborů, protože při zpracování velkých souborů se čas vytváření hashe může lišit v závislosti na zvoleném hashovacím algoritmu. Některé algoritmy mohou být pomalejší než ostatní, zatímco jiné mohou být rychlejší. Tento způsob je velmi často využíván v komerčních nástrojích pro detekci duplicitních souborů. Na obr. 2.5 lze vidět, jak se z běžného textu vytvoří zahashovaný text.



Obrázek 2.5: Ukázka hashovacího algoritmu [11]

2.3.2 Porovnání obsahů

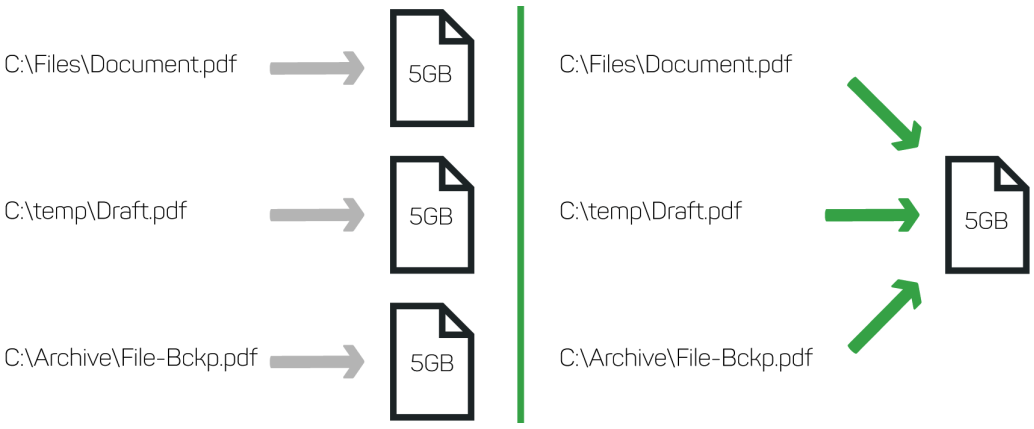
Tento způsob je založen na získání obsahu souborů a jejich následném porovnání. V případě identických obsahů lze prohlásit, že se jedná o duplicitní soubor. Ačkoliv se tato metoda zdá jednoduchá, má jednu zásadní nevýhodu: pro velké soubory je příliš pomalá [12]. S tímto způsobem je možné se setkat spíše ve verzovacích nástrojích jako je Git, ale je využíván i v příkazových řádcích operačních systémů pomocí příkazů `cmp` nebo `diff`. Na obr. 2.6 je možné vidět, jak jsou porovnávány jednotlivé části obsahu souborů.



Obrázek 2.6: Ukázka porovnání obsahu souborů

2.3.3 Porovnání metadat

Při použití této metody jsou porovnávány metadata souborů, jako je jejich velikost, datum vytvoření a datum poslední změny [13]. V případě shody lze prohlásit soubory za duplicitní. Výhodou tohoto způsobu je rychlost a jednoduchost. Nevýhodou však je, že ne vždy se podaří správně detekovat duplicity. Na obr. 2.7 je vidět příklad, kde jsou porovnávány různé soubory podle jejich velikosti. Všechny tři soubory mají stejnou velikost, a proto tato metoda označí soubory za duplicitní, i když ve skutečnosti nejsou.



Obrázek 2.7: Ukázka porovnání velikostí souborů [14]

2.4 Shrnutí

V této kapitole byl představen Document Management System spolu s problémy, které při jeho realizaci nastávají, jako je ukládání dat, způsoby verzování dat a detekce duplicit. Zároveň byly rozebrány i jednotlivé výhody a nevýhody možných řešení těchto problémů. Po získání základních znalostí o DMS je vhodné prozkoumat možnosti práce s tímto systémem, zejména využití REST API, což bude podrobněji vysvětleno v následující kapitole.

Využití REST API ve webových aplikacích

3

Tato kapitola slouží k náhledu do problematiky REST API a vytváření webových aplikací. Bude vysvětleno, jak je REST API dostupné a jak ho správně zabezpečit. Dále bude představeno několik systémů, technologií a nástrojů pro práci s REST API spolu s jejich klíčovými výhodami. Nakonec budou prozkoumány platformy, které jsou podporovány REST API a na co je třeba brát ohled u těchto platforem.

3.1 REST API

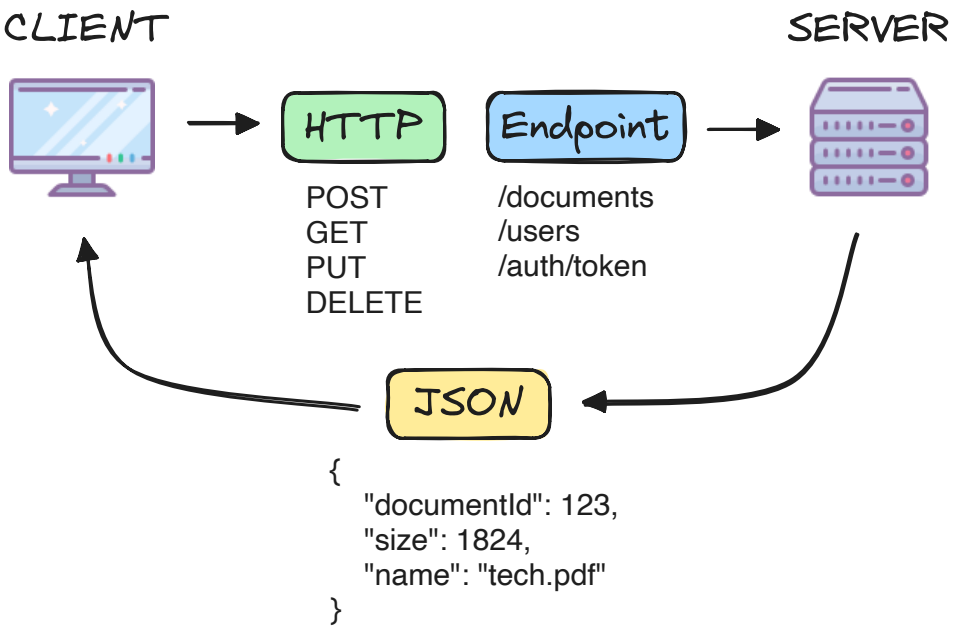
Mnoho aplikací v dnešní době musí komunikovat s dalšími aplikacemi, které mohou být jak interní, tak aplikace třetích stran, aby byly schopny vykonávat požadovanou činnost. Jedním ze způsobů, jak tuto komunikaci zařídit, je pomocí REST API.

REST (Representational State Transfer) je běžně používaná architektura pro webové služby. Rozhraní, která implementují architekturu REST by měla být bezestavová, což znamená, že jakékoliv změny v implementaci nesmí způsobit neočekávané selhání kódu uživatele API [15]. Dále by měla zajistit úplnou nezávislost mezi klientem a serverem. Měla by mít i jednotné rozhraní a možnost cachování zdrojů.

REST API je rozhraní dodržující principy architektury REST, které využívají počítače pro vzájemnou výměnu informací přes Internet. Komunikovat s REST API lze prostřednictvím HTTP požadavků, kterými typicky poskytuje standardní databázové operace CRUD: vytvoření dat (Create), čtení dat (Read), aktualizace dat (Update) a mazání dat (Delete). Odpovědi pak server posílá např. v JSON formátu. Toto rozhraní je často preferováno díky své jednoduchosti použití a snadnému využití ze strany klientů. Obr. 3.1 ukazuje, jak probíhá komunikace mezi klientem a serverem (REST API).

3.2 Dostupnost REST API

Pokud je požadována komunikace s REST API, je důležité, aby bylo zpřístupněno prostřednictvím serveru. Tento server slouží jako rozhraní mezi uživatelem a daty.



Obrázek 3.1: Ukázka komunikace s REST API

Pokud chce uživatel přistupovat k těmto datům, odešle na server požadavek, který obsahuje všechny potřebné informace pro manipulaci s daty. Poté, co server obdrží od uživatele požadavek, zpracuje jej podle definovaného REST API a odešle odpověď zpět uživateli. Tento server může být provozován organizací buď on-premise (interně), nebo v cloudu.

V případě, že se organizace rozhodne využít on-premise řešení, znamená to pro ni, že server bude hostován lokálně a že bude běžet na fyzickém zařízení, které je umístěno v prostorách dané organizace. Tím získávají plnou kontrolu nad těmito servery, což s sebou nese i zvýšené náklady na správu a údržbu [16]. Výhodou on-premise řešení je možnost plně přizpůsobit infrastrukturu specifickým potřebám organizace a zabezpečení dat podle vlastních standardů a předpisů. On-premise servery jsou často využívány velkými podniky či organizacemi, které mají specifické požadavky na zabezpečení či výkon.

Naopak při využití cloud serveru je server hostován v cloudu a je umístěn v prostorách poskytovatelů cloudu, kteří se zároveň starají o jejich správu a údržbu, což organizacím usnadňuje provoz. Přístup k těmto serverům je umožněn prostřednictvím webového prohlížeče či jiného uživatelského rozhraní [16]. Cloud servery umožňují organizacím snadnou škálovatelnost, flexibilitu a nižší náklady na provoz a údržbu ve srovnání s on-premise řešením. Kromě toho poskytují také možnost přístupu k datům a aplikacím odkudkoliv a kdykoliv, což zvyšuje efektivitu práce a mobilitu zaměstnanců.

3.3 Zabezpečení REST API

Zabezpečení REST API je kritickým aspektem vývoje moderních webových aplikací a služeb. Vzhledem k povaze těchto rozhraní, která umožňují přenos citlivých dat mezi klienty a servery, je nezbytné zajistit, aby tato data byla chráněna před neoprávněným přístupem, manipulací a zneužitím.

Napadené API může vést k řadě bezpečnostních hrozeb, včetně krádeže dat, neoprávněného přístupu a útoků typu odmítnutí služby (DoS) [17]. Útočník pak může využít zranitelnosti špatně zabezpečeného API k získání přístupu k citlivým informacím, manipulaci s daty nebo vložení škodlivého kódu do systému.

Úspěšný útok na nezabezpečené API může mít vážné následky, včetně krádeže dat, poškození pověsti, finančních ztrát a dokonce i porušení právních předpisů ohledně ochrany dat. Aby se předešlo těmto rizikům, je nezbytné implementovat spolehlivá bezpečnostní opatření v rámci REST API. Bezpečnostní prvky, jako je autentizace, autorizace, šifrování a ověřování vstupních dat, mohou pomoci minimalizovat rizika útoků na API a zajistit bezpečnou výměnu dat mezi systémy [18].

Bezpečnostní opatření:

- použití HTTPS – slouží k bezpečné komunikaci přes Internet a využívá SSL nebo TLS k šifrování přenášených dat mezi klientem a serverem [19],
- autentizace a autorizace – autentizace ověřuje identitu klienta, zatímco autorizace určuje, zda má klient povolení provést konkrétní akci [20]. Pomocí těchto mechanismů lze zajistit, že pouze autorizovaní uživatelé mají přístup ke zdrojům API,
- monitorování a logování – slouží k sledování provozu a aktivit v rámci API,
- silná hesla – hesla by měla být složitá a uložená bezpečným způsobem,
- validace vstupních hodnot – prevence proti SQL injection a cross-site scripting (XSS) útokům [21],
- omezení rychlosti (rate limit) – pomáhá předcházet útokům typu odmítnutí služby (DoS).

3.4 Systémy spjaté s dokumentací REST API

V této podkapitole budou představeny systémy, které jsou spjaty s dokumentací REST API. Na začátku bude probráno, jak se v dnešní době píše dokumentace REST API, a následně bude ukázán i běžný způsob zobrazení API, aby bylo pochopitelné i běžnými uživateli.

3.4.1 OpenAPI Specifikace

OpenAPI Specifikace definuje standardizovaný popis rozhraní pro HTTP API, který je nezávislý na programovacím jazyce. Vývojáři díky tomu mohou dodržovat předem danou strukturu při popisování struktury a chování jejich API. OpenAPI Specifikace dovoluje vývojářům definovat endpointy jejich API včetně parametrů a podporuje i pokročilé funkce, jako jsou například autentizace, validace dat a ošetřování chyb.

Jelikož se OpenAPI Specifikace píše v JSON nebo YAML formátu (viz zdrojový kód 3.1), je tato specifikace čitelná a srozumitelná jak lidmi, tak stroji. Díky tomu se při jejím psaní často používají nástroje pro automatické generování dokumentace daného API a nástroje pro automatické generování kódu serveru či klienta v několika různých programovacích jazycích.

Jedním z těchto nástrojů je OpenAPI Generator, který dovoluje, mimo generování kódu serveru a klienta, vygenerovat dokumentaci. Generování kódu provádí pomocí šablonovacího systému Mustache.

Při vývoji REST API je důležité, aby byla zachována konzistence mezi implementací a dokumentací API, což je díky těmto generátorům snadné dodržet [22]. Vývojářům tedy stačí, aby napsali OpenAPI Specifikaci a následně použili libovolný generátor pro vygenerování kódu, díky čemuž se vygeneruje kód, který této specifikaci odpovídá. V případě, že vývojáři již mají OpenAPI Specifikaci napsanou, je možné ji zobrazit například ve formě Swagger UI.

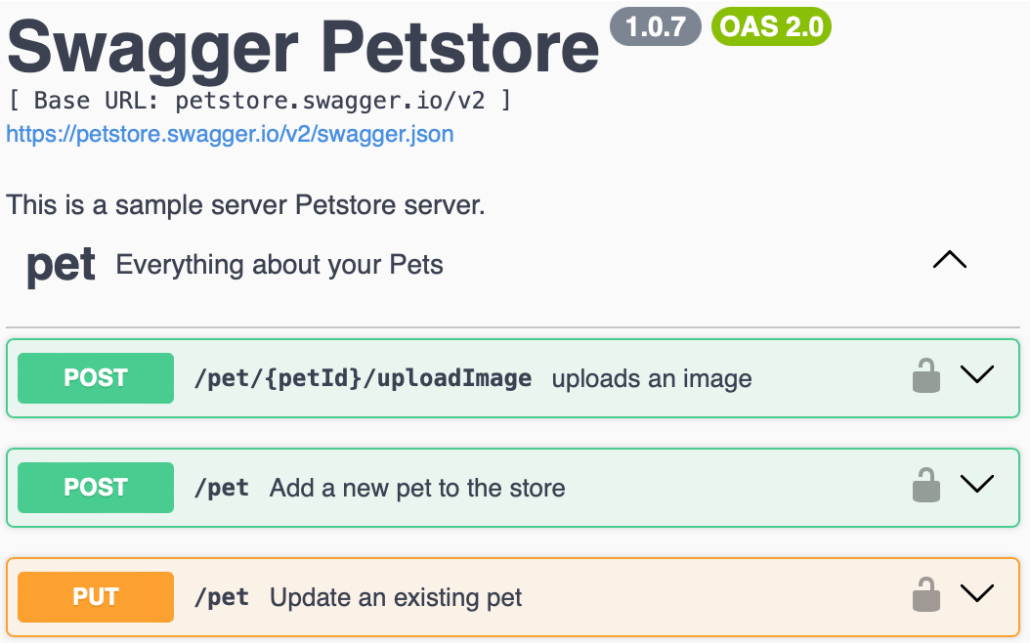
```yaml
openapi: "3.0.0"
paths:
  /pet/{petId}:
    get:
      description: Retrieves a pet.
      parameters:
        name: petId
        in: path
        required: true
        description: The pet ID.
        schema:
          type: integer
```

Zdrojový kód 3.1: Ukázka OpenAPI Specifikace v YAML formátu

3.4.2 Swagger UI

Swagger UI je kolekce HTML, JavaScript a CSS prvků, které dohromady tvoří vzhledově přehlednou dokumentaci z API popsaného pomocí OpenAPI Specifikace. Pomocí Swagger UI je jednoduché vizualizovat API a interagovat s ním bez nutnosti

znalosti jakékoliv implementační logiky [23]. Swagger UI bývá automaticky generován z OpenAPI Specifikace. Na obr. 3.2 lze vidět příklad vizualizace API pomocí Swagger UI.



Obrázek 3.2: Ukázka Swagger UI [24]

3.5 Technologie a nástroje pro práci s REST API

Tato podkapitola slouží k představení různých technologií a nástrojů, které lze využít, ať už pro vývoj či práci s REST API. Na začátku budou probrány technologie používané v různých programovacích jazycích, které umožňují snadné vytváření REST API. Dále pak budou představeny různé nástroje, které ulehčují práci s REST API a které dovolují jeho testování či další různé operace.

Konkrétně budou ukázány tyto technologie a nástroje:

- Java a Spring Boot – vývoj REST API,
- C# a ASP.NET Core – vývoj REST API,
- Apiary – dokumentace, testování a vývoj REST API,
- Postman – dokumentace, testování a vývoj REST API.

3.5.1 Java a Spring Boot

V případě vytváření webové aplikace v programovacím jazyce Java, je jedním z nejvíce oblíbených nástrojů, který to umožňuje, Spring Boot. Spring Boot je nadstavba nad Spring frameworkem, která usnadňuje vytváření webových aplikací a mikroslužeb využívajících REST API. Každá Spring Boot aplikace má v sobě zabudovaný webový server (nejčastěji Tomcat), takže se vývojáři nemusí starat o konfiguraci externího webového serveru. Další výhodou Spring Bootu je automatická konfigurace Spring frameworku a balíků třetích stran [25]. Spring Boot poskytuje tzv. starter dependencies, což jsou předem definované závislosti obsahující další potřebné závislosti. Tímto způsobem vývojáři nemusí ručně přidávat jednotlivé závislosti pro řešení specifických problémů. Stačí jim použít příslušnou starter dependency a všechny potřebné závislosti jsou automaticky začleněny do projektu. Zdrojový kód 3.2 znázorňuje příklad REST API napsaného v Javě a Spring Bootu.

Výhody:

- zabudovaný webový server,
- ulehčuje konfiguraci,
- poskytuje starter dependencies,
- možnost integrace s ostatními Java frameworky jako je například Hibernate.

```
@RestController
@RequestMapping("/documents")
public class DocumentController {
    private final DocumentService documentService;

    public DocumentController(
        DocumentService documentService
    ) {
        this.documentService = documentService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<Document> getDocument(
        @PathVariable Long id
    ) {
        Document document = documentService.getDocument(id);
        return ResponseEntity.ok(document);
    }
}
```

Zdrojový kód 3.2: Ukázka REST API v Javě a Spring Bootu

3.5.2 C# a ASP.NET Core

Pokud je potřeba vytvořit webovou službu založenou na REST API v programovacím jazyce C#, jedním z nejvhodnějších nástrojů je technologie ASP.NET Core. Tento open-source framework od společnosti Microsoft je určen pro vývoj webových aplikací a služeb. S ASP.NET Core je snadné vytvářet služby, které osloví širokou škálu klientů, včetně prohlížečů a mobilních zařízení [26]. Umožňuje i tvorbu webového uživatelského rozhraní, což znamená, že je možné vytvářet frontend i backend aplikace pomocí jedné jediné technologie. Zdrojový kód 3.3 znázorňuje příklad REST API napsaného v C# a ASP.NET Core.

Výhody:

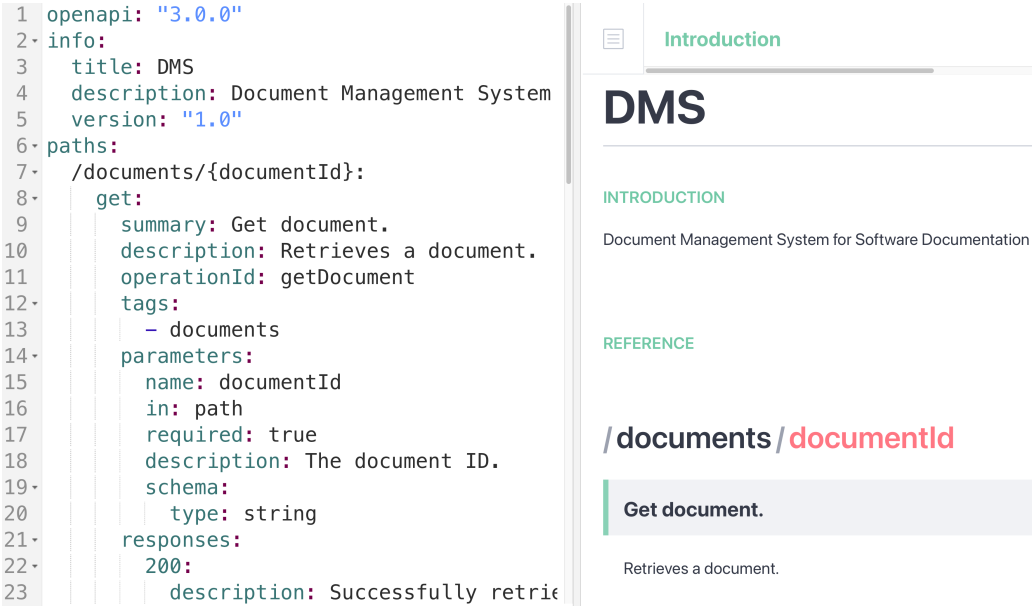
- automatická serializace tříd do správně naformátovaného JSONu,
- autentizace a autorizace – zabudovaná podpora pro JSON Web Token,
- podpora HTTPS – automatické vygenerování testovacího certifikátu, který lze použít pro povolení lokálního HTTPS,
- možnost integrace s moderními frameworky jako jsou například AngularJS a ReactJS,
- možnost napojení se na cloudové služby Microsoft Azure.

```
[Route("documents")]
[ApiController]
public class DocumentController : ControllerBase {
    private readonly DocumentService documentService;

    public DocumentController(
        DocumentService documentService
    ) {
        this.documentService = documentService;
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetDocument(Guid id) {
        Document document =
            await documentService.GetDocument(id);
        return Ok(document);
    }
}
```

Zdrojový kód 3.3: Ukázka REST API v C# a ASP.NET Core

3.5.3 Apiary

Apiary je nástroj, který ulehčuje a urychluje vývoj API. Má několik různých vlastností, které pokrývají většinu problémů, se kterými se lze při vývoji API setkat. Jednou z nich je například dokumentační editor, ve kterém je možné nadefinovat API a sdílet ho s dalšími vývojáři či uživateli [27]. Dokumentace je v tomto nástroji zobrazena v přehledné a srozumitelné formě (viz obr. 3.3), čímž se tento nástroj řadí mezi uživatelsky přívětivé. Apiary poskytuje i mock serveru, který je zprostředkován společností Oracle a který zajišťuje možnost interakce s API bez nutnosti zařizování si vlastního serveru. Výhodou tohoto nástroje je i možnost testování API dodáním dokumentace a vytvořením očekávaných výstupních hodnot. Každý test, který bude spuštěn, provede jeden z požadavků a zkontroluje, zda skutečné výstupní hodnoty odpovídají očekávaným hodnotám.



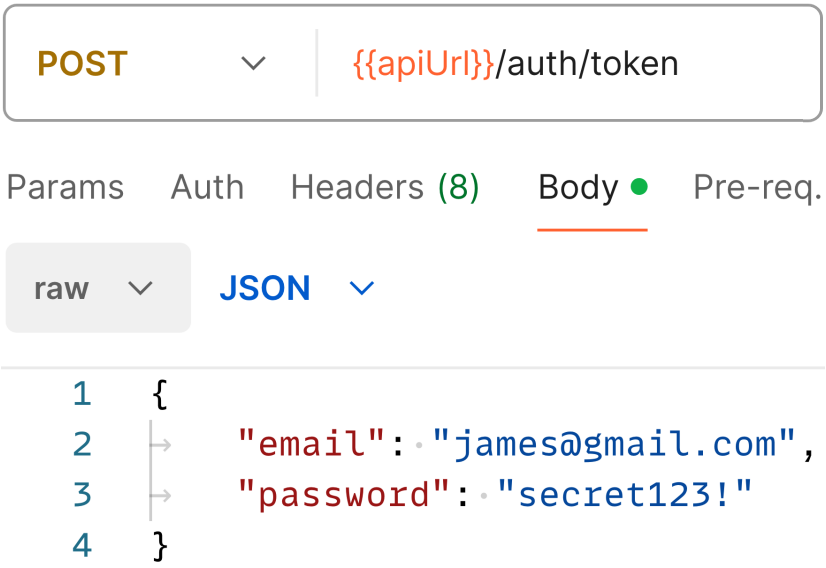
Obrázek 3.3: Ukázka Apiary

3.5.4 Postman

Asi nejznámějším a nejvíce používaným nástrojem pro práci s REST API je Postman. Jedná se o nástroj, který podporuje vývoj, testování, psaní dokumentace, sdílení a používání API bez nutnosti znalosti programování. Vývojáři ho často využívají k zjednodušení procesu testování API díky uživatelsky přívětivému rozhraní, které umožňuje vytvářet požadavky, zobrazovat odpovědi a ladit problémy [28]. Díky svým funkcím usnadňuje a urychluje vývoj a správu API. Na obr. 3.4 je ukázka požadavku v nástroji Postman.

Výhody:

- využívání kolekcí – seskupují požadavky a umožňují hierarchicky uspořádat požadavky do adresářů,
- podpora pro různé typy API – REST, SOAP a GraphQL,
- podpora CI/CD,
- možnost spolupráce více vývojářů,
- automatizované testy.



Obrázek 3.4: Ukázka požadavku v nástroji Postman

3.6 Podpora REST API na různých platformách

Při vytváření služby, která má být používána různými uživateli, je důležité brát v úvahu různé typy připojení k této službě. Je pravděpodobné, že uživatelé budou využívat různé platformy, jako je web, desktop nebo mobilní zařízení. V kontextu REST API služeb je snadno zařízena podpora všech těchto platforem z důvodu, že daná služba je zprostředkována pomocí HTTP požadavků, které tyto platformy podporují [29]. V podstatě se dá říci, že vývojáři REST API služeb se o podporu více platforem nemusí starat, protože je automaticky zařízena.

Existuje však jedno zásadní omezení, které může být ovlivněno různými platformami, a to je rozdílná rychlost internetového připojení mezi mobilními zařízeními a počítači. Uživatelé, kteří používají počítače, obvykle disponují rychlejším internetovým připojením, které umožňuje současně přenášet a přijímat větší objem dat. Naopak uživatelé mobilních zařízení mohou mít pomalejší připojení, což omezuje jejich schopnost přenášet a přijímat data.

3.7 Shrnutí

V této kapitole bylo vysvětleno co je to REST API, jak je dostupné a proč je potřeba ho zabezpečit proti různým útokům. Dále bylo ukázáno, jaké systémy spjaté s psaním dokumentace API existují a proč je výhodné je používat. Dále byly vysvětleny různé technologie a nástroje, které vývojáři mohou použít pro snadný vývoj REST API. Nakonec bylo vysvětleno, proč je REST API podporováno na všech platformách a jaké omezení existuje na mobilních zařízeních. Nyní budou představeny další technologie, které lze použít pro vývoj DMS využívajícího REST API.

Použité technologie 4

Tato kapitola slouží jako doplnění technologií, které ještě nebyly zmíněny a které lze při řešení práce využít. Bude probrán Spring framework a jeho jednotlivé nadstavby, které ulehčují specifickou oblast vývoje webových aplikací. Dále pak bude představen nástroj, pomocí kterého lze snadno vytvářet a upravovat databáze nezávisle na tom, jaká databáze je využívána. Nakonec bude ukázán způsob, pomocí kterého se ověřují uživatelé a který zajišťuje bezpečnou autentizaci a autorizaci.

4.1 Spring

Spring je populární open-source framework, který umožňuje snadné vytváření enterprise aplikací v programovacím jazyce Java [30]. Spring poskytuje funkcionalitu Dependency Injection, která umožňuje objektům definovat jejich závislosti, které jsou do nich následně vkládány pomocí Spring kontejneru. Spring kontejner je založen na návrhovém vzoru Inversion of Control a jedná se o způsob, kdy se vývojáři nemusí starat o to, kdy a jak jsou jejich objekty vytvářeny, protože se o to postará Spring kontejner [31]. Spring se stará o konfiguraci aplikace, aby se vývojáři mohli zaměřit na aplikační logiku. Název Spring obvykle označuje buď samotný framework a nebo celou skupinu projektů, jenž nad tímto frameworkem staví a přidávají k němu další funkcionality.

4.1.1 Spring Data JPA

Spring Data JPA je nadstavba nad Spring frameworkem, která výrazně ulehčuje implementaci vrstvy, která přistupuje k datům. Java Persistence API (JPA) se stará o tzv. object-relational mapping (ORM), což je způsob, jak tabulkám databáze přiřadit Java objekty. Každému záznamu v tabulce databáze odpovídá právě jeden objekt a vývojáři tak nemusí pracovat s tabulkou jako takovou, ale mohou využívat tyto objekty. Spring Data JPA redukuje množství potřebného kódu, který je potřeba pro práci s JPA. Jedním z příkladů je generování dotazů na základě názvů metod [32].

4.1.2 Spring Security

Spring Security je framework navržený pro integrování autentizace a autorizace do aplikací postavených na platformě Spring [33]. Kromě toho, že umožňuje implementaci těchto bezpečnostních funkcí, poskytuje ochranu aplikace před různými typy útoků, jako jsou například CSRF (Cross-Site Request Forgery) a XSS (Cross-Site Scripting). Tento framework podporuje i standard OAuth2, což umožňuje aplikacím integrovat autentizaci a autorizaci s třetími stranami, jako jsou sociální sítě nebo další služby.

Základní koncepty:

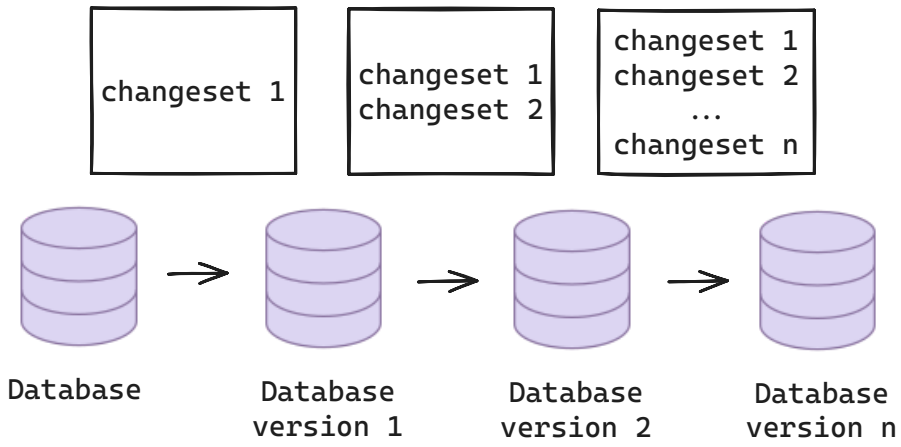
- autentizace – proces ověření, zda uživatel je opravdu ten, za koho se vydává,
- autorizace – proces ověření, zda má uživatel na danou operaci právo,
- ochrana před útoky – CSRF a XSS,
- principal – objekt, který obsahuje informace o aktuálně autentizovaném uživateli,
- filtry – každý příchozí požadavek nejprve prochází řadou bezpečnostních filtrů a až následně je akce vykonána,
- autority a role – přístupová práva,
- správa relací – správa a kontrola relací uživatelů.

4.2 Liquibase

Liquibase je open-source nástroj pro správu změn schématu databáze, který umožňuje snadno spravovat revize změn v databázi. Umožňuje nasazování a vracení změn pro konkrétní verze databáze. Dá se vlastně říci, že Liquibase je něco jako Git pro databáze.

Při práci s Liquibase se vývojáři setkají se 2 pojmy: changeset a changelog. Changeset představuje jednu změnu, kterou Liquibase provede v databázi a je unikátně označen pomocí atributů: author, id a changelog-file. Changelog je soubor, který obsahuje seznam jednotlivých changesetů, tedy seznam změn v databázi [34]. Liquibase používá tento changelog pro audit databáze a v případě, že některé změny (changesety) v changelogu ještě nebyly provedeny, provede je.

Liquibase podporuje desítky databází a díky tomu je možné vytvářet changelogy, které budou fungovat na vícero databázích. Na obr. 4.1 lze vidět, jak s jednotlivými changesety vznikají nové verze databáze.



Obrázek 4.1: Ukázka změn databáze pomocí changesetů

4.3 JSON Web Token

JSON Web Token (JWT) je standard, který definuje kompaktní a soběstačný způsob pro bezpečný přenos informací mezi jednotlivými stranami ve formě JSON objektu [35]. Přenášená informace je důvěryhodná, protože je digitálně podepsána, což umožňuje její ověření.

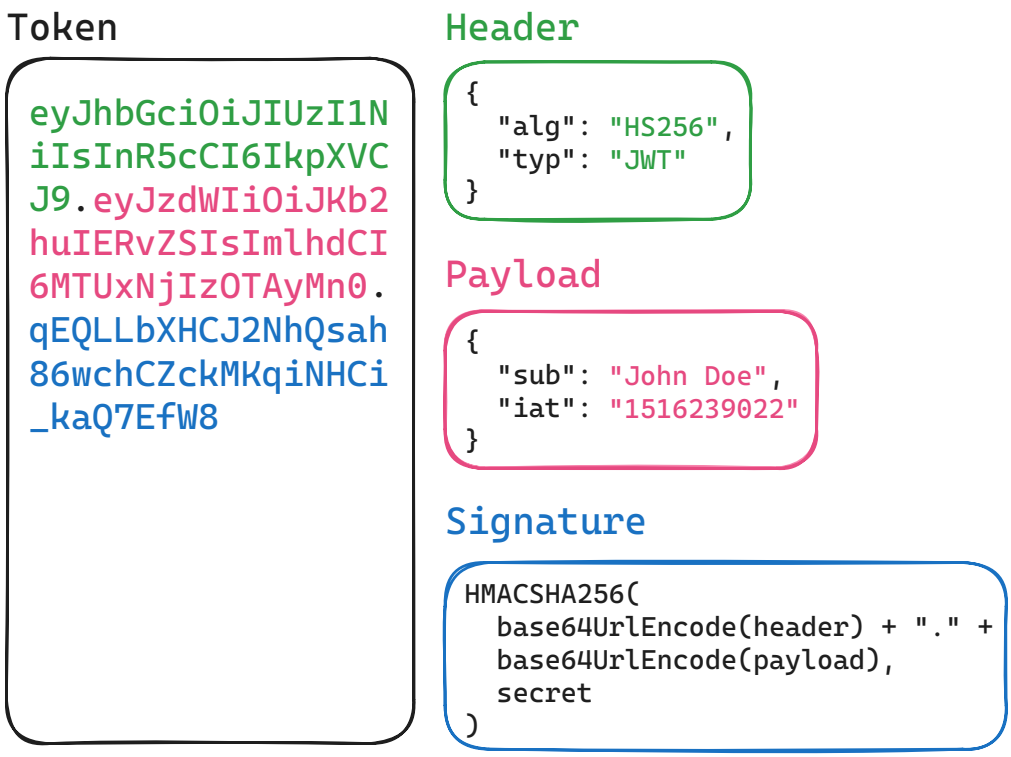
JWT token může být podepsán použitím sdíleného tajného klíče (pomocí HMAC algoritmu) nebo použitím dvojice klíčů veřejný/soukromý klíč (pomocí RSA algoritmu). V případě podepsání tokenu pomocí dvojice veřejný/soukromý klíč zaručuje podpis, že pouze strana, která vlastní soukromý klíč, je ta, která token podepsala.

Tento token také zvyšuje bezpečnost, protože uživatel nemusí v každém požadavku dodávat citlivé údaje jako je jméno a heslo, ale může dodat pouze JWT token, který mu byl vygenerován po úspěšné autentizaci [36].

4.3.1 Struktura JSON Web Tokenu

JWT se skládá ze tří částí (viz obr. 4.2), oddělených tečkou:

- hlavička (header) – typicky obsahuje 2 části: algoritmus pro podpis tokenu a typ tokenu,
- data (payload) – obsahuje jednotlivá tvrzení (claims), což jsou informace o uživateli a různá další data,
- podpis (signature) – slouží k ověření pravosti tokenu. Pro jeho získání se vezme zakódovaná hlavička, zakódovaná data, klíč a vše se podepíše algoritmem uvedeným v hlavičce.



Obrázek 4.2: Ukázka JSON Web Tokenu

4.4 Shrnutí

V této kapitole byly ukázány další technologie, které lze využít při vývoji DMS. Byly probrány moduly Spring frameworku, které je vhodné využít pro zajištění bezpečnosti REST API a k usnadnění práce s databází. Dále byl ukázán nástroj Liquibase, pomocí kterého lze zajistit vytváření databáze, která je nezávislá na konkrétním typu. Nakonec byl ukázán JWT, který lze použít k ověřování uživatelů, kteří chtějí s REST API komunikovat.

Návrh aplikace 5

Tato kapitola obsahuje návrh úložiště, REST API a logování. V sekci REST API jsou zahrnuty i návrhy endpointů pro: práci s uživateli, autentizaci a autorizaci uživatelů, správu dokumentů a správu revizí dokumentů. V návrhu jsou zmíněny jednotlivé technologie, nástroje a algoritmy, které byly představeny v předchozích kapitolách.

5.1 Úložiště

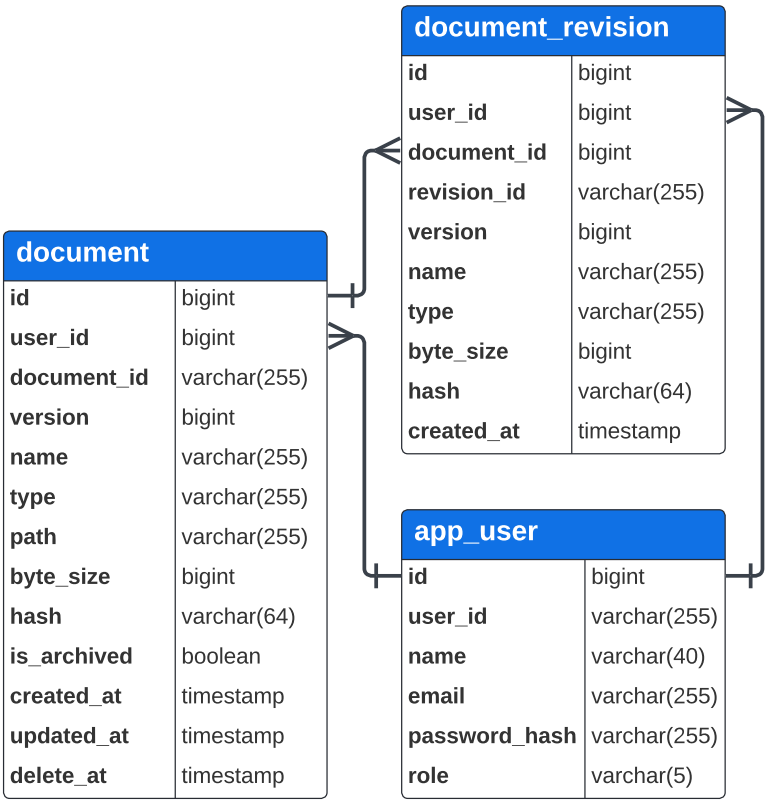
Jak už bylo dříve zmíněno, vzniklé dokumenty při vývoji softwaru je třeba někam ukládat. Pro ukládání dokumentů bude použit hybridní způsob, využívající lokální úložiště, který byl popsán v kapitole 2.1.4. Důvodem pro to je fakt, že díky tomu bude v budoucnu jednoduché zaměnit, v případě potřeby, lokální úložiště za cloudové.

Jelikož se bude používat databáze pro ukládání údajů uživatelů či metadat dokumentů, je nutné vytvořit databázový model. V praxi se často využívají různé databáze a proto bude pomocí Liquibase (viz kapitola 4.2) zajištěna podpora databází, jako jsou H2, PostgreSQL, Oracle a MS SQL. Databázi, která bude ve výsledku použita, bude možno nastavit v konfiguračním souboru.

Každý soubor bude do systému nahrán některým z vývojářů, a proto bude databáze obsahovat i tabulku uživatelů s tím, že jejich hesla budou hashována pomocí BCryptu. Dokumenty bude možné verzovat, takže spolu s tabulkou dokumentů se v databázi bude nacházet i tabulka revizí, jejíž záznamy budou reprezentovat jednotlivé verze dokumentů. Ty budou vytvářeny pomocí úplné duplikace dat (viz kapitola 2.2.1), ale jelikož v databázi budou uloženy pouze metadata dokumentů, tak budou duplikovány pouze tyto metadata a nikoliv soubory jako takové. Současně budou duplikována pouze vybraná metadata, protože ne všechna metadata bude potřeba verzovat. Důvodem pro použití úplné duplikace dat je jednoduchost této metody. Výsledný databázový model je znázorněn na obr. 5.1.

Nakonec je potřeba zajistit, jak budou detekovány duplicity dokumentů pro úsporné ukládání v úložišti. Ty budou detekovány pomocí hashovacího algoritmu popsaného v kapitole 2.3.1. Tento způsob je vhodný z důvodu, že v databázi bude

potřeba ukládat i ukazatel (místo na disku) na daný dokument. To lze zařídit pomocí hashe, protože v aplikaci bude umožněno zvolit si adresář na disku, do kterého budou dokumenty ukládány a hash pak bude sloužit jako cesta k souboru v tomto adresáři. Namísto toho, aby byly všechny dokumenty ukládány v jednom adresáři, budou např. první dva znaky hashe použity jako název podadresáře, ve kterém se daný dokument bude nacházet. Daný hashovací algoritmus si bude možné zvolit v konfiguračním souboru aplikace.



Obrázek 5.1: Databázový model

5.2 REST API

REST API bude implementováno za použití technologií Java a Spring Boot (viz kapitola 3.5.1), a to z důvodu přehledné dokumentace a jednoduché konfigurace. Toto API bude popsáno pomocí OpenAPI Specifikace (viz kapitola 3.4.1), aby z ní následně bylo možné vygenerovat kód serveru, čímž se zaručí pravdivost specifikace vůči implementaci. Pomocí Swagger UI (viz kapitola 3.4.2) pak bude možné si zobrazit specifikaci API v přehledné formě a zjistit, jak s ním komunikovat. REST API

bude provozováno na on-premise serveru (viz kapitola 3.2) z důvodu zajištění úplné kontroly nad serverovým prostředím.

Aby bylo možné Document Management System používat a pracovat s ním, budou implementovány různé endpointy, které umožní manipulaci s uživateli, včetně procesů autentizace a autorizace, a také práci s dokumenty a jejich revizemi. Níže budou vysvětleny jednotlivé endpointy podle jejich účelu.

5.2.1 Uživatelé

Jelikož DMS bude moci využívat více uživatelů zároveň, tak bude potřeba zajistit možnost vytvoření uživatelů. Uživatel se bude registrovat do systému pomocí jména, emailu a hesla, načež bude v systému vytvořen nový uživatel s rolí `USER`. Pro zajištění jednoznačné identifikace uživatele bude každý uživatel mít unikátní e-mailovou adresu. Aby se zvýšila bezpečnost, tak pro heslo bude vyžadováno, aby bylo dlouhé (např. minimálně 8 znaků) a aby obsahovalo číslo a speciální znak.

Z důvodu, že uživatel může zapomenout svoje heslo, tak v systému bude existovat uživatel s rolí `ADMIN`, který jako jediný bude moci provádět změnu hesla uživatelů dodáním emailu daného uživatele a nového hesla. Admin bude v systému automaticky vytvořen při spuštění aplikace a jeho údaje bude možné nastavit v konfiguračním souboru. Vytvoření (registrace) uživatele bude prvním krokem pro používání DMS. Dalším krokem bude autentizace, která bude představena v další sekci. V tab. 5.1 jsou ukázány jednotlivé endpointy pro práci s uživateli.

HTTP Metoda	Endpoint	Popis
POST	/users	Registrace uživatele.
GET	/users/me	Získání informací o právě autentizovaném uživateli.
PUT	/users/password	Změna hesla uživatele.

Tabulka 5.1: Endpointy pro práci s uživateli

5.2.2 Autentizace a autorizace uživatelů

Po úspěšné registraci se uživatel bude muset autentizovat, což je nezbytným krokem pro další práci s dokumenty v DMS. Autentizace uživatele tedy představuje druhý a zároveň poslední nutný krok pro plné využívání funkcionalit DMS.

Proces autentizace bude probíhat prostřednictvím ověření e-mailové adresy a hesla, které uživatel zadával při registraci. Po úspěšné autentizaci bude uživateli vrácen JSON Web Token (viz kapitola 4.3). Tento token bude uživatel dodávat v hlavičce každého následujícího požadavku, čímž se bude autorizovat. JWT token byl

vybrán pro autentizaci a autorizaci uživatelů z důvodu zvýšené bezpečnosti a snadné integrace do hlavičky požadavku.

Pomocí tohoto tokenu se uživatel bude moci dostat pouze ke svým dokumentům, protože při vývoji softwarového projektu se o dokumenty bude starat vždy pouze jeden uživatel. Tím bude zajištěna i bezpečnost, protože v případě, že by nějaký škůdce nahrál do systému soubor s virem, tak se k němu ostatní uživatelé nedostanou. V tab. 5.2 je ukázán endpoint pro autentizaci uživatele.

HTTP Metoda	Endpoint	Popis
POST	/auth/token	Autentizace uživatele.

Tabulka 5.2: Endpoint pro autentizaci uživatele

5.2.3 Dokumenty

Nad dokumenty budou poskytovány základní CRUD operace spolu s možností přepnutí na určitou revizi nebo archivací daného dokumentu.

Po úspěšném nahrání dokumentu do systému bude uživateli umožněno tento dokument stáhnout či získat jeho metadata. Dále bude možné nahrát novou verzi již existujícího dokumentu, čímž se vytvoří nová revize. Mezi jednotlivými revizemi dokumentu se pak bude moci přepínat. S ohledem na možnost ukládání velkého množství dokumentů bude zajištěno stránkování, aby nedocházelo k načítání všech dokumentů najednou. Uložené dokumenty pak bude možné smazat, čímž se smažou i všechny revize daného dokumentu.

Dokumenty bude možno řadit jak vzestupně, tak sestupně, a to podle několika metadat (sloupců), jako jsou: id, název, typ, velikost, verze apod. Spolu s řazením bude umožněno i filtrování dokumentů podle metadat (sloupců), jako jsou: název, typ, umístění či jestli je dokument archivován.

Archivace dokumentů bude realizována nastavením příznaku v databázi, který označuje, zda byl dokument archivován. Současně s tím bude nastaveno časové razítko, které určí, kdy má být dokument odstraněn z úložiště. Někdy se může stát, že uživatel změní názor ohledně archivace dokumentu a rozhodne se ji zrušit. Proto bude možné obnovit již archivovaný dokument, což znamená pouze změnu příznaku archivace a resetování časového razítka určujícího dobu odstranění dokumentu.

Dokumenty také půjde logicky přesouvat, což znamená, že každý dokument bude mít přiřazený určitý virtuální adresář, ve kterém se nachází, aby uživatel mohl hierarchicky řadit své dokumenty. V tab. 5.3 jsou ukázány jednotlivé endpointy pro práci s dokumenty.

HTTP Metoda	Endpoint	Popis
GET	/documents	Získání metadat dokumentů.
POST	/documents/upload	Nahrání dokumentu.
GET	/documents/{documentId}	Získání metadat dokumentu.
PUT	/documents/{documentId}	Nahrání nové verze dokumentu.
DELETE	/documents/{documentId}	Smazání dokumentu včetně jeho revizí.
GET	/documents/{documentId}/download	Stažení dokumentu.
GET	/documents/{documentId}/revisions	Získání revizí dokumentu.
PUT	/documents/{documentId}/revisions/{revisionId}	Přepnutí dokumentu na revizi.
PUT	/documents/{documentId}/move	Přesun dokumentu.
PUT	/documents/{documentId}/archive	Archivace dokumentu.
PUT	/documents/{documentId}/restore	Obnovení dokumentu z archivu.

Tabulka 5.3: Endpointy pro práci s dokumenty

5.2.4 Revize dokumentů

Podobně jako u samotných dokumentů, i u jejich revizí budou zavedeny operace umožňující jejich stažení či získání metadat. S ohledem na to, že každý dokument může mít několik revizí, bude zavedena i možnost jejich získávání pomocí stránkování. Vzhledem k tomu, že revize představují jednotlivé verze dokumentu, které nemusí být vždy relevantní, bude umožněno i jejich mazání. Stejně jako dokumenty, i revize bude možné řadit a filtrovat podle vybraných metadat. V tab. 5.4 jsou ukázány jednotlivé endpointy pro práci s revizemi dokumentů.

HTTP Metoda	Endpoint	Popis
GET	/revisions	Získání metadat revizí.
GET	/revisions/{revisionId}	Získání metadat revize.
DELETE	/revisions/{revisionId}	Smazání revize.
GET	/revisions/{revisionId}/download	Stažení revize.

Tabulka 5.4: Endpointy pro práci s revizemi dokumentů

5.3 Logování

Z důvodu, že REST API bude zpracovávat několik požadavků, je zapotřebí zajistit, aby tyto informace byly zaznamenávány pro následné monitorování běhu aplikace. To bude zařízeno pomocí logování tak, že bude vytvořeno několik souborů, do kterých se postupně budou zapisovat logy a po jejich naplnění se začnou přepisovat. Logovat se budou všechny příchozí požadavky, chyby a další důležité informace potřebné ke kontrole správného běhu aplikace.

Vzhledem k tomu, že REST API bude provozováno nepřetržitě, bude nutné zajistit možnost změny úrovně logování za běhu. Tuto možnost bude mít k dispozici pouze admin, který bude oprávněn měnit úrovně logování.

Implementace 6

V této kapitole bude podrobně popsán postup implementace výsledné aplikace a bude zde rovněž uvedeno i několik ukázek zdrojových kódů. Budou zde detailně představeny klíčové prvky aplikace, jako je úložiště, databáze, REST API, zabezpečení REST API a logování. Tato kapitola vychází z předchozí analýzy a návrhu aplikace.

6.1 Úložiště

Dokumenty jsou ukládány do lokálního úložiště, konkrétně do zvoleného adresáře na disku. Možnost volby adresáře je poskytnuta prostřednictvím konfiguračního souboru `application.yaml`, kde se nachází klíč `storage.path`, jehož hodnota udává adresář úložiště. V případě, že zvolený adresář neexistuje, bude automaticky vytvořen.

Aby se dokumenty neukládaly pouze do hlavního adresáře, vytvářejí se podadresáře s názvem odpovídajícím délce prefixu hashe dokumentu. Délka prefixu se nastavuje v konfiguračním souboru pomocí klíče `storage.subdirectory-prefix-length`. Tato hodnota je validována a její hodnota se musí vyskytovat v rozmezí 1–10, jiné hodnoty povoleny nejsou.

V úložišti jsou uchovávány pouze BLOBy dokumentů a jejich názvy jsou vytvářeny pomocí hashovacího algoritmu definovaného v `hash.algorithm`. Ve zdrojovém kódu 6.1 je zobrazen příklad konfigurace úložiště v konfiguračním souboru.

```
storage:
  path: /Users/kuba/Downloads/blob_storage
  subdirectory-prefix-length: 2

hash:
  algorithm: SHA-256
```

Zdrojový kód 6.1: Příklad konfigurace úložiště

6.2 Databáze

Databáze byla vytvořena pomocí Liquibase, díky čemuž jsou podporovány tyto databáze: **H2**, **PostgreSQL**, **Oracle** a **MS SQL**. Vytvořená databáze obsahuje 3 tabulky: `app_user`, `document`, `document_revision`. Ve zdrojovém kódu A.1 lze vidět způsob, kterým byla pomocí Liquibase vytvořena tabulka uživatelů. Při vytváření této tabulky nebylo možné použít slova `user` a `password`, protože se jedná o vyhrazená slova, ať už v PostgreSQL či Oracle databázi. Řešením bylo zaměnit `user` za `app_user` a `password` za `password_hash`. Do tabulky `document` a `document_revision` jsou ukládány pouze metadata a hashe dokumentů, protože BLOBy jsou ukládány do lokálního úložiště, jak bylo popsáno v předchozí kapitole.

Každá z výše uvedených tabulek má vytvořenou třídu (viz zdrojový kód B.1) pomocí Spring Data JPA, označenou anotací `@Entity`, která umožňuje mapování mezi příslušnou tabulkou a objektem v Javě. Spolu s tím má i každá tabulka vytvořený repozitář (rozhraní označené `@Repository`, viz zdrojový kód 6.2), který umožňuje provádění dotazů v databázi pomocí metod.

Databázi, která bude v aplikaci použita, je možné si zvolit v konfiguračním souboru `application.yaml`, konkrétně hodnotou klíče `spring.profiles.active`. Hodnota tohoto klíče může nabývat pouze těchto hodnot: `h2`, `postgresql`, `oracle` či `mssql`. Pro správnou funkčnost databáze je následně nutné změnit v konfiguračním souboru hodnoty `spring.datasource.url`, `spring.datasource.username` a `spring.datasource.password` u vybrané databáze (viz zdrojový kód 6.3).

```
@Repository
public interface UserRepository extends JpaRepository<User,
    Long> {
     Optional<User> findByEmail(String email);
     boolean existsByEmail(String email);
}
```

Zdrojový kód 6.2: Ukázka repozitáře uživatelů

```
spring:
  config:
    activate:
      on-profile: postgresql
  datasource:
    url: jdbc:postgresql://localhost:5432/dms
    driver-class-name: org.postgresql.Driver
    username: postgres
    password: postgres
```

Zdrojový kód 6.3: Příklad konfigurace PostgreSQL databáze

6.3 REST API

Navržené REST API bylo implementováno v technologiích Java verze 17 a Spring Boot. Nejdříve byla napsána dokumentace API pomocí OpenAPI Specifikace (viz další kapitola) a následně z ní byl vygenerován kód serveru za použití nástroje OpenAPI Generator, integrovaného jako plugin v Mavenu. Díky tomu je zaručena pravdivost specifikace vůči implementaci a zároveň je minimalizováno riziko rozporů mezi dokumentací a skutečným chováním API.

Generování kódu serveru bylo spuštěno pomocí příkazu `mvn compile`, který vytvořil jednotlivá rozhraní s koncovkami `Api` v adresáři `target/generated-sources/openapi`. Tato rozhraní byla následně implementována ve třídách balíku `controller`, označených jako `@RestController`, kde jsou definovány obslužné metody pro jednotlivé endpointy API. Ve zdrojovém kódu 6.4 je zobrazen controller pro endpointy dokumentů. Všechny endpointy REST API si lze vyzkoušet pomocí vyexportované kolekce z Postmana, která se nachází v souboru `DMS.json`. Dále budou popsány klíčové aspekty REST API, jako je dokumentace, stránkování, řazení a filtrování dat, a nakonec ošetření výjimek.

```
@RestController
@RequiredArgsConstructor
public class DocumentController implements DocumentsApi {
    private final DocumentService documentService;

    @Override
    public ResponseEntity<DocumentDTO> getDocument(
        String documentId
    ) {
        Document document =
            documentService.getDocument(documentId);
        DocumentDTO documentDTO =
            DocumentDTOMapper.map(document);

        return ResponseEntity.ok(documentDTO);
    }
}
```

Zdrojový kód 6.4: Ukázka controlleru pro endpointy dokumentů

6.3.1 Dokumentace

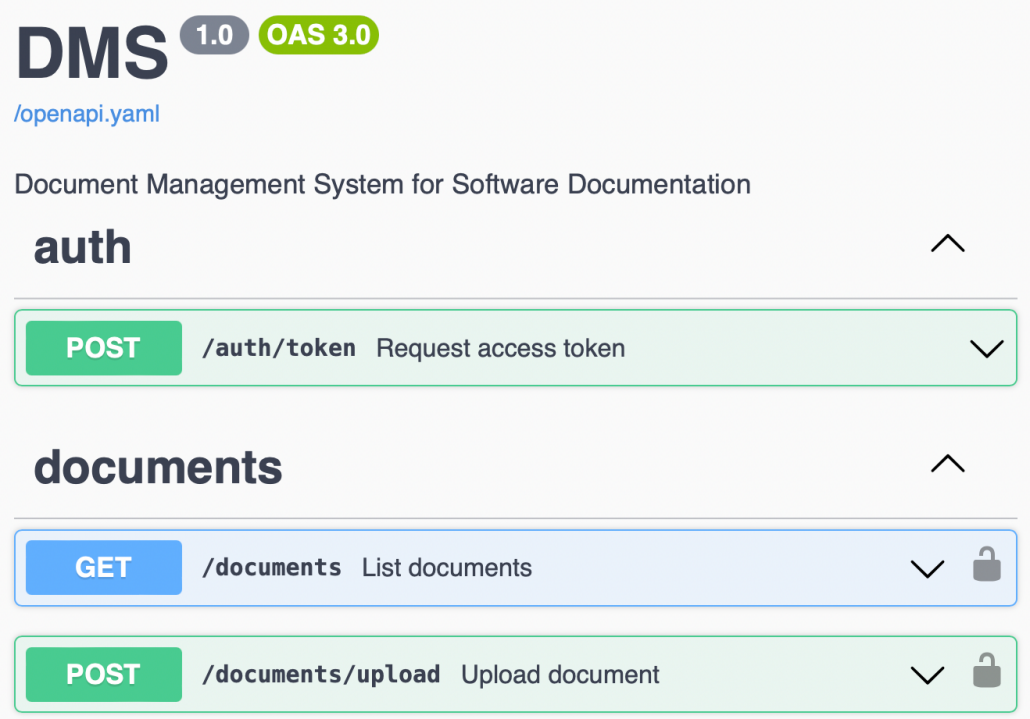
Dokumentace REST API byla napsána pomocí OpenAPI Specifikace v YAML formátu. V ní byly podrobně popsány všechny endpointy, včetně jejich parametrů, požadovaných dat a HTTP statusů. Ačkoliv se tato dokumentace obvykle píše v jednom

velkém YAML souboru, pro zlepšení přehlednosti a navigace byl zvolen přístup založený na vytvoření několika menších YAML souborů, kde každý z nich představuje jednu samostatnou část API.

Ve zdrojovém kódu 6.5 je znázorněn příklad `/documents/{documentId}` endpointu popsaného pomocí OpenAPI Specifikace. Pro lepší přehlednost dokumentace API byla pomocí knihovny springdoc přidána podpora pro Swagger UI (viz obr. 6.1).

```yaml
openapi: "3.0.0"
  paths:
    /documents/{documentId}:
      get:
        description: Retrieves a document.
        operationId: getDocument
        parameters:
          name: documentId
          in: path
          required: true
          description: The document ID.
          schema:
            type: string
```

Zdrojový kód 6.5: Ukázka popisu endpointu v OpenAPI Specifikaci



Obrázek 6.1: Ukázka dokumentace REST API ve Swagger UI

6.3.2 Stránkování dat

Vzhledem k tomu, že v systému bude uloženo několik dokumentů a jejich revizí, bylo implementováno stránkování pro jejich rychlejší získávání. Tímto způsobem je uživatel schopen si vybrat, která stránka má být načtena, a současně je také možné nastavit maximální počet získaných dat.

V jednotlivých endpointech, které stránkování podporují, je stránkování zajištěno pomocí parametrů požadavku: `page` a `limit`. Hodnota parametru `page` je omezená zdola nulou, protože záporné číslování stránek by mohlo způsobit nežádoucí chování. Naopak hodnota parametru `limit` může nabývat hodnot 1–50, přičemž nejsou povoleny záporné hodnoty. Jako maximální hodnota byla zvolena 50, což je běžná hodnota ve veřejných API. Výsledný požadavek na získání dokumentů s využitím stránkování pak může vypadat takto: `/documents?page=0&limit=3`. Ve zdrojovém kódu 6.6 je znázorněna ukázka použití stránkování ve Spring Data JPA.

```
// vytvoří objekt, který slouží ke stránkování
Pageable pageable = PageRequest.of(pageNumber, pageSize);

// získá dokumenty s využitím stránkování
Page<Document> documents = repository.findAll(pageable);
```

Zdrojový kód 6.6: Ukázka stránkování ve Spring Data JPA

6.3.3 Řazení dat

Pro lepší organizaci získaných dat bylo implementováno řazení ve vybraných endpointech, které umožňuje řadit dokumenty a revize dokumentů podle jejich metadat. Toto řazení lze provádět i s více metadaty současně, přičemž jednotlivá metadata musí být oddělena čárkou.

V jednotlivých endpointech, které řazení podporují, je řazení zajištěno pomocí parametru požadavku `sort`. Hodnotou tohoto parametru je řetězec ve formátu: `název_atributu:asc,název_atributu:desc`. Celý tento řetězec, kde `asc` značí vzestupné řazení a `desc` sestupné řazení, je validován pomocí regexu. Výsledný požadavek na získání seřazených dokumentů pak může vypadat takto: `/documents?sort=name:desc`. V tab. 6.1 jsou uvedeny atributy, které lze použít k řazení dokumentů, a v tab. 6.2 jsou uvedeny atributy povolené pro řazení revizí dokumentů.

Atributy:	document_id, name, type, path, size, version, created_at, updated_at

Tabulka 6.1: Povolené atributy dokumentů k řazení

Atributy:	revision_id, name, type, size, version, created_at

Tabulka 6.2: Povolené atributy revizí dokumentů k řazení

6.3.4 Filtrování dat

V některých vybraných endpointech bylo implementováno filtrování dokumentů a revizí dokumentů podle jejich metadat, což umožňuje vyhledávání v datech. Tato funkcionalita byla vyvinuta pomocí technologie Spring Data JPA Specification, která umožňuje vytváření složitějších dotazů do databáze. Filtrování může být uskutečněno i s více metadaty najednou, přičemž jednotlivá metadata musí být oddělena čárkou.

V jednotlivých endpointech, které filtrování podporují, je filtrování zajištěno pomocí parametru požadavku `filter`. Hodnotou tohoto parametru je řetězec ve formátu: `název_atributu:"hodnota"`, přičemž celý tento řetězec je validován pomocí regexu. Filtrování funguje na principu kontroly, zda hodnota vybraného atributu obsahuje daný podřetězec `hodnota`. Výsledný požadavek na získání filtrovaných revizí dokumentů pak může vypadat takto: `/revisions?filter=name:"manual",type:"pdf"`. V tab. 6.3 jsou uvedeny atributy, které lze použít k filtrování dokumentů, a v tab. 6.4 jsou uvedeny atributy povolené pro filtrování revizí dokumentů.

Atributy:	name, type, path, is_archived

Tabulka 6.3: Povolené atributy dokumentů k filtrování

Atributy:	name, type

Tabulka 6.4: Povolené atributy revizí dokumentů k filtrování

6.3.5 Ošetření výjimek

Pro zajištění detekce nežádoucích stavů aplikace a identifikace neplatných hodnot v požadavcích na REST API, byly vytvořeny vlastní výjimky spolu s jejich ošetřením. Všechny výjimky, které mohou nastat při zasílání požadavku na REST API, jsou z důvodu přehlednosti ošetřeny v jedné jediné třídě `GlobalExceptionHandler`.

Každá výjimka je zde v příslušné metodě zpracována a je z ní vytvořen objekt třídy `ProblemDetail`, který obsahuje všechny potřebné informace o chybě, a který je následně odeslán jako odpověď. `ProblemDetail` v sobě vždy obsahuje i HTTP status, který odpovídá danému typu výjimky, tedy například nenalezenému dokumentu odpovídá HTTP status `404 Not Found`. V případě mnohonásobných chyb či

neplatných hodnot je v objektu `ProblemDetail` dodáno i pole `messages` popisující všechny identifikované problémy. Pokud uživatel udělá chybu při zasílání požadavku, tak pomocí výše uvedené odpovědi ve formě `ProblemDetail` bude schopen tuto chybu snadno odstranit. Ve zdrojovém kódu 6.7 je znázorněn příklad ošetření výjimky ve Spring Bootu v případě nenalezeného dokumentu.

```
@ExceptionHandler(DocumentNotFoundException.class)
public ProblemDetail handleDocumentNotFoundException(
    DocumentNotFoundException exception
) {
    ProblemDetail problemDetail =
        ProblemDetail.forStatusAndDetail(
            HttpStatus.NOT_FOUND, exception.getMessage()
        );
    problemDetail.setTitle("Document Not Found");

    return problemDetail;
}
```

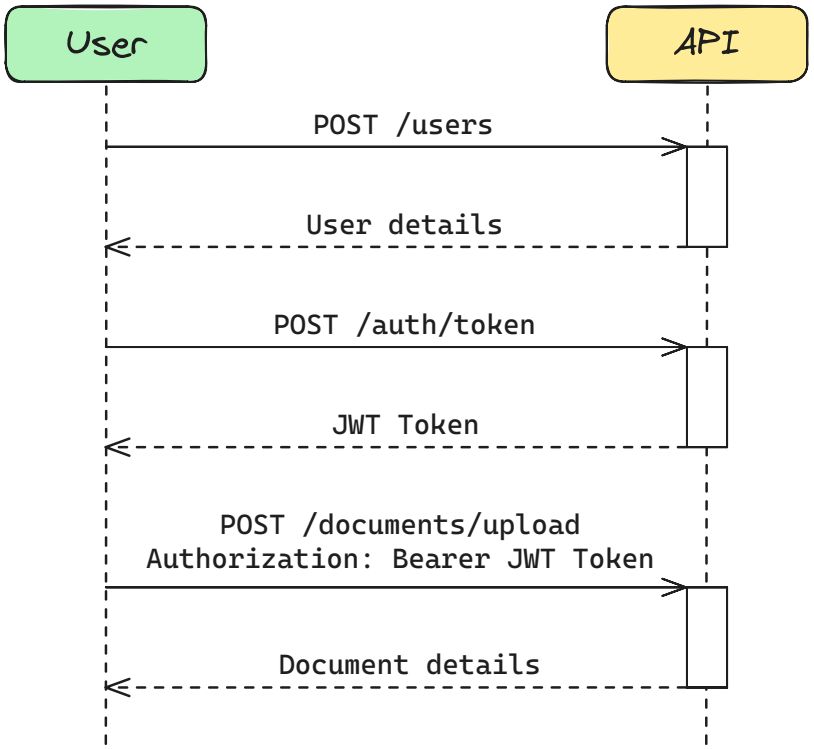
Zdrojový kód 6.7: Ukázka ošetření výjimky v případě nenalezeného dokumentu

6.4 Zabezpečení REST API

REST API bylo zabezpečeno pomocí Spring Security a JWT tokenu. Všechny endpointy API vyžadují v hlavičce požadavku JWT token. Výjimkou jsou endpointy `/users` a `/auth/token`, které token nevyžadují, protože se jedná o registraci a autentizaci uživatele. Na obr. 6.2 je znázorněno zabezpečení REST API pomocí JWT tokenu. Je zde zobrazen proces získání JWT tokenu a jeho následné dodání v hlavičce požadavku. Aplikace má 2 druhy uživatelských rolí: `ADMIN` a `USER`, přičemž uživatel s rolí `ADMIN` je vytvořen při spuštění aplikace. Tato část bude zaměřena na registraci a autentizaci uživatelů, generování RSA klíčů, sloužících pro podpis JWT tokenu, a nakonec na validaci vstupních hodnot a souborů.

6.4.1 Registrace uživatele

Pro vytvoření nového uživatele v systému je nutné, aby se daný uživatel nejprve zaregistroval. Registrace je umožněna prostřednictvím požadavku typu `POST` na endpoint `/users`. V těle tohoto požadavku uživatel dodá své uživatelské jméno, email a heslo s tím, že email musí být v systému jedinečný a heslo musí obsahovat alespoň jedno číslo a speciální znak. Ve zdrojovém kódu 6.8 je znázorněn příklad těla požadavku pro registraci uživatele. Po registraci uživatele je jeho heslo zahashováno pomocí BCryptu, přičemž daný hash je pak uložen do databáze, a je mu přiřazena role `USER`.



Obrázek 6.2: Ukázka zabezpečení REST API pomocí JWT tokenu

```
{
    "name": "james",
    "email": "james@gmail.com",
    "password": "secret123!"
}
```

Zdrojový kód 6.8: Ukázka těla požadavku pro registraci uživatele

6.4.2 Autentizace uživatele

Po úspěšné registraci se uživatel může autentizovat dodáním emailu a hesla, které zadával při registraci, v požadavku typu `POST` na endpoint `/auth/token`. Ve zdrojovém kódu 6.9 je zobrazen příklad těla požadavku pro autentizaci uživatele. V případě úspěšné autentizace, která probíhá na základě ověření emailu a hesla v databázi, se uživateli vygeneruje JWT token. Tento token pak uživatel dodává v hlavičce každého následujícího požadavku, čímž se autorizuje. Platnost JWT tokenu je nastavena na jednu hodinu, ale tuto dobu lze změnit v konfiguračním souboru `application.yaml` pomocí hodnoty klíče `token.expiration.time`. Vygenerovaný JWT

token v sobě obsahuje mimo jiné i email uživatele spolu s jeho rolí a je zabezpečen asymetrickým algoritmem RSA, využívajícím soukromý a veřejný klíč.

```
{
    "email": "james@gmail.com",
    "password": "secret123!"
}
```

Zdrojový kód 6.9: Ukázka těla požadavku pro autentizaci uživatele

6.4.3 Generování RSA klíčů

K zajištění bezpečného podpisu JWT tokenu byl použit asymetrický RSA algoritmus, který využívá soukromý a veřejný klíč. Pro správu těchto klíčů byla implementována třída `KeyManager`. Tato třída buď dané klíče vygeneruje, nebo je pouze načte, v závislosti na tom, zda již klíče existují či nikoliv. Klíče pak ukládá do souborů nastavených v konfiguračním souboru `application.yaml`, v hodnotách klíčů: `rsa.private-key` a `rsa.public-key` (viz zdrojový kód 6.10). Výchozí adresář, kam budou klíče uloženy, je `resources/certs`.

```
rsa:
  private-key: src/main/resources/certs/private.pem
  public-key: src/main/resources/certs/public.pem
```

Zdrojový kód 6.10: Příklad konfigurace souborů pro RSA klíče

6.4.4 Validace vstupních hodnot

Vzhledem k tomu, že je v požadavcích na REST API možné zadat libovolné hodnoty, byla implementována validace těchto hodnot. Validace hodnot byla psána v OpenAPI Specifikaci, jak je vidět ve zdrojovém kódu 6.11. Je to z toho důvodu, že generátor, který z OpenAPI Specifikace následně generuje kód serveru, je schopen tyto validace přenést do kódu prostřednictvím anotací (viz zdrojový kód 6.12).

```
email:
  type: string
  format: email
  description: "The email of the user."
  minLength: 3
  maxLength: 255
  example: james@gmail.com
```

Zdrojový kód 6.11: Ukázka validace emailu v OpenAPI Specifikaci

```
@Email
@NotNull
@Size(min = 3, max = 255)
@Schema(
    name = "email",
    example = "james@gmail.com",
    description = "The email of the user."
)
@JsonProperty("email")
public String getEmail() {
    return email;
}
```

Zdrojový kód 6.12: Ukázka validace emailu ve Spring Bootu

6.4.5 Validace souborů

V případě endpointů, které umožňují nahrávání souboru nastal problém. Problémem bylo, že pro nahrávání souborů je ve Spring Bootu použito rozhraní `MultipartFile`, které je nainicializováno i v případě nenahrání souboru. Kvůli tomu bylo možné do systému nahrát i neexistující soubor a uživatel tak nemohl být informován, že soubor není validní. Jediným řešením bylo dodat validaci pro soubory, která nedovolí nahrát neexistující soubory. Ovšem OpenAPI Specifikace ve výchozím provedení nepodporuje validování souborů a proto bylo nutné vytvořit si vlastní validaci.

Pro kontrolu souborů byla vytvořena anotace `@ValidFile` (viz zdrojový kód 6.13), která ověřuje jejich existenci. Tato anotace pak byla přidána do OpenAPI Specifikace pomocí `x-constraints: "@ValidFile"` (viz zdrojový kód 6.14). Klíč `x-constraints` pak bylo nutné přidat do OpenAPI generátoru, aby z něho byl schopen správně vygenerovat `@ValidFile` anotaci.

Na GitHubu OpenAPI generátoru byla nalezena šablona `formParams.mustache`, která je používána pro generování parametrů API a tato šablona byla následně upravena a přidána do projektu do adresáře `resources/static/templates`. Do této šablony byl na začátek přidán kód, který je znázorněn ve zdrojovém kódu 6.15. Při generování serveru pak byla využita tato šablona, čímž se před parametr odpovídající souboru přidala anotace `@ValidFile` a validace souboru tím byla úspěšně naimplementována.

```
@Documented
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = {FileValidator.class})
public @interface ValidFile {
```

```
    String message() default "File is mandatory";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Zdrojový kód 6.13: Ukázka vytvoření anotace `@ValidFile`

```
            file:
              type: string
              format: binary
              description: File to be uploaded.
              x-constraints: "@ValidFile"
```

Zdrojový kód 6.14: Ukázka vlastní validace souboru v OpenAPI Specifikaci

```
{{#vendorExtensions.x-constraints}}{{.}} {{/vendorExtensions.
   x-constraints}}
```

Zdrojový kód 6.15: Přidaný kód v šabloně OpenAPI generátoru

6.5 Logování

Jako logovací framework byl zvolen Log4j2, především kvůli jeho integraci se Spring Bootem. Konfigurace Log4j2 byla napsána v YAML formátu a nachází se v souboru `log4j2.yaml`. Logovány jsou všechny příchozí požadavky na REST API spolu s dalšími důležitými událostmi v aplikaci. Na základě závažnosti těchto událostí jsou voleny příslušné logovací úrovně. Logy jsou zapisovány do souborů (viz zdrojový kód C.1), které se přepisují a ukládají do adresáře `log`. Maximální počet těchto souborů je 4, přičemž jeden obsahuje aktuální logy a zbývající tři obsahují starší logy.

Po spuštění aplikace je možné za běhu měnit úrovně logování pomocí Spring Boot Actuatoru, který do REST API automaticky dodá několik dalších endpointů pro monitorování a správu aplikace. Jedním z těchto endpointů je i `/actuator/loggers`, ve kterém lze měnit úrovně logování libovolných loggerů aplikace. Změnu logovací úrovně nakonfigurovaného loggeru `com.dms` lze provést požadavkem typu `POST` na endpoint `/actuator/loggers/com.dms` a dodáním těla požadavku (viz zdrojový kód 6.16). Tuto změnu může provádět pouze uživatel s rolí `ADMIN`.

```
{
    "configuredLevel": "INFO"
}
```

Zdrojový kód 6.16: Ukázka těla požadavku pro změnu logovací úrovně na INFO

Testování 7

Pro zajištění kvality aplikace bylo nezbytné ověřit, zda aplikace plní svůj účel a funguje v souladu s očekáváním. Tento proces byl uskutečněn prostřednictvím testování, které zahrnovalo jak manuální testy, tak automatické jednotkové a integrační testy. Manuální testování umožnilo přímou interakci s aplikací a podrobné ověření jejích funkcionalit z pohledu koncového uživatele. Naopak, automatické jednotkové a integrační testy poskytly prostředek pro efektivní a opakované testování klíčových částí aplikace, což výrazně zrychlilo proces testování a zajišťovalo jeho konzistenci. Tyto testy hrály klíčovou roli při odhalování potenciálních chyb v kódu a zajištění správné integrace jednotlivých komponent aplikace. V této kapitole budou detailněji rozebrány jednotlivé typy testů, které byly použity v průběhu testování, a následně budou vyhodnoceny výsledky testů, včetně pokrytí kódu.

7.1 Manuální testy

Prvním krokem bylo provádění manuálních testů, které sloužily k ověření správného zpracování příchozích požadavků a odpovědí ze strany REST API. Tyto testy byly realizovány pomocí nástroje Postman (viz kapitola 3.5.4), který umožnil posílat různé požadavky s různými parametry a následně kontrolovat obsah odpovědí.

Pomocí manuálních testů bylo uskutečněno důkladné ověření funkcionality všech endpointů REST API, které byly představeny v předchozích kapitolách. Během testování se sledovalo, zda REST API korektně reaguje na platné požadavky a zda odpovědi obsahují očekávané informace. V případě neplatných požadavků bylo ověřeno, že REST API vrací odpovídající HTTP status a že odpovědi obsahují relevantní informace k identifikaci výskytu chyby. Dále byla ověřena bezpečnostní opatření implementovaná v REST API, aby se zajistila ochrana dat a prevence před potenciálními útoky.

Tento druh testů byl prováděn již od začátku vývoje REST API, což výrazně přispělo k jeho urychlení. Zároveň pomohl identifikovat a opravit většinu chyb, které se při vývoji API vyskytly. Po dokončení implementace došlo k opětovnému provedení těchto testů s cílem ověřit, zda již byly všechny identifikované chyby opraveny a zda

je funkčnost systému plně obnovena. V tab. 7.1 jsou uvedeny jednotlivé skupiny testovaných endpointů a čas zaokrouhlený na minuty, který označuje jak dlouho trvalo danou skupinu endpointů otestovat. I přestože manuální testování trvalo poměrně dlouho, přesněji 57 minut, všechny testy byly nakonec úspěšně dokončeny a žádné chyby nalezeny nebyly. V následujících kapitolách budou uvedeny způsoby, kterými byl proces manuálního testování urychlen a zautomatizován.

Skupina endpointů	**Čas**	**Výsledek**
uživatelé	9min	ÚSPĚCH
autentizace uživatele	4min	ÚSPĚCH
dokumenty	33min	ÚSPĚCH
revize dokumentů	11min	ÚSPĚCH

Tabulka 7.1: Výsledky manuálních testů a měření

7.2 Jednotkové testy

Automatické jednotkové testy byly vytvořeny z důvodu potřeby ověření nejmenších částí aplikace a zajištění kvality kódu. Pro testování byl použit framework JUnit verze 5 spolu s Mockitem, aby bylo možné simulovat chování určitých částí systému a izolovat testovaný kód od závislostí.

Tento druh testů sloužil k ověření správnosti mapování mezi objekty Data Transfer Object (DTO) a logikou služeb (třídami označenými jako `@Service`). Byly zahrnuty i negativní testovací scénáře s cílem ověřit, že aplikace správně zvládá chybné hodnoty a situace.

Zdrojový kód 7.1 znázorňuje jednotkový test pro kontrolu správnosti hashe. Pomocí anotace `@ExtendWith` se Mockito integruje do testovacího prostředí, což umožní používat metody z tohoto frameworku. Anotace `@InjectMocks` automaticky vkládá mock objekty do testované třídy, zatímco anotace `@Mock` označuje objekt, který má být vytvořen jako mock.

V tab. 7.2 jsou uvedeny jednotlivé testovací balíky, obsahující několik testovacích tříd. Dále jsou zde uvedeny informace o počtu testů v daném balíku a čas trvání všech testů v daném balíku. Celkový počet jednotkových testů činí **95**, což přispívá k robustnosti a spolehlivosti aplikace. Všechny testy byly vyhodnoceny úspěchem a proběhly bez zjištěných chyb nebo problémů. Výsledný čas pro vykonání všech testů nepřesahuje 2 sekundy, díky čemuž je tento typ testů vhodný pro rychlé a efektivní ověření funkcionality aplikace při každé iteraci vývoje. Zároveň jsou tyto testy násobně rychlejší než manuální testy a pokud se využijí opakovaně, umožní každé testování ušetřit 57 minut času, který by byl jinak vynaložen na ruční ověřování.

```
@ExtendWith(MockitoExtension.class)
class HashServiceTest {
    @InjectMocks
    private HashService hashService;

    @Mock
    private HashProperties hashProperties;

    @Test
    void shouldReturnHash() {
        String expectedHash = "4c2e9e6da31a64c7062";
        MockMultipartFile file = new MockMultipartFile(
            "document.pdf", "Some text".getBytes()
        );

        when(hashProperties.getAlgorithm())
            .thenReturn("SHA-256");

        String actualHash = hashService.hashFile(file);

        assertThat(actualHash).isEqualTo(expectedHash);
    }
}
```

Zdrojový kód 7.1: Ukázka jednotkového testu pro kontrolu správnosti hashe

Balík	Počet testů	Čas	Výsledek
mapper	29	0.05s	ÚSPĚCH
service	66	1.14s	ÚSPĚCH

Tabulka 7.2: Výsledky jednotkových testů a měření

7.3 Integrační testy

Po jednotkových testech bylo potřeba vytvořit i automatické integrační testy pro ověření správnosti integrace jednotlivých modulů a tříd v aplikaci. Integrační testy ve Spring aplikaci načítají vždy buď část, a nebo celý aplikační kontext, a proto tyto testy trvají déle, než jednotkové.

Pomocí těchto testů se ověřovala správná funkčnost databáze, zpracování REST API požadavků, zabezpečení REST API a vytváření RSA klíčů. Zároveň byly zahrnuty i negativní testovací scénáře pro kontrolu chybových odpovědí a HTTP statusů.

Zdrojový kód 7.2 znázorňuje integrační test pro autentizaci uživatele. Anotace `@SpringBootTest` označuje integrační test, jenž spustí celý aplikační kontext.

`@Transactional` zajistí, že všechny změny v databázi budou po každém testu vráceny do původního stavu. `@AutoConfigureMockMvc` automaticky nakonfiguruje `MockMvc` objekt, který umožňuje simulovat požadavky na REST API.

V tab. 7.3 jsou zobrazeny jednotlivé balíky testů, z nichž každý obsahuje několik testovacích tříd. Dále jsou zde poskytnuty údaje o počtu testů v jednotlivých balících a celkovém čase trvání testů v každém balíku. Celkový počet integračních testů je **175**. Všechny testy byly úspěšně dokončeny a proběhly bez zaznamenaných chyb nebo problémů. Výsledný čas pro vykonání všech testů se pohybuje okolo 5 sekund, a to z důvodu nutnosti načtení ať už celého či jen části aplikačního kontextu. Nejedná se však o dlouhou dobu a proto i tyto testy lze použít pro rychlé a efektivní ověření funkcionality aplikace. Zároveň jsou tyto testy násobně rychlejší než manuální testy a pokud se využijí opakovaně, umožní každé testování ušetřit zhruba 57 minut času, který by byl jinak vynaložen na ruční ověřování.

```
@SpringBootTest
@Transactional
@AutoConfigureMockMvc
class AuthControllerTest {
    @Autowired
    private MockMvc mvc;

    @Autowired
    private UserService userService;

    @Test
    void shouldReturnToken() throws Exception {
        User user = userService.createUser(user);

        mvc.perform(post("/auth/token")
                .contentType(MediaType.APPLICATION_JSON)
                .content("""
                        {
                            "email": "james@gmail.com",
                            "password": "secret123!"
                        }
                        """))
            .andExpectAll(
                status().isOk(),
                jsonPath("$.token").isNotEmpty()
            );
    }
}
```

Zdrojový kód 7.2: Ukázka integračního testu pro autentizaci uživatele

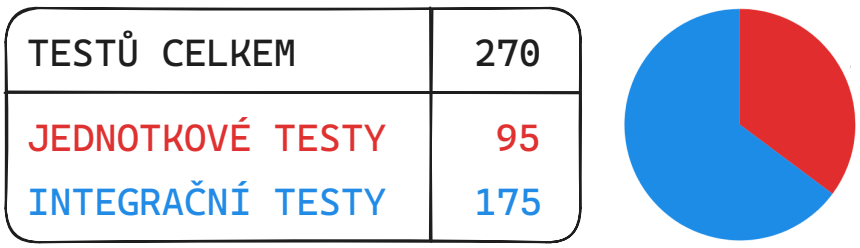
Balík	Počet testů	Čas	Výsledek
context	1	0.27s	ÚSPĚCH
controller	133	3.55s	ÚSPĚCH
repository	39	0.57s	ÚSPĚCH
util	2	0.74s	ÚSPĚCH

Tabulka 7.3: Výsledky integračních testů a měření

7.4 Pokrytí kódu testy

Automatické jednotkové a integrační testy odhalily několik chyb, které se během vývoje aplikace objevily, a tyto chyby byly následně opraveny. Celkový počet testů činí **270**, z toho je 95 jednotkových a 175 integračních (viz obr. 7.1). Tato rozmanitost testů zajišťuje komplexní pokrytí funkcí aplikace a umožňuje odhalení různorodých typů chyb a nedostatků v kódu. Celková doba trvání těchto testů nepřesahuje 7 sekund, což značně zkracuje časovou náročnost procesu testování a umožňuje rychlejší iterace vývoje. Díky efektivnímu testování jako je toto, je možné snadněji udržovat kvalitu kódu a rychleji reagovat na změny či požadavky.

Pokrytí kódu testy bylo vygenerováno pomocí příkazu `mvn test`, načež byl v adresáři `target/site/jacoco` vytvořen soubor `index.html`, který poskytuje přehled pokrytí kódu. Po otevření tohoto souboru, v libovolném prohlížeči, je možné si zobrazit celkové pokrytí kódu, jak je vidět na obr. 7.2. Do pokrytí kódu nejsou zahrnuty úseky kódu, které vygenerovaly příslušné frameworky či knihovny. Výsledné pokrytí kódu jednotkovými a integračními testy dosahuje významné hodnoty **90%**, což důrazně ilustruje robustnost a spolehlivost aplikace. Tato vysoká míra pokrytí poskytuje jistotu, že většina funkcí aplikace je důkladně testována a potenciální chyby jsou odhaleny ještě před nasazením do produkčního prostředí. Díky této systematické a důkladné testovací strategii bylo dosaženo vynikající úrovně kvality a spolehlivosti aplikace.



Obrázek 7.1: Celkový počet testů

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.dms.service		92%		78%
com.dms.exception		84%		n/a
com.dms.config		83%		62%
com.dms.util		82%		83%
com.dms.specification		89%		100%
com.dms.mapper.dto		100%		n/a
com.dms.controller		100%		100%
com.dms.mapper.entity		100%		100%
com.dms.validation		100%		100%
Total	374 of 4,053	90%	15 of 92	83%

Obrázek 7.2: Výsledné pokrytí kódu testy

7.5 Zhodnocení výsledků

Během psaní kódu byl využíván statický analyzátor `SonarLint`, který na konci vývoje ve zdrojovém kódu neodhalil žádné chyby, což svědčí o vysoké úrovni kvality implementace a pečlivosti při vývoji. Zároveň je kód komentovaný pomocí Javadocu, čímž je zajištěna dobře strukturovaná a srozumitelná dokumentace pro další vývoj a údržbu aplikace.

Výsledná aplikace byla otestována vytvořením několika uživatelů v systému a následným nahráváním dokumentů těmito uživateli. Uživatelé se v systému ověřovali pomocí JWT tokenu, který jim byl vygenerován po úspěšné autentizaci. Všechny dokumenty nahrané uživateli byly úspěšně uloženy v lokálním úložišti a aplikace i správně detekovala duplicitní dokumenty, a tudíž je znovu neukládala. Všichni uživatelé byli schopni si své uložené dokumenty stáhnout či vytvořit pro ně nové verze a následně se mezi těmito verzemi přepínat. Pro získávání metadat dokumentů a revizí využívali funkce stránkování, řazení a filtrování. Dokumenty také mohli archivovat s tím, že po nějaké době byly z úložiště odstraněny.

Všechny výše uvedené funkce byly prováděny pomocí manuálních testů v nástroji Postman, čímž byla ověřena použitelnost aplikace z pohledu běžného uživatele. Automatické jednotkové testy sloužily k ověření správné funkčnosti nejmenších částí aplikace. Automatické integrační testy pak sloužily k ověření správné komunikace mezi jednotlivými moduly aplikace a k ověření komunikace s REST API jako takovým. Díky těmto testům bylo potvrzeno, že aplikace splňuje stanovená očekávání a že je použitelná pro správu dokumentů.

Závěr 8

Hlavním cílem práce bylo navrhnout a implementovat DMS, se kterým se bude komunikovat prostřednictvím REST API. Mezi další cíle patřila i možnost úsporného a verzovaného ukládání dokumentů, detekce duplicitních dokumentů a archivace dokumentů. Dále pak celková bezpečnost aplikace a kompatibilita s několika vybranými SŘBD. Všechny tyto cíle byly úspěšně splněny.

Pro dosažení uvedených cílů bylo nejprve nezbytné provést podrobnou analýzu DMS, která zahrnovala zkoumání různých metod pro správu dokumentů v tomto systému. Následně bylo zapotřebí provést analýzu REST API z hlediska bezpečnosti a dostupnosti a seznámit se s nástroji a technologiemi, které jsou určeny pro práci s tímto rozhraním. Tato analýza pak sloužila jako základ pro návrh DMS a REST API.

Výsledkem práce je systém pro správu dokumentů, který kombinuje lokální úložiště a databázi pro uchovávání dokumentů. Zajištění integrity verzí dokumentů bylo dosaženo prostřednictvím úplné duplikace dat, přičemž detekce duplicit je realizována pomocí hashovacího algoritmu. Systém umožňuje archivaci dokumentů, které jsou po určitém čase odstraněny, ale uživatelé mají také možnost dokumenty z archivu obnovit. Kromě toho jsou podporovány různé databáze pro ukládání metadat o dokumentech, včetně H2, PostgreSQL, Oracle a MS SQL. Pro interakci s DMS bylo navrženo a implementováno REST API v Javě a Spring Bootu, které je podrobně zdokumentováno pomocí OpenAPI Specifikace a obsahuje detailní popis všech dostupných endpointů. Následně je možné si tuto dokumentaci zobrazit ve strukturované podobě pomocí Swagger UI. Zabezpečení REST API je zajištěno pomocí JWT tokenu, což umožňuje autentizaci a autorizaci uživatelů. Výsledná aplikace byla podrobena řadě testů, jako jsou manuální testy, automatické jednotkové a integrační testy. Pokrytí kódu automatickými testy dosahuje významných 90%, což důrazně potvrzuje spolehlivost aplikace. Díky těmto testům byla ověřena úspěšná implementace aplikace a její použitelnost pro správu dokumentů.

V budoucnu je možné na tuto práci navázat vytvořením uživatelského rozhraní pro správu aplikace DMS.

Přehled zkratek

DMS Document Management System

JWT JSON Web Token

DAO Data Access Object

DTO Data Transfer Object

SŘBD Systém Řízení Báze Dat

BLOB Binary Large Object

Bibliografie

[1] KIPLIE, Fatin Hazwani; YATIN, Saiful Farik Mat; ANGUTIM, Maizurah; HAMID, Nur Hanani Abdul. System development for document management system. *International Journal of Academic Research in Business and Social Sciences*. 2018, roč. 8, č. 9, s. 748–757.

[2] HART, Edmund M et al. *Ten simple rules for digital data storage*. Sv. 12. Public Library of Science San Francisco, CA USA, 2016. Č. 10.

[3] OBRUTSKY, Santiago. Cloud storage: Advantages, disadvantages and enterprise solutions for business. In: *Conference: EIT New Zealand*. 2016.

[4] SOUEIDI, Chukri. *Microsoft Azure Storage Essentials*. Packt Publishing Ltd, 2015.

[5] MATOS, Luis. *Azure Storage Account: Best Practices for efficient and secure storage* [online]. 2023. [cit. 2024-03-10]. Dostupné z: `https://luismts.com/azure-storage-account/`.

[6] GEEWAX, JJ. *Google Cloud platform in action*. Simon a Schuster, 2018.

[7] MARAN, Mihkel M; PANIAVIN, Nikolai A; POLIUSHKIN, Ilia A. Alternative approaches to data storing and processing. In: *2020 V International conference on information technologies in engineering education (Inforino)*. IEEE, 2020, s. 1–4.

[8] LU, Weiming et al. Hybrid storage architecture and efficient MapReduce processing for unstructured data. *Parallel Computing*. 2017, roč. 69, s. 63–77.

[9] SINGMAN, Paul. *Data versioning explained: Guide, examples & best practices* [online]. 2024. [cit. 2024-03-13]. Dostupné z: `https://lakefs.io/blog/data-versioning/`.

[10] MEERA, K; SANKAR, P Krishna; KUMAR, K Sriram. Redundant file finder, remover in mobile environment through SHA-3 algorithm. In: *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*. IEEE, 2015, s. 1440–1447.

[11] SETH, Preeti. *Duplicate files: A study of their detection and management* [online]. 2023. [cit. 2024-03-15]. Dostupné z: `https://www.systweak.com/blogs/duplicate-files-a-study-of-their-detection-and-management/`.

[12] PATIL, S; JAGTAP, N; RAJPUT, S; SANGORE, R. A Duplicate File Finder System. *International Journal of Science Spirituality Business and Technology*. 2017, s. 10–14.

[13] VIJAYARANI, S; MUTHULAKSHMI, M. An efficient string matching technique for desktop search to detect duplicate files. *International Journal of Information Technology and Computer Science*. 2017, roč. 9, s. 69–76.

[14] MARDER, Joachim. *How To Find Duplicate Files* [online]. 2021. [cit. 2024-03-20]. Dostupné z: `https://www.jam-software.com/blog/find-remove-duplicate-files.shtml`.

[15] MURPHY, Lauren; ALLIYU, Tosin; MACVEAN, Andrew; KERY, Mary Beth; MYERS, Brad A. Preliminary analysis of REST API style guidelines. *Ann Arbor*. 2017, roč. 1001, s. 48109.

[16] FISHER, Cameron. Cloud versus on-premise computing. *American Journal of Industrial and Business Management*. 2018, roč. 8, č. 9, s. 1991–2006.

[17] BROOKS, Rosie. *The definitive guide to rest API security: Best practices* [online]. 2023. [cit. 2024-03-24]. Dostupné z: `https://blog.stoplight.io/the-definitive-guide-to-rest-api-security-best-practices-and-advanced-strategies`.

[18] MADDEN, Neil. *API security in action*. Simon a Schuster, 2020.

[19] HUSSAIN, Fatima; HUSSAIN, Rasheed; NOYE, Brett; SHARIEH, Salah. Enterprise API security and GDPR compliance: Design and implementation perspective. *IT professional*. 2020, roč. 22, č. 5, s. 81–89.

[20] HUSSAIN, Muhammad Imran; DILBER, Naveed. Restful web services security by using ASP. NET web API MVC based. *Journal of Independent Studies and Research*. 2014, roč. 12, č. 1, s. 4.

[21] QAZI, Farhan. Application Programming Interface (API) Security in Cloud Applications. *EAI Endorsed Transactions on Cloud Systems*. 2022, roč. 7, č. 23, s. 1–14.

[22] SCHWICHTENBERG, Simon; GERTH, Christian; ENGELS, Gregor. From open API to semantic specifications and code adapters. In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, s. 484–491.

[23] DOS SANTOS, Jéssica Soares; AZEVEDO, Leonardo Guerreiro; SOARES, Elton FS; THIAGO, Raphael Melo; SILVA, Viviane Torres da. Analysis of Tools for REST Contract Specification in Swagger/OpenAPI. In: *ICEIS (2)*. 2020, s. 201–208.

[24] DASH, Tadit. *Effortless Swagger UI Installation* [online]. 2022. [cit. 2024-04-01]. Dostupné z: `https://blog.taditdash.com/effortless-swagger-ui-installation`.

[25] WALLS, Craig. *Spring Boot in action*. Simon a Schuster, 2015.

[26] LOCK, Andrew. *ASP. NET core in Action*. Simon a Schuster, 2023.

[27] BELL, Andrew; RENSEN, Sander; WEIR, Luis; WILKINS, Phil. *Implementing Oracle API Platform Cloud Service: Design, deploy, and manage your APIs in Oracle's new API Platform*. Packt Publishing Ltd, 2018.

[28] WESTERVELD, Dave. *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd, 2021.

[29] AHMAD, Imam; SUWARNI, Emi; BORMAN, Rohmat Indra; ROSSI, Farli; JUSMAN, Yessi et al. Implementation of RESTful API Web Services Architecture in Takeaway Application Development. In: *2021 1st International Conference on Electronic and Electrical Engineering and Intelligent System*. 2021, s. 132–137.

[30] GUTIERREZ, Felipe. *Introducing spring framework: a primer*. Apress, 2014.

[31] JOHNSON, Rod et al. The spring framework-reference documentation. *interface*. 2004, roč. 21, s. 27.

[32] TUDOSE, Catalin; BAUER, Christian; KING, Gavin. *Java Persistence with Spring Data and Hibernate*. Simon a Schuster, 2023.

[33] SPILCA, Laurentiu. *Spring security in action*. Simon a Schuster, 2020.

[34] JAIN, Manik. Automated Liquibase Generator And Validator(ALGV). *International Journal of Scientific & Technology Research*. 2015, roč. 4, s. 248–256.

[35] SHINGALA, Krishna. JSON Web Token (JWT) based client authentication in Message Queuing Telemetry Transport (MQTT). *ArXiv*. 2019.

[36] BUCKO, Ahmet; VISHI, Kamer; KRASNIQI, Bujar; REXHA, Blerim. Enhancing JWT Authentication and Authorization in Web Applications Based on User Behavior History. *Computers*. 2023, roč. 12, č. 4, s. 78.

Vytvoření tabulky v Liquibase

```
- createTable:
    tableName: app_user
    columns:
      - column:
          name: id
          type: bigint
          defaultValueSequenceNext: app_user_id_sequence
          constraints:
            primaryKey: true
            nullable: false
      - column:
          name: name
          type: varchar(40)
          constraints:
            nullable: false
      - column:
          name: email
          type: varchar(255)
          constraints:
            nullable: false
            unique: true
      - column:
          name: password_hash
          type: varchar(255)
          constraints:
            nullable: false
      - column:
          name: role
          type: varchar(5)
          constraints:
            nullable: false
```

Zdrojový kód A.1: Ukázka vytvoření tabulky uživatelů v Liquibase

Entita uživatele

B

```
@Entity
@Table(name = "app_user")
public class User {
    @Id
    @SequenceGenerator(
        name = "app_user_id_generator",
        sequenceName = "app_user_id_sequence",
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "app_user_id_generator"
    )
    private Long id;

    @Column(nullable = false)
    private String userId;

    @Column(length = 40,nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(name = "password_hash", nullable = false)
    private String password;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private Role role;
}
```

Zdrojový kód B.1: Ukázka entity uživatele

Vytvoření souboru pro logy v Log4j2

```
Appenders:

  RollingFile:
    - name: InfoFile
      fileName: log/log.log
      filePattern: log/log_%i.log
      append: false
      immediateFlush: true

      LevelRangeFilter:
        maxLevel: INFO
        onMatch: accept
        onMismatch: deny

      PatternLayout:
        pattern: "%d{yyyy-MM-dd HH:mm:ss.SSS} %5level ---
   [%15.15t] %C{36} - %msg%n%wEx"
        disableAnsi: true

      Policies:
        SizeBasedTriggeringPolicy:
          size: 10MB

      DefaultRolloverStrategy:
        fileIndex: max
        min: 1
        max: 3
```

Zdrojový kód C.1: Ukázka vytvoření souboru pro logy v Log4j2

Programátorská dokumentace

D.1 Struktura projektu

```
src
├── main
│   ├── java
│   │   └── com.dms
│   │       ├── config ..... konfigurační třídy
│   │       ├── controller ..... REST API controller
│   │       ├── entity ..... databázové entity
│   │       ├── exception ..... výjimky a jejich ošetření
│   │       ├── mapper ..... třídy pro mapování objektů
│   │       │   ├── dto ..... třídy pro mapování DTO objektů
│   │       │   └── entity ..... třídy pro mapování entit
│   │       ├── repository ..... DAO objekty
│   │       ├── service ..... třídy obsahující logiku aplikace
│   │       ├── specification ..... specifikace pro filtrování
│   │       ├── util ..... pomocné třídy
│   │       ├── validation ..... anotace pro validaci
│   │       └── DocumentManagerApplication.java .. hlavní třída aplikace
│   └── resources
│       ├── certs ..... vygenerované RSA klíče
│       ├── db.changelog ..... Liquibase changelogy
│       ├── static ..... OpenAPI soubory
│       ├── application.yaml ..... konfigurační soubor aplikace
│       └── log4j2.yaml ..... konfigurační soubor Loggeru
└── test
    └── java
        └── com.dms
            ├── integration ..... integrační testy
            └── unit ..... jednotkové testy
```

D.2 Požadavky na instalaci a spuštění

Pro úspěšnou instalaci a spuštění aplikace je zapotřebí mít nainstalované tyto technologie:

- Java 17+,
- Maven 3.9.6+.

Dále je pak nutné mít funkční alespoň jednu z těchto databází v závislosti na tom, jaká databáze bude používána:

- H2,
- PostgreSQL,
- Oracle,
- MS SQL.

D.3 Instalace a spuštění

1. **Otevření adresáře s aplikací**:
   - Otevřete kořenový adresář aplikace.
2. **Konfigurace**:
   - V konfiguračním souboru `src/main/resources/application.yaml`, aktualizujte tyto hodnoty:
     - **storage.path**: adresář na lokálním disku, kam budou ukládány soubory,
     - **storage.subdirectory-prefix-length**: délka předpony podadresáře,
     - **hash.algorithm**: hashovací algoritmus,
     - **archive.retention-period-days**: doba (ve dnech) uchování dokumentu,
     - **token.expiration.time**: doba (v hodinách) platnosti JWT tokenu,
     - **rsa.private-key**: soubor na lokálním disku, kam bude uložen soukromý klíč (musí končit na .pem),
     - **rsa.public-key**: soubor na lokálním disku, kam bude uložen veřejný klíč (musí končit na .pem),
     - **server.error.path**: přípona v URL adrese pro chyby,

  - **admin.name**: jméno uživatele s rolí ADMIN,
  - **admin.email**: email uživatele s rolí ADMIN,
  - **admin.password**: heslo uživatele s rolí ADMIN,
  - **spring.profiles.active**: výběr databáze:
    * **h2** pro použití H2,
    * **postgresql** pro použití PostgreSQL,
    * **oracle** pro použití Oracle,
    * **mssql** pro použití MS SQL.
  - Aktualizujte nastavení databáze ve zvoleném profilu:
    * **spring.datasource.url**: URL databáze,
    * **spring.datasource.username**: uživatelské jméno pro přístup do databáze,
    * **spring.datasource.password**: heslo pro přístup do databáze.

3. **Vygenerování Javadoc dokumentace a JaCoCo reportu** (nepovinné):

   - Pro **Windows**: `mvnw.cmd install`
   - Pro **Linux** a **macOS**: `./mvnw install`

4. **Instalace a spuštění aplikace**:

   - Pro **Windows**: `mvnw.cmd spring-boot:run`
   - Pro **Linux** a **macOS**: `./mvnw spring-boot:run`

D.4 Zdrojové kódy

Všechny zdrojové kódy, včetně Javadoc dokumentace, jsou k dispozici na GitHubu: `https://github.com/JakubPavlicek/DMS`.

Uživatelská dokumentace

E.1 Používání aplikace

Aplikace bude spuštěna na portu 8080 a bude možné s ní dělat následující věci:

- **posílání HTTP požadavků** (např. pomocí Postmana),
- **zobrazení dokumentace API ve Swagger UI** – viz `http://localhost:8080/swagger-ui/index.html`.

Pro možnost vyzkoušení si jednotlivých endpointů se v kořenovém adresáři projektu nachází soubor `DMS.json`. V tomto souboru je vyexportovaná kolekce z Postmana, která obsahuje všechny endpointy REST API. Daný soubor je pak možné naimportovat do Postmana, což zpřístupní jednotlivé endpointy spolu s ukázkovými hodnotami. Endpointy vycházejí z hodnot proměnných, které lze nalézt po kliknutí myší na kolekci `DMS` a následně na záložku `Variables`. Hodnoty těchto proměnných je pak nutné měnit v závislosti na tom, jaké ID či JWT tokeny REST API vygeneruje.

První kroky, které je třeba udělat:

1. Vytvoření uživatele.
   - Toho se dosáhne tím, že poskytnete jméno, email a heslo na `/users` endpoint (v Postmanovi: `Create user`).
2. Získání přístupového tokenu, abyste mohli používat API.
   - Toho se dosáhne tím, že poskytnete email a heslo na `/auth/token` endpoint (v Postmanovi: `Request access token`).
3. Po získání přístupového tokenu je třeba ho poskytnout v hlavičce každého následujícího požadavku (v Postmanovi: změna hodnoty příslušné proměnné v `DMS` kolekci).

Obsah přílohy

- `Text_prace` ..... text práce ve formátu PDF spolu se zdrojovými soubory
  - `img` ..... obrázky, které využívá šablona
  - `images` ..... vlastní obrázky použité v textu
  - `bakalarka.tex` ..... text práce v .tex formátu
  - `bakalarka.pdf` ..... text práce v .pdf formátu
- `Aplikace_a_knihovny` ..... zdrojové kódy, konfigurační soubory atp.
  - `DocumentManager` ..... aplikace DMS
    - `src` ..... zdrojové kódy a konfigurační soubory
    - `target` ..... přeložené zdrojové kódy
      - `generated-sources` ..... vygenerované zdrojové kódy
      - `site` ..... JaCoCo report
    - `javadoc` ..... Javadoc dokumentace
    - `DMS.json` ..... vyexportovaná kolekce z Postmana
    - `pom.xml` ..... konfigurační soubor k sestavení projektu
- `Readme.txt` ..... popis adresářové struktury

Seznam obrázků

Seznam tabulek

Seznam výpisů