FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Návrh a implementace backend pro projekt Česká elektronická knihovna
Student:	Bc. Tomáš Chvosta
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Ústavu pro českou literaturu (ÚČL) se podařilo digitalizovat cca 1 700 sbírek obsahujících přes 80 000 básní. Obsahy sbírek jsou uloženy v XML souborech a doplněny i nascanovanými stránkami daných fyzických knih. Cílem této práce je analyzovat navrhnout a vytvořit backend pro tuto knihovnu.
Postupujte v těchto krocích:
1. Seznamte se se strukturou vstupních dat.
2. Shromážděte požadavky na budoucí datové úložiště, zejména z pohledu API pro frontend a zamýšlenéstrojové zpracování.
3. Na základě požadavků zvolte vhodnou implementační platformu pro uložení dat.
4. Data převeďte do zvoleného úložiště.
5. Navrhněte, implementujte, otestujte a zdokumentujte API dle požadavků z bodu 2.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 20. dubna 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Návrh a implementace backend pro projekt Česká elektronická knihovna

Bc. Tomáš Chvosta

Katedra softwarového inženýrství
Vedoucí práce: Ing. Michal Valenta, Ph.D.

2. května 2021

Poděkování

Velké poděkování patří vedoucímu práce Ing. Michalovi Valentovi, Ph.D. za stálou odbornou pomoc a podporu v průběhu psaní této práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 2. května 2021 . . . . . .

Odkaz na tuto práci

Chvosta, Tomáš. *Návrh a implementace backend pro projekt Česká elektronická knihovna*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Diplomová práce se zabývá vytvořením backendu webové aplikace pro Ústav pro českou literaturu Akademie věd České republiky. Tato aplikace umožňuje snadno prohlížet a spravovat digitalizované sbírky básní. Nejprve je provedena a popsána analýza požadavků, která vychází především z existujícího řešení, které však neumožňuje snadnou správu a koordinaci dat. Následně je provedena samotná analýza děl a je vytvořen společný model digitalizovaných sbírek. Na základě analýzy požadavků a dat je proveden návrh a implementace datového úložiště a také REST API, které využívá frontend aplikace pro práci s daty. Pro implementaci datového úložiště byl použit databázový systém MongoDB, který spadá do kategorie NoSQL databází. Implementace aplikačního serveru, který poskytuje REST API, využívá především technologie Node.js a Express.js a je připravena pro budoucí rozšíření. Poslední část práce je věnována testování funkčnosti a výkonu implementovaného řešení.

Klíčová slova ÚČL AV ČR, NoSQL, MongoDB, RESTful API, Javascript, Node.js, Express.js, JSON, XML, Mongoose, Mongoeye, Swagger, TEI

Abstract

The diploma thesis is focused on the creation of a backend of a web application for the Institute of Czech Literature. This application allows for an easy way to browse and manage digitized collections of poems. First of all, an analysis of requirements is performed and described. This analysis is mainly based on the existing solution, which does not allow easy management and coordination of data. In the next chapter, an analysis of the actual works is performed and a common model of digitized collections is created. Based on the analysis of requirements and data analysis, a design and implementation of a database is realized, as well as the REST API, which is used by the frontend of the application, which then works with the data. MongoDB system, which falls into the NoSQL category of databases, was used for the implementation of the data layer. Application server was implemented using Node.js and Express.js technologies and is ready for future extensions. The last part of thesis is focused on testing of the functionality and performance of the implemented solution.

Keywords Institute of Czech Literature, NoSQL, MongoDB, RESTful API, Javascript, Node.js, Express.js, JSON, XML, Mongoose, Mongoeye, Swagger, TEI

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Donedávna byly nejpoužívanějším způsobem pro efektivní ukládání a správu dat tradiční databázové systémy. Nejpopulárnější z nich jsou bezpochyby databáze relační. Svět nerelačních databází jde však poslední dobou kupředu a tyto technologie jsou neustále rozvíjeny a zdokonalovány a pokrývají čím dál tím větší potřeby běžných uživatelů. Jednou z těchto potřeb je poskytnout Ústavu pro českou literaturu Akademie věd České republiky lepší databázi pro jejich digitalizované sbírky dat.

ÚČL AV ČR disponuje zhruba 1 700 digitalizovanými sbírkami, které obsahují přes 80 000 básní. Obsahy sbírek jsou uloženy ve formátu XML a k některým z nich jsou k dispozici i naskenované stránky fyzických knih. Dále existuje webová aplikace, která poskytuje díla široké veřejnosti. Současné řešení však nevyhovuje požadavků ÚČL AV ČR na správu jednotlivých děl.

Tato práce se zabývá návrhem a implementací backendu nové webové aplikace, která umožní lépe spravovat jednotlivá díla, usnadní badatelskou práci a umožní další analýzu v souvislosti s vytěžováním znalostí z dat.

V první fázi bude třeba analyzovat požadavky na datové úložiště a vybrat databázový systém, který bude tyto požadavky splňovat. Vzhledem k rozrůstajícím se možnostem NoSQL databází se předpokládá, že datové úložiště bude implementováno pomocí tohoto typu databází. Tradiční relační databáze pro toto řešení nebudou uvažovány.

V druhé fázi bude třeba analyzovat požadavky na samotnou webovou aplikaci. Požadavky se budou dělit na část týkající se frontendu aplikace a na část týkající se backendu aplikace. Tato práce však bude popisovat pouze část týkající se backendu, druhou část bude mít na starosti Filip Hladej v rámci bakalářské práce. Tyto požadavky je nutné zohlednit při výběru architektury aplikace a při samotném návrhu aplikace. Ten se bude týkat především návrhu aplikační vrstvy, se kterou bude komunikovat frontend aplikace pomocí Web service API a bude obsahovat veškerou aplikační logiku.

V dalším kroku bude potřeba zvolit vhodnou implementační platformu

pro navrhnuté datové úložiště a backend server aplikace. Dále bude následovat samotná implementace řešení. Na závěr bude nutné implementované řešení otestovat a odhalit všechny jeho nedostatky. Výstupem má být funkční a otestovaná aplikace, která umožní vědcům z ÚČL snadno spravovat jejich digitalizované sbírky básní a zároveň tato aplikace bude poskytovat rozhraní pro širokou veřejnost, která bude mít možnost provádět operace pro čtení těchto sbírek.

Kapitola 1

Kontext a cíle práce

Ústav pro českou literaturu Akademie věd České republiky (dále jen ÚČL) je veřejná výzkumná instituce, která provádí vědecký výzkum české literatury od počátků do současnosti. Zabývá se přitom jak literaturou uměleckou, tak i populární či triviální, dále pak dějinami českého divadla a dramatu. ÚČL se věnuje literární historii a teorii, literárněvědné lexikografii, editologii, textologii, bibliografii a jiným oborům literární vědy. Vedle vědecké činnosti udržuje ÚČL několik informačních zdrojů, které jsou přístupné široké veřejnosti. [6]

Jedním z těchto zdrojů je Česká elektronická knihovna obsahující poezii 19. a počátku 20. století. Záměrem tohoto projektu bylo shromáždit básnické texty od počátků novodobé česky psané poezie a tyto texty poskytnout široké veřejnosti ve formě elektronické knihovny. Jelikož byl rozsah české poezie 19. století příliš velký, omezil se ÚČL pouze na knižní vydání básnických sbírek. Předpoklad byl takový, že tato knihovna bude sloužit badatelům, studentům a ostatním zájemcům o poezii a umožní nové formy výzkumu. [7]

1.1 Současné řešení

V současné době disponuje ÚČL soubory dvou typů. V první řadě se jedná o XML[1] soubory obsahující bibliografické údaje o knize, text knihy a údaje o edičních poznámkách a komentářích. Tyto soubory obsahují i styly textu. K vytváření a úpravám těchto souborů používá ÚČL aplikaci Microsoft Word, ve které má připravenou šablonu s jednotlivými elementy knihy, které lze použít. Z této aplikace je následně exportován XML soubor. Druhým typem souborů jsou obrázky s naskenovanými stranami knih. Celkem má ÚČL 1700 sbírek ve formátu XML a ke 495 sbírkám existují obrázky s naskenovanými stranami.

[1] Extensible Markup Language

Pro zobrazení sbírek a práci s nimi nabízí ÚČL webovou aplikaci dostupnou na URL[2] `http://www.ceska-poezie.cz/cek/`. Ta po přihlášení zpřístupňuje všech 1700 básnických knih. Pro přihlášení do aplikace je nejprve nutné registrovat účet pomocí uživatelského jména a emailu. Po registraci je novému uživateli odesláno heslo na zadanou emailovou adresu. Po přihlášení do systému vidí uživatel následující uživatelské rozhraní.

Obrázek 1.1: Současné řešení České elektronické knihovny

Aplikace umožňuje zobrazení textu knihu nebo přehledné struktury celé knihy. Nechybí ani bibliografické údaje o knize a údaje o edičních úpravách. Také je možné zobrazit obrazovou podobu knihy případně výtvarný doprovod. Kromě práce s jednou knihou může uživatel vytvořit seznam vybraných děl a s těmi pak dále pracovat. Tato díla lze také zobrazit najednou. K dispozici

[2]Uniform Resource Locator

je také zobrazení statistických údajů jako počet slov a veršů, délka slov či veršů a také lze sestavit frekvenční či abecední slovník. Nechybí ani fulltextové a kontextové vyhledávání ve vybraném seznamu děl. [7]

1.2 Cíle práce

Současné řešení splňuje požadavky běžných uživatelů. Co však nesplňuje, jsou požadavky zaměstnanců ÚČL, kteří by chtěli využít aplikaci pro správu digitalizovaných sbírek. Přidávání či mazání sbírek do/z aplikace je aktuálně nemožné. Pro úpravu sbírek je nutno upravit přímo daný XML soubor. Jakékoliv rozšíření aplikace není možné, neboť ÚČL nemá přístup ke zdrojovým kódům. Navíc ani nemá přístup k databázi, kterou aplikace využívá. Z těchto pohledů je současné řešení nevyhovující.

Cílem této práce je analyzovat, navrhnout a vytvořit backend pro tuto knihovnu. V prvním kroku je třeba analyzovat strukturu vstupních dat a následně shromáždit požadavky na budoucí datové úložiště zejména z pohledu API[3] pro frontend a zamýšlené strojové zpracování. Na základě nových požadavků na aplikaci je třeba zvolit vhodnou implementační platformu pro uložení dat. Data je nutné následně převést do zvoleného úložiště a poté je třeba navrhnout, implementovat a zdokumentovat řešení. Mezi nové požadavky na aplikaci patří:

1. nové přístupné datové úložiště pro digitalizované sbírky
2. společné schéma pro všechny digitalizované sbírky
3. role pro jednotlivé uživatele
4. možnost importu sbírek do datového úložiště
5. možnost smazání sbírek z datového úložiště
6. možnost úpravy sbírek
7. lepší možnosti filtrace sbírek než v předchozím řešení
8. ukládání předchozích verzí sbírek
9. možnost budoucího rozšíření aplikace

Výsledné řešení je také potřeba otestovat a případně navrhnout opravu nalezených chyb.

[3] Application Programming Interface

Kapitola 2

Teoretická část

V této kapitole jsou popsány základní technologie týkající se práce. Na základě dohody s vedoucím práce bylo rozhodnuto, že pro uložení dat bude využita dokumentová databáze MongoDB. Jelikož se jedná o NoSQL databáze, je vhodné nejprve probrat teorii ohledně tohoto tématu. Dále jsou popsány základní datové formáty pro uložení sbírek básní, konkrétně se jedná o XML a JSON formát a také je přiblížena utilita Mongoeye, kterou lze využít pro analýzu dokumentů uložených v Mongo databázi.

2.1 NoSQL databáze

Není to tak dávno, kdy pod slovem „databáze“ si lidé představili především relační databáze. V posledních letech však začaly vznikat nové typy databází, jako např. dokumentové či objektové databáze. Každá z těchto databází se hodila na něco jiného, někde se prosadily více, někde méně. Žádný z nově vzniklých typů databází se nestal tak široce používaným, aby se o něm mluvilo jako o náhradě klasických relačních databází.

Pojem NoSQL neznamená „No to SQL“, tedy „Ne jazyku SQL“, jak je někdy mylně uváděno. Tyto databáze nemají nahradit klasické relační SQL[4] databáze ani jiný typ databází. Cílem těchto databází je nabídnout řešení pro nové typy aplikací, pro které současné databázové systémy nebyly navrženy, a tudíž jim nevyhovují. NoSQL ale neznamená ani „Not only SQL“, tedy „Nejen jazyk SQL“. Například systémy Oracle nebo PostgreSQL podporují širokou škálu jiných principů než jen jazyk SQL, ale do skupiny NoSQL databází je neřadíme.

U klasických databází většinou známe strukturu uložených dat. Aby bylo možné se na data dotazovat různými způsoby, jsou datové struktury rozděleny na co nejmenší kompaktní celky. Každý celek je uložen v samostatné tabulce

[4]Structured Query Language

a odpověď na dotaz je pak sestavena v závislosti obsahu těchto tabulek. Efektivita databázového systému závisí na implementaci jednotlivých operací, například zda jsou využity sekundární indexy. Relační databáze také poskytují záruky zachování konzistence databáze tím, že plně implementují transakční zpracování dotazů (vlastnosti ACID[5], tedy atomicitu, konzistenci, izolovanost a trvalost). Dále u tradičních relačních databází jsou poměrně náročné změny ve schématu. Schéma je sice možné měnit i za běhu, avšak tyto operace jsou v SQL realizovány pomocí jazyka DDL[6], který není obvykle přístupný všem uživatelům.

NoSQL databáze je možné využít i případech, kdy je schéma spravovaných databází nejednotné a proměnlivé, což je přesně případ sbírek básní ÚČL, kdy má každá kniha odlišnou strukturu. Míra této flexibility je u každé NoSQL databáze jiná, některé databáze vůbec schéma nemají, jiné mají pravidla pro schéma přísnější. Dále platí, že databázová schémata nemusí být nutně normalizovaná. Najít vhodné schéma je pro efektivnost zpracování jednotlivých databázových operací zásadní. Špatně zvolené schéma může efektivitě velmi uškodit. [8]

2.1.1 Základní principy

U NoSQL nesmíme opomenout teorii týkající se datové distribuce. Tato sekce tedy popisuje základní principy škálování a konzistence, modely distribuce a CAP teorém.

2.1.1.1 Škálovatelnost

Škálovatelnost je schopnost systému flexibilně reagovat na množství přibývajících dat a požadavků ze strany uživatelů, které zatěžují systém. Rozlišujeme dva základní typy škálování.

První typ představuje vertikální škálování (mnohdy také označováno jako up/down škálování). Podstatou tohoto škálování je zvyšování výkonu jediného serveru prostřednictvím výkonnějšího hardwaru. Mezi výhody tohoto škálování patří bezpochyby velmi snadná implementace. Nevýhodou je však to, že výkonný hardware pro servery vyrábí pouze velmi omezený počet firem. Dá se tedy očekávat, že náklady na pořízení budou poněkud vyšší. Další nevýhodou je, že i nejvýkonnější server má horní hranici svých možností. Také může postupem času nastat tzv. *vendor lock-in*, tedy že zákazník je nucen kupovat další výkonnější prvky stále od stejné firmy.

Dalším typem je horizontální škálování (mnohdy označováno jako out/in škálování), tedy distribuce problému na více uzlů. Jednotlivé uzly obsahují běžný hardware, výkon získáváme tím, že přidáváme mnoho uzlů. Výhodou tohoto škálování je cenová dostupnost oproti vertikálnímu škálování. Dále se

[5] Atomicita, konzistence, izolovanost, trvalost
[6] Data definition language

jedná o flexibilnější způsob škálování. Nevýhodou je nutnost použití distribuce dat, čímž nám narůstá komplexita systému.

S horizontálním škálováním souvisí pojem cluster představující kolekci vzájemně propojených uzlů. Jde o samostatné stroje s vlastními komponentami (procesorem, RAM[7] pamětí, disky atd.) a vlastním operačním systémem. [9] Tyto uzly jsou propojeny pomocí síťového protokolu (například TCP/IP[8]) a komunikují spolu pomocí zpráv. Mezi těmito uzly je třeba distribuovat data a zatížení serveru.

2.1.1.2 Distribuce dat

Ve světě NoSQL může nastat situace, kdy data nedistribuujeme a jsou uložena pouze na jediném místě. V některých databázových systémech je distribuce dat dokonce velmi obtížná. Naopak v jiných případech není možné zpracovávat data a požadavky v rámci jediného databázového serveru, a proto je třeba použít distribuci dat. Obecně máme k dispozici dva modely distribuce dat.

První model představuje rozdělení různých částí dat na různé uzly v clusteru a tím získání větší kapacity systému. Tento model také nazýváme *sharding* a jednotlivým částem dat říkáme *shardy*. Rozdělená data nemusí představovat nutně disjunktní podmnožiny. Každý uživatel pak může přistupovat k jiným uzlům podle toho, jaká data potřebuje. Při rozdělování dat se snažíme dosáhnout kompromisu mezi následujícími cíli:

1. rozdělení dat rovnoměrně na všech uzlech
2. minimalizace počtu uzlů, které jsou potřeba k získání konkrétních dat
3. optimalizace fyzického rozmístění dat vzhledem k geografickému umístění serveru, který data obsahuje.

Druhým modelem distribuce je tzv. *replikace*, která představuje vytvoření kopií dat na více uzlech v clusteru. Díky replikaci zvýšíme dostupnost a propustnost systému. Replikace dat může být buď typu *Master-slave* nebo *Peer-to-peer*.

U replikace typu Master-slave je jeden z uzlů určen jako primární (*master*), ostatní uzly jsou sekundární (*slaves*). Primární uzel obvykle obsluhuje požadavky pro zápis, naopak sekundární obsluhuje požadavky pouze pro čtení. Tento způsob replikace je vhodný v případě, že jsou data především čtena a minimálně modifikována. Pokud selže primární uzel, mají sekundární uzly kopii jeho dat, tudíž se v případě potřeby může kterýkoliv z nich stát primárním uzlem. Naopak u replikace typu Peer-to-peer jsou si všechny uzly rovny a mohou tedy obsluhovat požadavky pro zápis i čtení. Pokud dojde k výpadku některého z uzlů, nepřicházíme o možnost čtení a zápisu dat. Na

[7] Random Access Memory
[8] Transmission Control Protocol/Internet Protocol

druhou stranu vznikají problémy s konzistencí dat. Pokud dva různé požadavky potřebují aktualizovat stejná data, je potřeba zajistit, aby data zůstala konzistentní. Tento problém lze řešit dvěma způsoby. Buď se uzly při zápisu dat musí koordinovat, čímž jsou zvýšeny nároky na komunikaci v systému, nebo lze využít volební princip „rozhoduje většina“, který se realizuje pomocí tzv. *kvóra*, kdy stanovujeme minimální počet uzlů, na kterých musí operace proběhnout, aby byla prohlášena za úspěšně dokončenou.

V praxi většinou používáme kombinaci replikace a rozdělení. Nejprve jsou data rozdělena podle cílů shardingu a následně je zvolen jeden z uvedených typů replikace. [8]

2.1.1.3 CAP teorém

Předpokládejme, že máme distribuovaný systém, ve kterém používáme replikaci a sharding. Poté můžeme uvažovat CAP teorém (též zvaný Brewerův teorém podle Erica Brewera), který popisuje nejdůležitější vlastnosti pro distribuované databázové systémy:

- **C (consistency)** představuje striktní konzistenci. Systémy s touto vlastností uspořádají dotazy do posloupnosti a provádějí je vždy nad stavem, který je výsledkem provedení všech předchozích dotazů v posloupnosti.
- **A (availability)** představuje dostupnost systému a zaručuje, že veškeré požadavky na čtení nebo zápis budou systémem vyřízeny.
- **P (partition tolerance)** představuje odolnost vůči výpadkům sítě. Tato vlastnost označuje systémy, u kterých návrh počítá s výpadky během komunikace. Tyto systémy fungují dál i v případě, že dojde ke zdržení či ztrátě části zpráv během komunikace v síti.

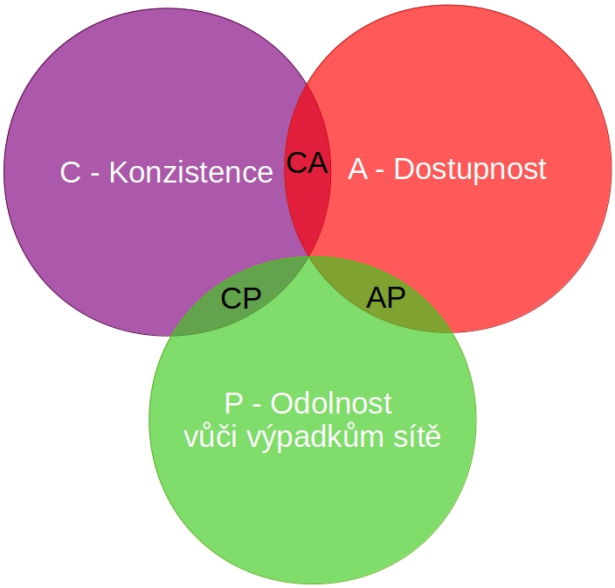
Podle Brewerova teorému nemůže databáze splňovat všechny vlastnosti CAP najednou, avšak všechny ostatní kombinace dvou vlastností jsou přípustné. Můžeme tedy zvolit, které dvě vlastnosti zachováme a kterou vlastnost zanedbáme s rizikem pádu systému:

- **CA (consistency + availability)** používáme k zajištění striktní konzistence a dostupnosti systému. K zajištění striktní konzistence je nutná synchronizace celé databáze, která ale vyžaduje zaručenou komunikaci mezi všemi uzly systému. To však vylučuje vlastnost P.
- **CP (consistency + partition tolerance)** používáme k zajištění striktní konzistence a odolnosti vůči výpadkům komunikace. Přerušení komunikace u konzistentní databáze však vylučuje vlastnost A, tedy vyhodnotit některé dotazy z posloupnosti dotazů.

- **CA (availability + partition tolerance)** používáme k zajištění dostupnosti a odolnosti vůči výpadkům komunikace. Pokud není databáze zcela propojena, nelze vyhodnotit spolehlivě dotaz v případě výpadku komunikace. Spolehlivě se dá pouze zjistit hodnota dat ve stavu, který předcházel výpadku spojení. Vyhodnocení dotazu tedy nevychází z aktuální hodnoty, a proto nelze splnit vlastnost C.

Při implementaci datového úložiště je třeba zvolit, kterou z těchto kombinací je vhodné využít. Volba závisí především na charakteru aplikace. [10]

Obrázek 2.1: CAP teorém



2.1.1.4 Občasná konzistence

U tradičních databázových systémů máme vlastnosti ACID (atomicitu, konzistenci, izolovanost a trvalost). Ve světě distribuovaných systémů máme model BASE, díky kterému získáváme škálovatelnost za cenu nižší míry konzistence dat. Vlastnosti tohoto modelu jsou:

- **BA (basically available)** představuje převážnou dostupnost. Systém je převážně dostupný po celou dobu a nikdy nedojde k výpadku celého systému.
- **S (soft state)** představuje volný stav, tedy že v systému neustále dochází ke změnám.

- **E (eventual consistency)** představuje občasnou konzistenci. Systém se může nacházet v konzistentním stavu, ta však není vždy zaručena.

Občasné konzistence se hodí v případech, kdy je rychlost aplikací mnohem důležitější než konzistence dat. U klasických relačních databází je vynucována silná konzistence, která z principu zpomaluje načítání dat. Díky občasné konzistenci můžeme zefektivnit manipulaci s daty. Vzniká však otázka, zda nemůže případná nekonzistence v datech způsobit nějaký problém. U aplikací je obvykle předpokládáno, že většina dotazů slouží ke čtení dat a daleko menší část dotazů vyžaduje modifikaci dat. Tento přístup je někdy označován jako optimistický a předpokládá, že k problémům s konzistencí dat dochází minimálně. Pokud však k problému dojde, umí ho systém obvykle řešit. [8]

2.1.2 Typy NoSQL databází

V současné době máme několik typů NoSQL databází s různou topologií. Jednotlivé typy se od sebe mohou výrazně lišit. V následujících sekcích jsou uvedeny nejznámější typy NoSQL databází.

2.1.2.1 Grafové databáze

Již název vypovídá, že v grafových databázích pracujeme se strukturou, kterou je graf. Graf je množina vrcholů, které jsou vzájemně propojeny pomocí hran. Vrcholy v grafu odpovídají objektům a mohou mít atributy jako například jméno, věk, místo narození. Vztahy mezi těmito objekty pak vyjádříme pomocí orientovaných hran. Hrany také mohou mít atributy, dobu platnosti vztahu, podmínky platnosti vztahu apod. Obecně nemáme žádné omezení na konkrétní typ vztahů, ani na jejich počet.

Hlavní výhodou použití grafové databáze namísto uložení grafové struktury pomocí relační databáze je vlastnost tzv. $index\text{-}free\text{-}adjacency$, což znamená, že je zajištěno nejefektivnější uložení vzhledem k předpokládaným klasickým operacím nad grafy. Druhou výhodou je, že průchod grafem nevyhodnocujeme v okamžiku vyhodnocování dotazu, ale máme předpřipravené a uložené vhodné datové struktury, které jsou optimální bez ohledu na strukturu konkrétního grafu. [8].

Grafové databáze jsou, obdobně jako všechny typy NoSQL databází, určené pro specifickou kategorii problémů. Není tedy žádoucí transformovat všechna fungující schémata z relačních databází do grafových databází. Pro použití grafové databáze musí být dobrý důvod. Typickým příkladem pro využití grafových databází jsou sociální sítě, tedy jakákoliv oblast, která je bohatá na vztahy. Jiným příkladem mohou být třeba doporučovací systémy. [11]

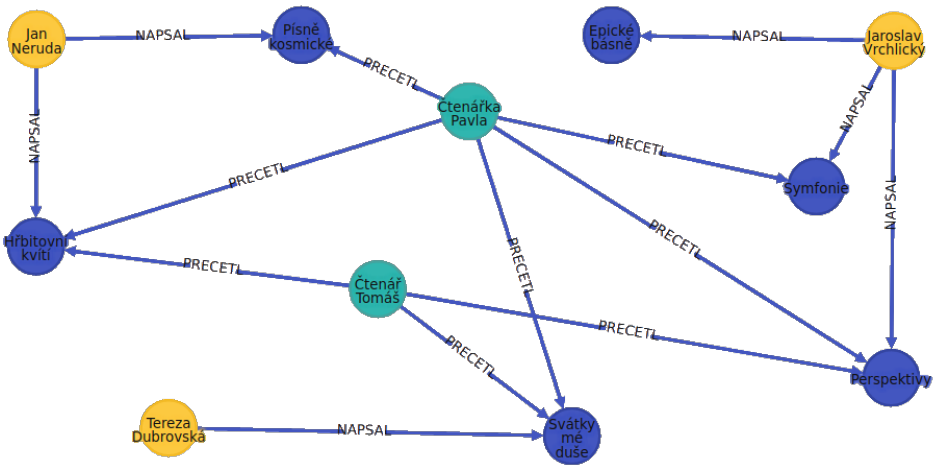
Mezi nejznámější a nejpopulárnější grafové databáze patří systém Neo4j, který je implementován v jazyce Java. Tento systém je velmi jednoduchý, intuitivní, spolehlivý, škálovatelný a také je dobře dostupný v open source

i komerční verzi. Datový model v Neo4j zahrnuje uzly a hrany. Hrany mají svůj typ, který je určený názvem, a jsou vždy orientované. Díky tomu můžeme rozlišit, zda se jedná o vstupní nebo výstupní hranu. Uzly i hrany mohou mít své atributy, které představují dvojici (klíč, hodnota). Klíč vždy představuje datový typ `string`. Naopak hodnota může mít více různých datových typů jako například `integer`, `float`, `byte` či `boolean`. V Neo4j nenajdeme hodnotu `null`. Pokud chceme vyjádřit, že uzel či hrana nemá daný atribut, pak příslušnou dvojici (klíč, hodnota) neuvedeme. [12] Na následujících obrázcích je zobrazen příklad jednoduchého schématu grafové databáze Neo4j.

Obrázek 2.2: Ukázka vytvoření jednoduché databáze

```
CREATE
(ctenar_Tomas:Ctenar {name:'Čtenář Tomáš'}),
(ctenarka_Pavla:Ctenar {name:'Čtenářka Pavla'}),
(vrchlicky:Spisovatel {name:'Jaroslav Vrchlický'}),
(neruda:Spisovatel {name:'Jan Neruda'}),
(dubrovska:Spisovatel {name:'Tereza Dubrovská'}),
(epickebasne:Kniha {name:'Epické básně'}),
(perspektivy:Kniha {name:'Perspektivy'}),
(symfonie:Kniha {name:'Symfonie'}),
(hrbitovnikviti:Kniha {name:'Hřbitovní kvítí'}),
(pisnekosmicke:Kniha {name:'Písně kosmické'}),
(svatkymeduse:Kniha {name:'Svátky mé duše'}),
(vrchlicky)-[:NAPSAL]->(epickebasne),
(vrchlicky)-[:NAPSAL]->(perspektivy),
(vrchlicky)-[:NAPSAL]->(symfonie),
(neruda)-[:NAPSAL]->(hrbitovnikviti),
(neruda)-[:NAPSAL]->(pisnekosmicke),
(dubrovska)-[:NAPSAL]->(svatkymeduse),
(ctenar_Tomas)-[:PRECETL]->(perspektivy),
(ctenarka_Pavla)-[:PRECETL]->(symfonie),
(ctenarka_Pavla)-[:PRECETL]->(perspektivy),
(ctenar_Tomas)-[:PRECETL]->(svatkymeduse),
(ctenarka_Pavla)-[:PRECETL]->(svatkymeduse),
(ctenar_Tomas)-[:PRECETL]->(hrbitovnikviti),
(ctenarka_Pavla)-[:PRECETL]->(pisnekosmicke),
(ctenarka_Pavla)-[:PRECETL]->(hrbitovnikviti);
```

Obrázek 2.3: Ukázka grafové databáze [1]



Pro přístup k databázi Neo4j lze využít Java API, REST API nebo můžeme využít celou řadu rozhraní nabízejících speciální jazyky pro práci s grafy. Konkrétně se jedná například o jazyky Gremlin nebo Cypher.

Databáze Neo4j není samozřejmě jediná grafová databáze. Mezi další patří například databázové systémy Sparksee, InfiniteGraph, OrientDB nebo třeba Titan. Dále je dobré zmínit RDF databáze, které jsou zřejmě nejznámějším typem databází pro grafová data reprezentována RDF[9] formátem.

Grafové databáze jsou optimální pro ukládání grafových struktur nebo dat, ve kterých je příliš mnoho vztahů. Je však nutno dodat, že nejsou vhodné pro jakákoliv data, vždy je potřeba zvážit výhody a nevýhody použití databáze pro konkrétní aplikace. Pojďme se nyní podívat, zda jsou grafové databáze vhodné pro realizaci datového úložiště pro digitalizované knihy ÚČL.

Tabulka 2.1: Využití grafových databází pro potřeby řešení ÚČL

Graf. DB	Výhody	Nevýhody
	• žádná omezení na definovaný model • rychlý průchod grafem	• vhodné pro data vyžadující mnoho vazeb M:N • vhodné pro data, kde jsou vztahy stejně nebo více důležité než entity

Jak je v tabulce výše uvedeno, největší výhodou je minimálně omezené schéma databáze. Vzhledem k tomu, že mají digitalizované knihy Ústavu pro českou literaturu navzájem odlišnou strukturu, je tato vlastnost velmi důležitá. Druhou výhodou je velmi efektivní a rychlý průchod datovou strukturou. Problém však představuje skutečnost, že data z knih nejsou příliš vhodná pro grafová vyhledávání. Dále je vhodné využít grafové databáze v případě, že data obsahují mnoho vazeb M:N. Z analýzy v následující kapitole však vyplývá, že data obsahují převážně vazby 1:N a 1:1. Dále se doporučuje využít grafové databáze v případě, že vztahy jsou stejně nebo více důležité než entity. Tuto vlastnost také data z knih nemají. Z těchto důvodů byly grafové databáze vyhodnoceny jako nevyhovující pro implementaci datového úložiště.

2.1.2.2 Databáze typu klíč-hodnota

Databázové systémy typu klíč-hodnota (*key/value stores*) jsou velice jednoduché. Lze si je představit třeba jako asociativní pole nebo jako hašovací tabulku. Data jsou jednoduše ukládána v páru, kdy první hodnota v páru je klíč a druhá je příslušná hodnota k danému klíči. V tomto typu databáze můžeme uchovávat prakticky jakékoliv datové typy. Databázový systém primárně poskytuje vyhledávání uložené hodnoty podle klíče, neposkytuje však efektivní vyhledávání podle hodnoty. [13]

[9] Resource Description Framework

Struktura dat v databázích typu klíč-hodnota je velmi jednoduchá a spolu se základními operacemi systémů poskytuje pohodlnou a efektivní práci s daty. API těchto systémů poskytují většinou tři základní operace. První z nich je operace na vložení hodnoty pro příslušný klíč (operace `PUT`). Další operací je vyhledání uložené hodnoty pro příslušný klíč (operace `GET`). Poslední základní operací je smazání klíče a jeho hodnoty z databáze (operace `DELETE`). Toto však nejsou jediné operace, které databázové systémy typu klíč-hodnota poskytují. Mnoho těchto systémů nabízí širokou škálu operací, které usnadňují práci s daty.

Některé z nás určitě napadne, zda je možné ukládat k jednomu klíči více různých typů hodnot. Většina databázových systémů typu klíč-hodnota umožňuje rozdělit data do jednotlivých přihrádek podle jejich typu. Tyto přihrádky oddělují záznamy různých typů a jsou stále uloženy fyzicky v jednom hašovacím prostoru. Přihrádky představují tzv. jmenné prostory (*namespaces*). V následující tabulce je uveden příklad uložení dat do jednoho jmenného prostoru. [8] U každého knižního díla potřebujeme uložit hlavičku (metadata o knize), textový obsah knihy a komentáře. Tyto informace můžeme uložit v jednom jmenném prostoru (např. `Namespace Dilo`).

Tabulka 2.2: Příklad uložení dat z díla v jednom jmenném prostoru

Namespace Dilo

Klíč:	bookID
Hodnota:	hlavicka text komentare

Většina databázových systémů typu klíč-hodnota může pracovat v distribuovaném režimu nebo v lokálním režimu. V distribuovaném režimu jsou data rozdělena mezi více uzlů distribuovaného systému. Který uzel bude uchovávat konkrétní dvojici (klíč, hodnota), závisí buď přímo na konkrétním klíči nebo na výstupu hašovací funkce[10] $h(klic)$. V lokálním režimu jsou data uložena přímo v lokálním datovém úložišti. Pro tento způsob existuje řada knihoven např. LevelDB nebo RocksDB.

Jedním z nejznámějších databázových systémů typu klíč-hodnota je systém Redis[11]. Jedná se o velmi flexibilní systém, který lze velmi dobře škálovat. Redis je možné používat na jediném stroji, kdy jsou data uložena pouze v operační paměti, ale také je možné nakonfigurovat systém tak, aby byla data rozložena mezi více strojů (tzv. sharding). Dále je možné použít architekturu

[10] Hašovací funkce h zpracovává neomezeně dlouhá vstupní data M a vrací výstup $h(M)$ představující hašový kód. Musí být jednosměrná (z M lze jednoduše spočítat $h(M)$, obráceně je to výpočetně náročné) a bezkolizní (je výpočetně náročné nalézt dva vstupní texty, které mají stejné hašové kódy a také je výpočetně náročné nalézt ke konkrétnímu textu jiný text se stejným hašovým kódem). [14]

[11] Remote Dictionary Server

Master-Slave, kdy Redis replikuje data na pozadí mezi uzly typu Master a uzly typu Slave. Redis je v dnešní době hodně využívaný i díky tomu, že k němu lze přistupovat z mnoha programovacích jazyků, mezi které patří známé jazyky jako C, C++, Java, Python, Javascript, Go, Lua a spousta dalších. Jednou z jeho největších výhod je možnost práce s komplexnějšími datovými strukturami, jako je například seznam (`list`), množina (`set`) nebo asociativní pole (`hash`), a nad těmito strukturami lze provádět i složitější operace. [15]

Jak již bylo řečeno, každé úložiště typu klíč-hodnota poskytuje základní tři operace pro vkládání, vyhledávání a mazání hodnot pro příslušné klíče. Ani v případě Redisu tomu není jinak. V následující ukázce je zobrazen příklad použití základních operací, kdy nejprve je znázorněn pokus získání hodnoty neexistujícího klíče, následně je uložena hodnota k danému klíči, poté je tato hodnota získána a nakonec je znázorněno vymazání hodnot k zadaným klíčům. Časové složitosti jednotlivých operací jsou znázorněny v tabulce pod ukázkou.

redis> **GET autor**

výstup: (nil)

redis> **SET autor “Jan Neruda“**

výstup: “OK“

redis> **GET autor**

výstup: “Jan Neruda“

redis> **SET titul “Hrbitovni kviti“**

výstup: “OK“

redis> **DEL autor titul**

výstup: (integer) 2

redis> **GET autor**

výstup: (nil)

redis> **GET titul**

výstup: (nil)

Tabulka 2.3: Časové složitosti základních operací v systému Redis [4]

Operace	**Časová složitost**	**Doplnění**
GET	$\mathcal{O}(1)$	
SET	$\mathcal{O}(1)$	
DEL	$\mathcal{O}(N)$	N je počet mazaných klíčů

Stejně jako u grafových databází je nutné zmínit, že databázové systémy typu klíč-hodnota nejsou vhodné pro každá data. Následující tabulka obsahuje výhody a nevýhody v souvislosti s využitím databází typu klíč-hodnota pro implementaci úložiště digitalizovaných knih.

Tabulka 2.4: Využití databází typu klíč-hodnota pro potřeby řešení ÚČL

K-V DB	Výhody	Nevýhody
	• jednoduchost a výkonnost • rychlost nezávisí na množství uložených dat	• k datům se nelze dostat jinak než přes znalost jejich klíče • žádná možnost transakčního zpracování či hlídání integrity dat

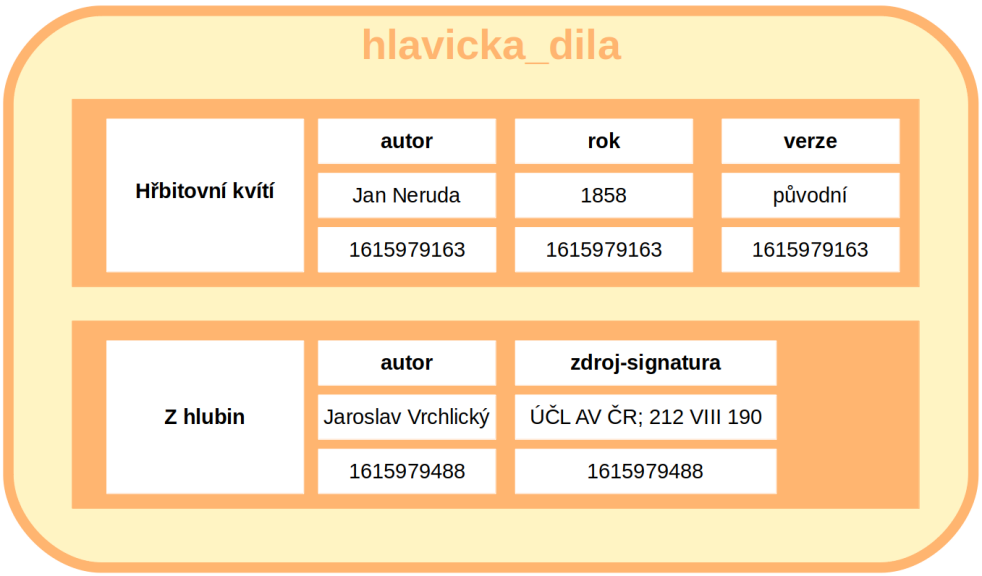
Pokud bychom využili databázové systémy typu klíč hodnota pro uložení knih, pak by rychlost odezvy jednotlivých dotazů byla stabilní bez ohledu na to, kolik dat je v databázi uloženo. Pro uložení digitalizovaných děl ÚČL je však takové úložiště nevyhovující. Databáze typu klíč-hodnota nenabízí žádnou možnost kontroly konzistence a integrity dat. Dále je velmi obtížné provádět projekci dat a také není možné provést dotaz, který najde a vrátí data jednoho konkrétního typu. Z těchto důvodů tento typ databázových systémů nebyl použit pro implementaci datového úložiště.

2.1.2.3 Sloupcové databáze

Dalším důležitým typem NoSQL databází jsou sloupcové databáze, které jsou také často označovány jako „column family stores“, „wide column stores“ nebo „columnar stores“. Datový model sloupcových databází se skládá z řádků (*row*), které jsou identifikované klíčem řádku (*row key*). Každý řádek může mít velmi mnoho sloupců, které mají svůj název, hodnotu a také časové razítko (*timestamp*) obsahující datum a čas uložení hodnoty. Jednotlivé sloupce dohromady tvoří skupinu sloupců (*column families*). Důležitou vlastností je, že jednotlivé řádky mohou v rámci skupiny sloupců vytvářet libovolné množství různých sloupců s různými názvy. [8]

Na následujícím obrázku je zobrazena zjednodušená skupina sloupců pro hlavičkové údaje z knih. V prvním sloupečku je obsažen klíč řádku, v dalších pak jednotlivá data z knih. Skupina sloupců je velmi podobná relační tabulce s tím rozdílem, že každý řádek může mít odlišný počet sloupců s odlišnými názvy. Zde například kniha s názvem Hřbitovní kvítí má sloupce obsahující informaci o autorovi, roku vydání a verzi vydání. Druhá kniha Z hlubin neobsahuje sloupec s rokem a verzí vydání a naopak obsahuje navíc informaci o signatuře zdroje. Na obrázku nechybí ani časové známky u jednotlivých dat.

Obrázek 2.4: Ukázka skupiny sloupců



Dále mohou některé sloupcové databáze vytvářet tzv. supersloupce (*super columns*), které se skládají z více jednoduchých sloupců. Do předchozího obrázku můžeme přidat například supersloupce pro údaje o autorovi a pro údaje o vydání knihy. Supersloupec pro údaje o autorovi následně může obsahovat sloupce pro jméno autora, rok narození nebo jeho rodné město. Supersloupec pro údaje o vydání knihy by mohl obsahovat například sloupce pro nakladatelství, rok vydání nebo informace o tom, o kolikáté vydání se jednalo. [16]

Jedním z nejznámějších systémů spadajících do kategorie sloupcových databází je systémy Cassandra. Apache Cassandra je open source systém, který se hodí do velkých projektů, jelikož je navržený pro práci s velkým objemem dat. Systém je implementovaný v Javě a má svůj vlastní dotazovací jazyk CQL[12]. V souladu s datovým modelem sloupců umožňuje implementace systému Cassandra vytvářet v podstatě neomezený počet sloupců pro každý řádek, což je velká odlišnost oproti klasickým relačním databázím. Další výhodou systému je fyzické ukládání hodnot sloupců blízko sebe (fyzická kolokace dat) a distribuovaná architektura systému. [8][17]

Od verze 1.2 odpovídal přístup pohledu na data jako na multidimenzionální asociativní pole. V současné verzi 3.4.5 však tento model není doporučovaný a nejspíše v budoucnu nebude podporován. Současná interpretace datového modelu vnímá jednotlivé skupiny sloupců jako tabulky. Celý jazyk CQL vychází ze známého jazyka SQL. Základní definice dat vypadá například takto:

[12]Cassandra query language

```
CREATE TABLE hlavicka_knihy(
    titul text PRIMARY KEY,
    autor text,
    rok int,
    verze list<text>,
    komentare map<int, text>
);
```

V ukázce definice dat si můžeme všimnout, že Cassandra umožňuje vytvářet sloupce typů kolekce. Zejména se jedná o kolekce typu seznam (`list`), množina (`set`) a asociativní pole (`map`). V následující ukázce je zobrazeno jednoduché vkládání dat pomocí CQL.

```
INSERT INTO books (titul, rok, verze, komentare)
    VALUES (
        'Hřbitovní kvítí',
        1858,
        ['původní', 'nezměněná'],
        {
            123 : 'text komentáře 124',
            456 : 'text komentáře 456'
        }
    );
```

Při definici dat byl vytvořen sloupec `autor`. V klasických relačních databázích by se při chybějící hodnotě při vkládání dat fyzicky uložila do databáze hodnota `null`. V systému Cassandra tomu tak není. V databázi není uložena hodnota žádná, avšak ve výpisu dat by se hodnota `null` objevila.

Následují další ukázky manipulace dat pomocí CQL, konkrétně se jedná o změnu hodnoty v databázi a selekci dat z databáze podle určitého pravidla. Zajímavostí je to, že Cassandra neumožňuje použít vnořený dotaz typu SELECT. [18]

```
UPDATE books
    SET komentare = {
        '123' : 'komentar 123'
    }
    WHERE titul = 'Hřbitovní kvítí';

SELECT autor, rok FROM books
 WHERE rok >= 1800
   AND rok < 1860;
```

Nyní je třeba promyslet, zda jsou sloupcové databáze vhodné pro implementaci úložiště, které bude uchovávat sbírky ÚČL. Tak jako v předchozích

sekcích i v této části je třeba shrnout výhody a nevýhody použití sloupcových databází.

Tabulka 2.5: Využití sloupcových databází pro potřeby řešení ÚČL

Sl. DB	Výhody	Nevýhody
	• ideální pro strukturu dat knih • snadná práce s daty	• obtížné definování schématu databáze

Na rozdíl od grafových databází a databází typu klíč-hodnota jsou sloupcové databáze vhodné pro strukturu dat digitalizovaných knih. Každá kniha může mít trochu odlišnou strukturu. Ve sloupcových databázích může mít také každý řádek trochu jinou strukturu, jelikož není žádné omezení na počty ani názvy sloupců. Z tohoto pohledu se sloupcové databáze jeví jako ideální. Další výhodou je velmi snadná manipulace s daty, jak jsme si mohli všimnout u systému Cassandra. Jediná nevýhoda se týká definice schématu. Většina sloupcových databázových systémů vyžaduje definování společného datového modelu obdobně jako u systému Cassandra. To by mohl být problém u digitalizovaných knih ÚČL vzhledem k jejich různorodosti. To však není nedostatek, který by znemožňoval využít sloupcové databáze pro uložení knih. Proto sloupcové databáze představují jednu z možných variant pro implementaci datového úložiště.

2.1.2.4 Dokumentové databáze

Posledním typem NoSQL databází, který si v této kapitole představíme, jsou dokumentové databáze. „Jak název napovídá, pro tyto systémy je klíčový koncept dokumentu, jenž v daném kontextu znamená datovou strukturu, která má samopopisný charakter, tedy obsahuje kromě samotných dat i metadata popisující význam jednotlivých částí datové struktury.“ [8] Mezi příklady dokumentů patří neodmyslitelně formát JSON[13] nebo třeba XML. Tyto formáty představují stromovou datovou strukturu, která obsahuje asociativní pole, seznamy a základní datové typy. Samotné dokumenty jsou hlavně používané pro ukládání dat, nicméně některé aplikace je využívají i jako prostředek pro komunikaci. [13]

Při použití relačních databází také často dochází k používání těchto formátů. V první řadě je využíváno tzv. ORM[14] pro ukládání paměťových datových struktur. V druhé řadě je potřeba převádět relační data na JSON pro snazší přenesení mezi jednotlivými komponentami aplikací. Z toho vyplývá výhoda pro dokumentové databáze. Při použití dokumentových databází je možné uložená data přímo použít pro komunikaci a nemusíme je převádět,

[13] JavaScript Object Notation
[14] Objektově relační mapování

jako tomu je v případě relačních databází. Formáty dokumentů jsou velmi podobné struktuře tříd objektového programování. Proto je velmi jednodušší jejich vzájemná konverze. V tomto případě mluvíme o tzv. ODM[15].

Jednou z největší výhod dokumentových databází je volnost datového modelu. Do databáze můžeme ukládat různorodé dokumenty, jejichž struktura se může více či méně lišit. Tyto dokumenty mohou být uloženy v jedné kolekci dokumentů v databázi. Následně je možné strukturu dokumentů jakkoliv měnit, aniž bychom museli měnit schéma databáze.

V dokumentových databázích máme k dispozici dva základní přístupy, jak ukládat data do kolekcí. První přístup představuje použití vnořených objektů (*embedded documents*). Tento přístup umožňuje snadnou manipulaci se všemi daty. Je vhodný v případě, že modelujeme vztahy 1:1 nebo 1:N, tedy že dokument obsahuje pouze jeden vnořený dokument a nebo že vnořené dokumenty mají pouze jeden nadřazený dokument. Naopak pro vztahy M:N je tento dokument nevhodný. Nevýhodou tohoto konceptu je, že pokud vkládáme do nadřazeného dokumentu příliš mnoho vnořených záznamů, může být čtení a zápis výrazně pomalejší. V následující ukázce je uveden příklad použití vnořených dokumentů. Zde si můžeme všimnout hodnoty `<ObjectId_A>` se jménem `_id`, která představuje primární klíč dokumentu v kolekci.

```
kolekce "knihy"

{
    "_id": <ObjectId_A>,
    "hlavicka": {
        "autor": "Jan neruda",
        "titul": "Hřbitovní kvítí"
    },
    "text": {
        "sbirka": [],
        "komentar": "text"
    },
    "poznamka": "neuvedena"
}
```

Naopak druhý přístup upřednostňuje použití odkazů. Odkazy jsou realizovány pomocí primárních klíčů `_id` a představují obdobu cizích klíčů v relačních databázích. Tento koncept je oproti předchozímu vhodný i pro vztahy M:N. V následující ukázce je uvedeno uložení předchozího dokumentu pomocí konceptu odkazů.

```
kolekce "knihy"
```

[15] Objektově dokumentové mapování

```
{
    "_id": <ObjectId_A>,
    "poznamka": "neuvedena"
}

kolekce "hlavicky"

{
    "_id": <ObjectId_B>,
    "kniha_id": <ObjectId_A>,
    "autor": "Jan neruda",
    "titul": "Hřbitovní kvítí"
}

kolekce "texty"

{
    "_id": <ObjectId_C>,
    "kniha_id": <ObjectId_A>,
    "sbirka": [],
    "komentar": "text"
}
```

Tato varianta odpovídá normalizovanému schématu dat v klasických relačních databázích. Výhodou tohoto konceptu je, že zabraňuje vytváření duplicitních dat. Nevýhodou je naopak složitější získávání záznamů obsahujících příliš mnoho vazeb. Dokumentové databáze totiž většinou neumožňují propojení více záznamů v rámci jednoho dotazu, a je tedy potřeba dotázat se vícekrát. [8]

Jednou z klíčových vlastností dokumentových databází je vytváření a používání vyhledávacích indexů. Stejně jako v relačních databázích, i zde se indexy používají pro efektivnější vyhodnocování dotazů. Pokud pro danou kolekci a její výběr není vytvořen vhodný index, pak je třeba projít všechny dokumenty v databázi sekvenčně, což může být značně neefektivní. Indexy jsou velmi často implementovány pomocí B-stromů. B-stromy představují datovou strukturu vyhledávacích stromů. [19] Od většiny ostatních vyhledávacích stromů se však liší tím, že v jejich uzlech jde uložit více než jeden prvek. Vlastnosti B-stromů jsou následující:

- maximální počet prvků v uzlu r (kapacita) je stejný a volíme ho vždy na začátku
- jednotlivé uzly musí být alespoň z poloviny zaplněné a zároveň nesmí překročit r

- prvky uložené v uzlu jsou seřazeny vzestupně
- uzel je list nebo má o jednoho více následníků, než je počet jeho prvků
- následník obsahuje prvky větší než jeho levá hodnota v nadřazeném uzlu a menší než jeho pravá hodnota v nadřazeném uzlu
- listy jsou v B-stromu pouze na jeho poslední hladině. [20]

Vyhledávání hodnoty x v B-stromu za předpokladu, že jsme v uzlu u funguje následovně:

1. pokud byla hodnota x v u nalezena, pak vyhledávání úspěšně končí
2. pokud hodnota x nebyla nalezena v u a u představuje list, pak vyhledávání končí neúspěšně
3. pokud hodnota x nebyla nalezena v u a u nepředstavuje list, pak změníme u na následníka, který by měl prvek obsahovat a pokračujeme krokem 1. [20]

Jednou z nejznámějších dokumentových databází je MongoDB. Tomuto databázovému systému se však práce věnuje v samostatné kapitole (viz 2.3). Pojďme se nyní podívat na výhody a nevýhody použití dokumentových databází pro implementaci datového úložiště pro potřeby ÚČL.

Tabulka 2.6: Využití dokumentových databází pro potřeby řešení ÚČL

Dok. DB	**Výhody**	**Nevýhody**
	• ideální pro strukturu dat knih • volnost datového modelu • knihy jsou ve formátu XML	• pomalejší práce s daty v případě vnořených dokumentů

Z analýzy vyplývá, že dokumentové databáze jsou vhodné pro strukturu dat digitalizovaných knih, jelikož každá kniha má poněkud odlišnou strukturu a dokumentové databáze nabízejí ukládání dokumentů s libovolnou strukturou. Další výhodou je skutečnost, že digitalizované knihy mají formát XML dokumentů, proto se přímo nabízí využít dokumentové databáze. Jedinou nevýhodou je koncept ukládání knih. Ty představují koncept vnořených dokumentů, nad kterými bude náročnější provádět komplexnější dotazy. Stejně jako sloupcové databáze jsou dokumentové databáze vhodné pro implementaci datového úložiště pro digitalizované knihy ÚČL.

2.1.3 Shrnutí NoSQL databází

V předchozí sekci byly popsány jednotlivé typy NoSQL databází. Konkrétně se jednalo o databáze grafové, sloupcové, dokumentové a databáze typu klíč-hodnota. Každý z těchto typů NoSQL databází je vhodný pro data jiné struktury. Grafové databáze se hodí pro ukládání grafových struktur, ve kterých je příliš velké množství vztahů. Databáze typu klíč-hodnota jsou vhodné pro data, ve kterých potřebujeme vyhledávat hodnoty podle jejich klíče a ve kterých není příliš mnoho provázaností. Sloupcové databáze je dobré využít v případě, že je potřeba uložit velký objem dat, se kterým chceme následně efektivně pracovat. Dokumentové databáze se hodí, pokud máme data různé struktury, která se v čase ještě může libovolně měnit.

Z analýzy vyplývá, že pro implementaci datového úložiště pro potřeby ÚČL nejsou vhodné grafové databáze a databáze typu klíč-hodnota. Naopak je možné použít sloupcové a dokumentové databáze. Na základě konzultace s vedoucím práce bylo rozhodnuto, že k implementaci datového úložiště bude využita dokumentová databáze konkrétně databázový systém MongoDB. MongoDB používá dokumenty podobné formátu JSON (formát BSON[16]), do kterého lze snadno převést všechny digitalizované knihy ve formátu XML, jimiž disponuje ÚČL.

2.2 Datové formáty

V této sekci jsou přiblíženy datové formáty, se kterými v této práci budeme pracovat. V první řadě se jedná o formát XML, ve kterém jsou uložena veškerá digitalizovaná díla ÚČL. Jelikož byla pro implementaci datového úložiště vybrána databáze MongoDB, je třeba zmínit formáty JSON a BSON. Také je třeba zmínit formát TEI[17], který ÚČL využívá pro popis knih.

2.2.1 XML

Extensible Markup Language je obecný značkovací jazyk, který vyvinulo a standardizovalo konsorcium W3C[18]. Jedná se o zjednodušenou podobu staršího jazyka SGML[19], který umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé typy dat. V současné době se hojně využívá pro serializaci dat obdobně jako formát JSON. Zpracování XML je podporováno řadou nástrojů a programovacích jazyků. Existují dva nejčastější přístupy ke zpracování XML dokumentu. Prvním je DOM[20] parser, který z XML dokumentu vyrobí obraz v paměti. Druhým je SAX[21] parser, který postupně prochází XML dokument

[16] Binární JSON
[17] Text Encoding Initiative
[18] World Wide Web Consortium
[19] Standard Generalized Markup Language
[20] Document Object Model
[21] Simple API for XML

a vyvolává události, které jsou následně zpracovávány uživatelem. [21]

Samotný princip XML je velmi jednoduchý. Každý XML dokument se skládá z navzájem do sebe vnořených elementů. Samotné elementy se vyznačují pomocí tagů. Téměř každý element má počáteční a ukončovací tag. Názvy tagů se zapisují mezi znaky '<' a '>', ukončovací tag má na začátku '</'. Kromě elementů mohou XML dokumenty obsahovat i atributy. Nechybí zde ani komentáře. V následující ukázce je uveden příklad komentáře, elementu a jeho atributu:

```
<!-- Příklad elementu -->
<autor id=1234>Jan Neruda</autor>
```

2.2.1.1 XML schéma

XML schéma je dokument, pomocí kterého lze přesně definovat, jaké elementy a atributy se v rámci dokumentu mohou vyskytovat, jaké datové typy představují obsahy elementů a případně v jakém pořadí a kolikrát se mohou elementy vyskytovat. Schéma můžeme využít pro validaci XML dokumentů, tedy kontrolu, zda dokument obsahuje všechny potřebné náležitosti popsané ve schématu. [22]

Mezi úplně nejznámější jazyky pro tvorbu XML schémat patří W3C XML Schema. Tento jazyk používá pro svůj zápis přímo formát XML. Všechny elementy ve schématu musí patřit do jmenného prostoru, který obvykle používá prefixy `xs` a `xsd` (jmenný prostor `http://www.w3.org/2001/XMLSchema`). Celé schéma musí být vždy uzavřeno v elementu `schema` obsahujícím definice jednotlivých elementů, které je možné použít jako kořenové elementy. Každý element musí mít definovaný svůj datový typ. Rozlišujeme dva druhy datových typů, kterými jsou jednoduchý a komplexní datový typ. Jednoduché datové typy se používají pro hodnoty obsahující například řetězec, číslo, logickou hodnotu a další. Komplexní datové typy (`complexType`) slouží k modelování struktury dokumentu, protože se mohou skládat z elementů a atributů. U elementů můžeme určit v jakém pořadí se mají vyskytovat, kolikrát se mohou opakovat, zda jsou povinné či volitelné. Komplexní datové typy vytvoříme pomocí elementu `sequence`. [23]

V následující ukázce je zobrazen příklad XML schématu, které definuje, že dokument musí obsahovat element `kniha` s atributem obsahující identifikátor `id` datového typu `integer`. Tento element představuje komplexní datový typ skládající se z dalších podelementů. První z těchto podelementů představuje autora knihy (`autor`) a je datového typu `string`. Dále je zde název knihy (`titul`), který je taktéž datového typu `string` stejně jako místo vydání (`mistovydani`). U roku vydání (`rokvydani`) očekáváme celočíselnou hodnotu, proto je datového typu `integer`. Poslední podelement představuje datum poznámky v knize (`datumpoznamky`) a je datového typu `date`. Při validaci XML

dokumentů pomocí tohoto schématu musí mít každý z těchto podelementů odpovídající datový typ.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="kniha">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="autor" type="xs:string"/>
        <xs:element name="titul" type="xs:string"/>
        <xs:element name="rokvydani" type="xs:integer"/>
        <xs:element name="mistovydani" type="xs:string"/>
        <xs:element name="datumpoznamky" type="xs:date"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2.2.1.2 XPath

XML dokument je mnohdy chápán jako stromová struktura, kde elementy a atributy představují jednotlivé uzly. Pro hledání a vybírání jednotlivých částí XML dokumentu slouží jazyk XPath[22]. Výrazy XPath jsou velmi podobné zápisu cest v adresářové struktuře. Pomocí lomítka oddělíme jednu úroveň v XML stromu. Pokud nám na hloubce vnoření nezáleží, použijeme dvě lomítka bezprostředně za sebou. Aktuální uzel stromu lze zapsat pomocí tečky, nadřazenou úroveň pomocí dvou teček. Pro vyhledávání atributů je třeba použít symbol '`@`' před jménem atributu. Podmínky lze zapsat za výraz do hranatých závorek.

Pokud bychom měli XML dokumenty splňující schéma uvedené v poslední ukázce předchozí sekce (viz 2.2.1.1), pak bychom například autora knihy získali pomocí výrazu `/kniha/autor` případně `//autor`. Pokud bychom chtěli vybrat knihu s `id = 5`, můžeme použít výraz `/kniha[@id = 5]`.

Další důležitou součást jazyka XPath představují identifikátory osy. Ty určují směr procházení XML dokumentu. Pokud identifikátor osy neuvedeme, použije se identifikátor `child::` a jsou prohledáváni přímí potomci aktuálního uzlu. Celkem je v jazyce XPath definováno 13 identifikátorů os. Jednotlivé identifikátory jsou uvedeny v následující tabulce:

[22] XML Path Language

Tabulka 2.7: Identifikátory os v jazyce XPath [5]

Identifikátor	Směr procházení uzlů
`child::`	přímí potomci aktuálního uzlu
`descendand::`	všichni potomci aktuálního uzlu
`descendand-or-self::`	aktuální uzel a všichni potomci
`self::`	aktuální uzel
`ancestor-or-self::`	aktuální uzel a jeho předci
`ancestor::`	všichni předci aktuálního uzlu
`parent::`	rodič aktuálního uzlu
`following::`	všechny uzly za aktuálním
`preceding::`	všechny uzly před aktuálním
`following-sibling::`	následující sourozenci aktuálního uzlu
`preceding-sibling::`	předcházející sourozenci aktuálního uzlu
`attribute::`	atributy aktuálního uzlu
`namespace::`	jmenné prostory

2.2.2 JSON

Formát JSON je v dnešní době jeden z nejvyužívanějších formátů pro ukládání dat a pro výměnu dat mezi různými systémy. Oproti formátu XML je dobře čitelný a zapisovatelný i pro člověka. Jedná se o podmnožinu programovacího jazyka Javascript, která umožňuje definovat a reprezentovat základní datové struktury. Zápis JSON formátu je tedy platným zápisem jazyka Javascript, což je jedna z jeho výhod.

JSON nabízí šest základních datových typů:

- **JSON Object** - složený objekt
- **JSON Array** - pole
- **JSON String** - textový řetězec
- **JSON Number** - číslo
- **JSON Boolean** - logická hodnota
- **JSON Null** - hodnota null.

JSON objekt není ekvivalentní objektu v Javascriptu a obsahuje pouze data, nenajdeme v něm žádné metody. Jedná se tedy spíše o hash nebo asociativní pole. Každá datová položka má svůj klíč a svou hodnotu. Klíč je vždy typu JSON String. Hodnota může představovat jakýkoliv ze šesti základních datových typů, tedy i objekt. Objekt tedy může obsahovat libovolně složitou a zanořenou strukturu. Hloubku zanoření lze omezit pouze konkrétní implementací Javascriptu nebo JSON parseru.

JSON Array je pole obsahující hodnoty jakéhokoliv ze šesti základních datových typů. Ohraničují jej hranaté závorky. JSON String představuje řetězec, který musí být vložen do uvozovek a může obsahovat veškeré znaky Unicode. V případě vložení uvozovek nebo zpětného lomítka do stringu je třeba před tyto symboly přidat zpětné lomítko. JSON Number je datový typ pro uložení čísla. Číslo může být v libovolném formátu, může být zapsané i pomocí exponentu. Desetinnou čárku je potřeba zapsat pomocí tečky. JSON Boolean může nabývat pouze hodnot `true` a `false`. JSON Null představuje standardní `null` hodnotu. [24]

```
{
    "dilo": {
        "hlavicka": {
            "autor": "Jan Neruda",
            "titul": "Hřbitovní kvítí",
            "rok": 1858,
            "moto": null
        },
        "text": {
            "basne": [["sloka1", "sloka2"], {"sloka": "sloka"}],
            "naskenovany": true
        }
    }
}
```

V ukázce výše si můžeme všimnout JSON objektu `dilo`, který má v sobě další vnořené objekty `hlavicka` a `text`. Hlavička obsahuje datové položky `autor` a `titul` typu JSON String, `rok` typu JSON Number a `moto` není definováno, proto je typu JSON Null. Objekt `text` obsahuje pole `basne` obsahující další pole JSON Stringů a objekt. Dále obsahuje položku `naskenovany` datového typu JSON Boolean. [25]

2.2.2.1 JSON Schema

Mnohdy nám nestačí samotné JSON dokumenty a potřebovali bychom popis struktury dokumentů například pro jejich validaci. K tomu nám může posloužit JSON Schema, tedy jazyk pro zápis schémat pro dokumenty ve formátu JSON. JSON schéma umožňuje detailně popsat strukturu dokumentů a lze vůči němu validovat jakýkoliv dokument JSON a zjistit, zda vyhovuje danému schématu.

Samotné schéma je JSON dokument reprezentující objekt. Obsahuje položku `$schema`, která určuje, o jakou verzi JSON Schema se jedná. Pomocí klíčového slova `type` můžeme u datové položky specifikovat, o jaký datový typ se jedná. Lze použít stejné datové typy jako pro JSON formát, navíc můžeme použít například datový typ `integer` pro celočíselné hodnoty.

Pokud však zvolíme jako datový typ objekt, pak je třeba definovat jeho strukturu v položce `properties`. V `properties` můžeme uvést volitelné datové položky, ale musíme uvést povinné. Povinné datové položky určíme pomocí klíčového slova `required`. Dále je možné určit počet datových položek v objektech pomocí `minProperties` a `maxProperties`. Pokud je existence některé z datových položek závislá na jiné položce, pak ji lze vynutit pomocí klíčového slova `dependencies`. V případě potřeby definovat názvy datových položek v objektu můžeme použít `propertyNames` nebo `patternProperties` v případě svázaní daného formátu s konkrétními vlastnostmi datové položky.

Obdobně u datového typu pole musíme definovat jeho položky pomocí klíčového slova `items`. V položce `items` lze definovat společné vlastnosti pro všechny prvky pole, ale i pro jednotlivé prvky zvlášť. V případě definování vlastností pro prvky zvlášť lze použít klíčové slovo `additionalItems`, kterým lze popsat vlastnosti dalších prvků, případně lze výskyt dalších prvků zakázat pomocí hodnoty `false`. Dále lze omezit velikost pole pomocí položek `minItems` a `maxItems` a také lze vynutit unikátnost prvků s `uniqueItems`.

U řetězců můžeme definovat jejich délku obdobně jako u polí, akorát s použitím `minLength` a `maxLength`, a také můžeme definovat odpovídající formát řetězce pomocí klíčového slova `format` (např. datum) nebo `pattern` (regulární výraz). U čísel můžeme definovat, zda jsou násobkem jiného čísla položkou `multipleOf` a také můžeme definovat jejich rozsah pomocí `minimum`, `maximum`, `exclusiveMaximum` a `exclusiveMinimum`. U logických hodnot žádné speciální vlastnosti definovat nelze, stejně tak u hodnoty `null`.

V praxi může nastat situace, kdy máme pro jeden typ dokumentů více různých schémat a chtěli bychom je kombinovat. K tomu můžeme použít klíčová slova `anyOf`, `oneOf` a `allOf`. Do těchto položek můžeme vložit pole s jednotlivými schématy. Klíčové slovo `anyOf` můžeme použít v případě, pokud dokument musí být validní vůči alespoň jednomu ze schémat v poli. V případě `oneOf` musí být dokument validní vůči přesně jednomu ze schémat v poli. Pokud však použijeme `allOf`, pak musí být dokument validní vůči všem schématům v poli. Pokud bychom naopak chtěli zajistit, že dokument není validní vůči nějakému schématu, pak můžeme použít klíčové slovo `not`. Dále je možné validovat dokumenty vůči různým schématům pouze za určité podmínky. K tomu lze využít známý koncept klíčových slov `if`, `then`, `else`.

Velmi důležité je klíčové slovo `definitions`, pomocí kterého lze definovat schéma jednotlivých objektů a následně je možné tyto objekty použít vícekrát. V takovém případě nemusíme pokaždé psát celé schéma a můžeme odkázat na definici uvedenou v položce `definitions` a schéma přepoužít.

Poslední věcí, kterou je třeba zmínit, jsou klíčová slova `title`, `default`, `examples` a `description`. Tato slova neslouží k validaci, nýbrž k popisu jednotlivých částí schématu. Klíčové slovo `title` představuje název části schématu. Tento název by měl být relativně krátký. Naopak `description` může být delší a slouží k podrobnému popisu dané části. Obě tyto položky mohou obsahovat pouze řetězec. Klíčové slovo `default` resp. `examples` slouží pro

specifikaci výchozích resp. ukázkových hodnot dané položky.

JSON Schema však nabízí daleko více. Navíc s novějšími verzemi přibývají další nové možnosti. V této sekci byly uvedeny pouze základní vlastnosti podporované do verze JSON Schema 7.0. [26] V následující ukázce je uveden příklad jednoduchého JSON schématu:

```
{
    "\$schema": "https://json-schema.org/draft/2020-12/schema",
    "title": "Dílo",
    "type": "object",
    "required": ["hlavicka"],
    "properties": {
        "hlavicka": {
            "type": "object",
            "description": "Hlavičkové údaje knihy",
            "required": ["titul", "rok"],
            "properties": {
                "titul": {
                    "type": "string",
                    "minLength": "3",
                    "description": "Řetězec minimální délky 3",
                },
                "rok": {
                    "type": "integer",
                    "minimum": 1800,
                    "maximum": 1900,
                    "description": "číslo v rozmezí 1800-1900 ",
                }
            }
        }
    }
}
```

2.2.3 BSON

BSON je zkratka pro Binary JSON a představuje binárně kódovanou serializaci dokumentů, které jsou podobné JSON dokumentům. Obdobně jako JSON podporuje BSON vnořené dokumenty a také pole obsahující dokumenty a další pole. Je tedy možné vytvářet libovolně zanořené dokumenty. BSON obsahuje rozšíření umožňující reprezentovat datové typy, které v JSON specifikaci nenajdeme. Příkladem mohou být datové typy pro datum nebo pro binární data. V porovnání s jinými binárními formáty, jako je například *Protocol Buffers*, nemá BSON explicitně definované schéma, díky čemuž je daleko flexibilnější. BSON formát byl navržen tak, aby měl následující tři vlastnosti:

1. **Odlehčenost (lightweight)** - snížení prostorových režijních nákladů na minimum
2. **Snadné procházení (traversable)** - formát lze velmi snadno procházet, což je velmi důležitá vlastnost pro databázový systém MongoDB
3. **Efektivnost (efficient)** - kódování a dekódování dat do/z BSON formátu je velmi rychlé a efektivní v mnoha programovacích jazycích, například v C.

Hodnoty ve formátu BSON jsou ukládány jako seřazené dvojice (klíč, hodnota). Všechny tyto dvojice dohromady tvoří dokument. BSON standard je popsán pomocí syntaxe bezkontextové gramatiky v pseudo-BNF[23]. Terminály této bezkontextové gramatiky jsou všechny základní typy. Každý z těchto typů musí být serializován ve formátu little-endian. Konkrétně se jedná o formáty `byte`, `int32`, `int64`, `uint64`, `double` a `decimal128`. Ostatní datové formáty představují neterminály gramatiky. Ve specifikaci BSON lze najít 37 datových formátů, mezi které patří například dokument (`"\x00"`), 64-bitový double (`"\x01"`), UTF-8[24] string (`"\x02"`), vnořený dokument (`"\x03"`), pole (`"\x04"`), binární data (`"\x05"`), ObjectID (`"\x07"`), UTC[25] datový formát (`"\x09"`) nebo třeba Javascript kód (`"\x0D"`). [27]

```
{
    "autor": "Jan Neruda",
    "titul": ["Hřbitovní", "kvítí"],
    "rok": 1858
}
```

```
\x52\x00\x00\x00
\x02autor\x00\x0b\x00\x00\x00Jan Neruda\x00
\x04titul\x00\x27\x00\x00\x00
\x02\x30\x00 \x0c \x00\x00\x00Hřbitovní\x00
\x02\x31\x00\x08\x00\x00\x00kvítí\x00\x00
\x10rok\x00\x42\x07\x00\x00
\x00
```

Předchozí ukázka zachycuje porovnání JSON a BSON formátu. Uvedená data v JSON formátu byla následně převedena do BSON formátu. První bajt zapsaný hexadecimálně `\x52` obsahuje velikost celého dokumentu, tedy přesně 82 bajtů. Následuje bajt `\x02`, který značí, že následující datová položka je typu `string`. Za datovým typem je samotný klíč, počet bajtů obsahující hodnotu (včetně `\x00` na konci) a samotná hodnota. Další datovou položkou

[23] Backusova–Naurova forma
[24] Unicode Transformation Format
[25] Coordinated Universal Time

je pole (\x04). Pole je klasický dokument s celočíselnou hodnotou klíčů pro jednotlivé prvky pole. Klíče začínají hodnotou 0 a postupně jsou zvyšovány o 1. Samotné pole ["Hřbitovní", "kvítí"] je následně kódováno jako dokument {"0": "Hřbitovní", "1": "kvítí"}. Poslední datovou položkou je 32-bitový integer (\x01). Hodnota 1858 je převedena do hexadecimální podoby a je doplněna na 32 bitů. Dokument je ukončen bajtem \x00.

2.2.4 TEI

Formát Iniciativy pro kódování textů TEI[26] je jedním z nejstarších metadatových schémat. Jedná se o rozsáhlý formát, kterým lze popisovat všechny typy textových i netextových formátů. Využívá se hlavně v oblasti literatury. Pro popis digitalizovaných básní ho používá i ÚČL. Tento formát původně vycházel ze SGML, avšak nyní se již orientuje na standardy XML nebo W3C.

TEI se skládá z několika částí. Nejzákladnější částí je hlavička, která obsahuje všechny prvky a údaje sloužící bibliografickému popisu zdroje. Hlavička TEI se vztahuje vždy k jedné popisné jednotce a může být složena z několika jednotek nižší úrovně. Hlavičky jsou označeny a uzavřeny párovým tagem <teiHeader> a skládají se z částí. Tyto části obsahují informace o souboru, věcném profilu textu, kódování a historii změn.

Část obsahující popis souboru je označena tagem <fileDesc>. Najdeme v ní veškeré popisné údaje o daném objektu a můžeme jí využít k sestavení citace nebo pro vytvoření katalogizačního záznamu v knihovně. V popisu lze také nalézt informace o zdrojích, které byly pro daný objekt použity. Ze všech čtyř zmíněných částí je tato jediná povinná. Následující části jsou dobrovolné.

Další část hlaviček obsahuje informace o kódování a je označena tagem <encodingDesc>. Najdeme zde informace o tom, jakým způsobem byl zdroj modifikován a zda došlo ke změně kódování. Také zde můžeme najít veškeré informace o transkripcích a jiných změnách.

Profil textu najdeme v tagu <profileDesc> a obsahuje doplňkové nebibliografické popisné a kontextové informace o textu. Pod těmito informacemi si můžeme představit například informace o jazyku zdroje, situaci vzniku. Jsou zde také uvedeny věcné výrazy z různých věcných klasifikací a tezaurů.

Poslední část s tagem <revisionDesc> obsahuje podrobnou evidenci změn, ke kterým došlo v průběhu vzniku dokumentu nebo při jeho modifikaci. Tato část je zejména důležitá v případě, kdy je nutné odlišit a identifikovat různé verze jednoho zdroje. Díky této části je možné dohledat jakoukoliv dřívější verzi a zjistit, jaké změny byly v dané verzi provedeny. [28]

[26] Text Encoding Initiative

2.3 MongoDB

Jak již bylo zmíněno v předchozích sekcích, pro implementaci datového úložiště byla vybrána databáze MongoDB, která splňuje všechny požadavky pro uložení digitalizovaných sbírek básní. MongoDB se řadí mezi multiplatformní dokumentové NoSQL databáze a využívá dynamické databázové schéma, které umožňuje rychle a jednoduše vytvořit a integrovat data v rámci aplikací. Databázový systém MongoDB je vyvíjen společností MongoDB Inc. a do října roku 2018 byl zdarma pod licencí GNU AGPL[27] v3.0. Nyní běží pod veřejnou licencí na straně serveru (SSPL[28]). [29]

V následujících sekcích je popsáno, jak MongoDB funguje a jaké jsou základní vlastnosti systému z hlediska způsobu uložení dat, manipulace s daty, indexace, distribuce dat a také z hlediska zabezpečení.

2.3.1 Způsob uložení dat

Jelikož je MongoDB dokumentová databáze, ukládá záznamy dat jako dokumenty. Tyto dokumenty jsou ve formátu BSON, (viz 2.2.3), tedy v binární reprezentaci JSON dokumentů. Jednotlivé dokumenty se skládají z datových položek typu klíč-hodnota. Hodnoty datových položek mohou představovat kterýkoliv z datových typů BSON formátu, tedy i další dokumenty, pole nebo pole dokumentů. Názvy datových položek jsou datového formátu `string` a od verze MongoDB 3.6 nemohou začínat symboly '`.`' a '`$`'.

Název `"_id"` je rezervován pro primární klíč dokumentů, a tedy musí být unikátní v rámci kolekce databáze. Tento primární klíč nelze měnit a může být jakéhokoliv datového typu s výjimkou pole. Každý dokument uložený v kolekci musí mít unikátní `_id`. Tento identifikátor musí být v dokumentu vždy na první pozici. Pokud server přijme dokument, který má tuto položku na jiné pozice, pak ji automaticky přesune na začátek dokumentu. Pokud dokument při vkládání do kolekce neobsahuje tuto položku, pak je automaticky vygenerována a má typ `ObjectID`.

`ObjectID` je jeden z datových typů BSON formátu a představuje krátkou unikátní hodnotu, kterou lze velmi rychle vygenerovat. Jedná se o strukturu o délce 12 bajtů, která se skládá ze tří částí:

- časová známka (`timestamp`) o délce 4 bajtů reprezentující Unixový čas vytvoření daného `ObjectID`
- náhodná hodnota o délce 5 bajtů
- hodnota čítače o délce 3 bajtů (čítač je inicializován náhodně).

[27] GNU Affero General Public License
[28] Server Side Public License

Kromě povinného výskytu datové položky s názvem `"_id"` není struktura dokumentu nijak omezena. Dokumenty mohou být dokonce do sebe libovolně zanořené. Hloubka zanoření není nikterak omezena. V případě, že dokument obsahuje pole, pak toto pole může mít libovolný počet rozměrů. Omezena je však maximální velikost dokumentu. Jeden BSON dokument může mít velikost maximálně 16 megabajtů. Maximální velikost dokumentu pomáhá zajistit, aby jeden dokument nemohl využívat nadměrné množství paměti RAM a také aby nedošlo k problémům při přenosu dokumentů. Pokud je potřeba uložit větší dokumenty, pak lze využít rozhraní GridFS API. [2]

2.3.1.1 Databáze a kolekce

V MongoDB lze vytvářet libovolné množství databází. V tradičních relačních databázích jsou data ukládána do tabulek. V MongoDB jsou k tabulkám analogické tzv. kolekce. Ty slouží k ukládání jednotlivých dokumentů. V jedné databázi můžeme mít prakticky neomezený počet kolekcí s neomezeným počtem dokumentů.

Pro používání databází a manipulaci s daty lze použít `mongo` konzoli. V ní pak pomocí příkazu `use myDB` zvolíme, s jakou databází budeme pracovat. Pomocí stejného příkazu je možné i vytvoření databáze. V seznamu databází se ale objeví až při vložení prvních dat. Pro zobrazení všech dostupných databází lze využít příkaz `show databases`.

Pro vytvoření kolekce je také potřeba nahrát první data. Pokud ale chceme vytvořit kolekci explicitně, můžeme použít příkaz `db.createCollection()`. V této metodě můžeme nastavit různé vlastnosti kolekce, kterými může být například maximální velikost kolekce nebo validační pravidla pro jednotlivé dokumenty. Pokud nepotřebujeme nastavit žádné speciální vlastnosti kolekce, pak je naprosto zbytečné vytvářet kolekci tímto způsobem. Pokud bychom chtěli vlastnosti kolekce změnit později, lze využít příkaz `collMod`. [2]

2.3.1.2 Validace vůči schématu

Databáze MongoDB umožňuje vkládání dokumentů s různou strukturou do jedné kolekce databáze. Jsou však situace, kdy bychom chtěli zajistit, aby do jedné kolekce bylo možné nahrát pouze dokumenty s určitým schématem. Od verze MongoDB 3.2 disponuje nástroj možností validace dokumentů vůči schématu. Pro specifikaci validačních pravidel pro novou kolekci lze použít již zmíněný příkaz `db.createCollection()` s možností `validator`. Případně lze použít příkaz `collMod`, pokud potřebujeme přidat validační pravidla pro již existující kolekci.

Pro definici validačních pravidel lze použít operátory jako `$regex`, `$type` nebo `$in`. Daleko lepším způsobem vytvoření validačních pravidel je využití JSON Schema (viz 2.2.2.1). Ve výrazu `validator` lze použít mongo operátor `"$jsonSchema"`, ve kterém stačí pouze specifikovat dané JSON Schema.

Schéma lze vytvořit podle pravidel v JSON Schema dokumentace, ale lze použít i pár odlišností. Klasické JSON Schema nabízí pouze šest základních datových typů, které je třeba specifikovat pomocí datové položky s názvem `type`. MongoDB ale umožňuje využít daleko více datových typů, jelikož dokumenty neukládá v JSON formátu, ale ve formátu BSON. Proto je možné místo názvu `type` použít `bsonType`, jehož hodnotou může být kterýkoliv datový typ podporovaný BSON dokumenty.

Ve výrazu `validator` lze uvést ještě dvě další možnosti chování systému. První z nich je `validationAction`, která určuje, co se bude dít v případě, že dokument nebude odpovídat danému schématu. Pokud je hodnota položky `validationAction` nastavena na `warn` a dokument není validní vůči definovanému schématu, pak MongoDB sice povolí vložení či modifikaci daného dokumentu, zároveň však tuto skutečnost zaloguje. Pokud je hodnota položky `validationAction` nastavena na `error`, pak je nahrání či modifikace dokumentu zamítnuto v případě porušení validačních pravidel. Pokud tato položka není ve výrazu `validator` uvedena, pak je jako výchozí hodnota použita `error`.

Druhou možností je `validationLevel`, která určuje, jak striktně budou validační pravidla aplikována. Zde můžeme použít dvě hodnoty, konkrétně se jedná o `strict` a `moderate`. V případě použití hodnoty `strict` jsou validační pravidla aplikována na všechna vkládání a aktualizování dokumentů. Pokud se ale rozhodneme použít hodnotu `moderate`, pak jsou validační pravidla aplikována pouze na všechna vkládání nových dokumentů a na aktualizaci pouze těch dokumentů, které byly před aktualizací validní vůči danému schématu. Pokud aktualizovaný dokument před změnou nebyl validní, pak není validita kontrolována. [2]

2.3.2 Dotazy a manipulace s daty

V této sekci je vysvětlena základní manipulace s daty. Nejprve je potřeba zmínit základní syntaxi práce s poli a vnořenými dokumenty. V obou případech používáme notaci s tečkou (tzv. *Dot Notation*). Pokud dokument obsahuje pole, pak k jeho prvkům lze přistupovat přes index pomocí zápisu `"<array>.<index>"`. První prvek v poli je na pozici odpovídající indexu 0. U zanořených dokumentů je to podobné. Pro přistoupení k datové položce vnořeného dokumentu použijeme zápis `"<embedded document>.<field>"`.

Nyní si pojďme ukázat jednotlivé CRUD[29] operace. Vložení dokumentu do kolekce provedeme pomocí metody `db.collection.insertOne()`. Jako parametr této metody lze použít daný dokument ve formátu JSON. Pokud je potřeba vložit více dokumentů do jedné kolekce najednou, pak lze použít metodu `db.collection.insertMany()`. Pomocí těchto metod je možné nahrát dokumenty pouze do jedné kolekce. Všechny operace pro zápis jsou

[29] Operace Create, Read, Update, Delete

atomické na úrovni dokumentu. Jednoduché nahrání dokumentu do databáze může vypadat například takto:

```
db.books.insertOne(
    {
        autor: "Jan Neruda",
        titul: "Hřbitovní kvítí",
        rok: 1858
    }
)
```

Pomocí operací pro čtení lze získat dokumenty z kolekcí. Základní metodou pro získání dokumentů je metoda `db.collection.find()`. V této metodě lze uvést kritéria selekce (první parametr) a projekce (druhý parametr). Následující dotaz vrátí seznam prvních 10 autorů sbírek básní, které byly napsány do roku 1900 včetně:

```
db.books.find(
    { rok: { $gte: 1900 } },
    { autor: 1 }
).limit(10)
```

Pro aktualizaci dokumentů v kolekci jsou v MongoDB k dispozici tři základní metody `db.collection.updateOne()`, `db.collection.updateMany()` a `db.collection.replaceOne()`. Syntaxe příkazů je stejná jako v případě předchozích operací pro čtení. Uvedený dotaz aktualizuje položku z názvem `"autor"` u všech sbírek, které byly napsány před rokem 1600:

```
db.books.updateMany(
    { rok: { $lt: 1600 } },
    { $set: { autor: "Neznámý" } }
)
```

Zbývá doplnit pouze operace pro mazání dokumentů. K tomu slouží metody `db.collection.deleteOne()` a `db.collection.deleteMany()`. V níže uvedené ukázce je dotaz, který z databáze odstraní všechny knihy napsané před rokem 1600:

```
db.books.deleteMany(
    { rok: { $lt: 1600 } }
)
```

2.3.3 Agregované dotazy

Kromě standardních CRUD operací umožňuje MongoDB vyhodnocovat i složitější agregované dotazy. Pomocí nich lze seskupovat hodnoty z více dokumentů

a nad těmito daty následně provádět výpočty. MongoDB nabízí tři základní možnosti, jak provádět agregace nad dokumenty.

První z možností se nazývá *Aggregation Pipeline* a je postavena na konceptu *pipeline*. V tomto konceptu jsou dokumenty postupně zpracovávány pomocí jednotlivých operací a tím jsou transformovány na agregovaný výsledek. Jednotlivé operace v transformaci mohou představovat jednoduché filtry nebo transformace, které upravují strukturu výstupního dokumentu, ale také se může jednat o operace seskupování a třídění dokumentů podle konkrétních datových položek v dokumentech. Také nelze opomenout operace provádějící výpočty nad seskupenými dokumenty, například výpočet průměru a podobné. Pro Aggregation Pipeline lze použít metodu `db.collection.aggregate()` a jako parametr uvést list operací, které je třeba nad dokumenty v kolekci provést. Jednoduchý příklad použití je uveden níže. V první fázi jsou filtrovány dokumenty podle datové položky žánr a v následující fázi už jsou zpracovány pouze dokumenty žánru `poezie`. Ve druhé fázi jsou dokumenty seskupeny podle jména autora a je vypočítán rok, kdy autor napsal první knihu a také rok, kdy napsal poslední:

```
db.books.aggregate([
    { $match: { zanr: "poezie" } },
    { $group: {
        _id: "$autor",
        prvni_kniha: { $min: "$rok" },
        posledni_kniha: { $max: "$rok" }
    } }
])
```

Druhá možnost, kterou MongoDB nabízí pro provádění agregací, jsou tzv. jednoúčelové agregační operace. Mezi tyto operace řadíme například operaci `distinct`, která vrací seznam unikátních hodnot dané datové položky. Jiným příkladem může být operace `count`, která vrací počet dokumentů.

Poslední možností je využití principu MapReduce. Základní myšlenka tohoto principu je založena na funkcích `Map` a `Reduce`. Funkce `Map` slouží ke zpracování všech objektů z množiny vstupních dat. Výstupem jsou pak dvojice (klíč, hodnota) pro každý ze zpracovávaných objektů. Druhá funkce `Reduce` slouží ke sloučení hodnot pro jednotlivé klíče z předchozí fáze. Pomocí tohoto sloučení je sestaven výsledek. [8]

Pro MapReduce operace poskytuje MongoDB dva agregační operátory `$accumulator` a `$function`. Pomocí těchto operátorů lze definovat své vlastní agregační výrazy.

Dokumentace MongoDB uvádí, že Aggregation Pipeline nabízí daleko výkonnější a použitelnější řešení než MapReduce operace. Dále je zde uvedeno, že všechny MapReduce operace lze přepsat pomocí operátorů Aggregation Pipeline. [2]

2.3.4 Indexy

Při vyhodnocování dotazů musí MongoDB projít všechny dokumenty v kolekci sekvenčně a vybrat ty, které splňují všechna kritéria uvedená v dotazu. To ale mnohdy nemusí být úplně efektivní, proto nabízí MongoDB indexy, které podporují efektivnější provádění dotazů. Pokud existují indexy pro daný dotaz, pak systém může výrazně snížit počet dokumentů, které prochází. Indexy jsou speciální datové struktury ukládající pouze malou část dat, které lze velmi snadno a rychle procházet. Indexy ukládají hodnoty konkrétních datových položek nebo seřazenou množinu těchto hodnot. Seřazení datových položek indexu podporuje efektivní vyhledávání na základě rovnosti či rozsahu. Kromě toho využívá MongoDB indexy i v případě, kdy je v rámci dotazu požadován seřazený výsledek.

MongoDB vytváří unikátní index na datové položce `_id` při vkládání nového dokumentu do kolekce. Díky tomu není možné vložit dva dokumenty se stejnou hodnotou `_id` a lze tuto hodnotu použít jako primární klíč. Vytváření indexů na této datové položce nelze žádným způsobem zrušit.

Pro vytvoření indexů nad danými klíči v kolekci lze v MongoDB použít metodu `db.collection.createIndex()`, která má dva parametry. První parametr by měl obsahovat klíče, nad kterými chceme vytvářet indexy a také typy jednotlivých indexů. V druhém parametru je možné uvést další možnosti indexů. V těchto možnostech například můžeme specifikovat jméno indexu. Vytvořené indexy používají strukturu B-stromů, které byly popsány v sekci týkající se dokumentových databází (viz 2.1.2.4). Vytvoření indexu je možné pouze v případě, že ještě neexistuje index se stejnou specifikací. Jaké indexy již kolekce obsahuje, lze zjistit pomocí metody `db.collection.getIndexes()`.

MongoDB poskytuje hned několik typů indexů, které podporují specifické typy dotazů. Jedná se o tyto typy:

- **Single Field Index** - index pro pouze jeden klíč lze využít například pro operace řazení
- **Compound Index** - složený index pro více klíčů lze použít pro operace řazení podle více klíčů
- **Multikey Index** - index pro hodnoty v poli
- **Geospatial Index** - indexy pro geografická data se dělí na 2D indexy pro body v rovině a 2D Sphere indexy pro sférickou geometrii
- **Text Index** - index pro snadné vyhledávání v textu
- **Hashed Index** - index pro hash hodnoty.

U každého z typů indexu lze ještě definovat jeho vlastnosti. Konkrétně se jedná o pět základních vlastností:

- **Unique Indexes** - hodnoty indexovaných klíčů musí být unikátní
- **Partial Indexes** - indexují se pouze dokumenty odpovídající danému výrazu
- **Sparse Indexes** - index obsahuje pouze záznamy dokumentů, nad kterými je vytvořen index
- **TTL**[30] **Indexes** - indexy pro automatické mazání dokumentů z kolekce po uplnynutí dané doby
- **Hidden Indexes** - indexy nejsou viditelné a nelze je využít pro dotazování.

Po nějaké době se nám může stát, že chceme indexy z kolekce odebrat. K tomu slouží metoda `db.collection.dropIndex()`. [2]

2.3.5 Transakce

V databázovém systému MongoDB je každá operace prováděná nad jedním dokumentem atomická. Ve většině případů jsou k zachycení vztahů mezi daty využívány vnořené dokumenty místo normalizace a uložení částí dokumentů do více kolekcí. Tato vlastnost operací vylučuje potřeby transakcí nad více dokumenty. Pro situace, které vyžadují atomicitu i u operací pro zápis a čtení nad více dokumenty, nabízí MongoDB tzv. *multi-document* transakce. Tyto transakce lze využít při práci s více operacemi, kolekcemi, databázemi, dokumenty a shardy.

2.3.6 Replikace a Sharding

V sekci týkající se NoSQL databází byly vysvětleny základní pojmy ohledně distribuce dat (viz 2.1.1.2). Pojďme se nyní podívat, jak řeší distribuci dat databázový systém MongoDB.

Replica set je skupina MongoDB serverů, na kterých jsou spravována stejná data. Tato skupina obsahuje uzly s daty a volitelně jeden speciální uzel označovaný jako *arbiter node*. Dále je z uzlů zvolen primární uzel, ostatní jsou sekundárními uzly.

Primární uzel zpracovává všechny operace pro zápis. Replica set může obsahovat pouze jeden uzel, který zpracovává a potvrzuje operace pro zápis. Ve výjimečných případech může být přechodně primárním uzlem i jiná z MongoDB instancí serveru. Všechny provedené změny nad daty jsou vždy logovány. Sekundární uzly si udržují data tak, aby co nejlépe odrážely data uložená na primárním uzlu. Pokud dojde k výpadku primárního uzlu, pak jeden ze sekundárních uzlů převezme funkci primárního. O tom, který z nich

[30] Time to live

to bude, je rozhodnuto volbami, kterých se účastní všechny dostupné uzly. Uzel typu arbiter se pouze účastní voleb, neobsahuje replikaci dat a zůstává po celou dobu uzlem typu arbiter. Primární uzel se však může stát uzlem sekundárním a naopak.

Sekundární uzly aplikují jednotlivé operace na replikovaná data asynchronně a díky tomu může Replica set neustále fungovat i v případě výpadku jednoho či více uzlů. Tyto uzly si neustále udržují kopii dat primárního uzlu. Při kopírování dat však může dojít ke zpoždění. Malé doby zpoždění jsou přijatelné, avšak při narůstající době zpoždění se mohou vyskytnout značné problémy.

Výpadek primárního serveru se také označuje jako *Automatic Failover*. Pokud primární uzel přestane komunikovat po dobu konfigurovanou v položce `electionTimeoutMillis` (výchozí hodnota je 10 sekund), pak jeden ze sekundárních uzlů zahájí volbu nového primárního uzlu. Dokud není volba nového primárního uzlu dokončena, není možné zpracovávat operace zápisu. Sekundární uzly mohou dále zpracovávat operace pro čtení i přesto, že volba ještě nebyla dokončena. [2]

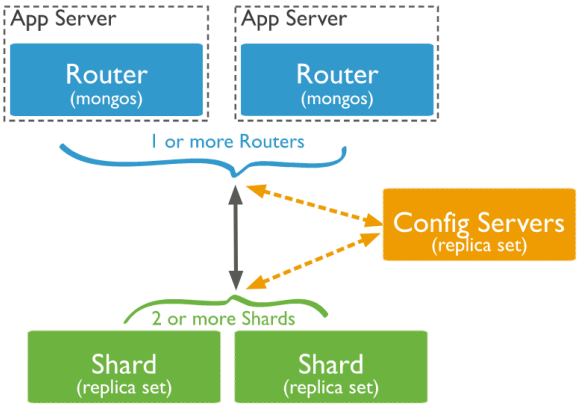
Replikace je zajisté velmi důležitá metoda distribuce dat, avšak jejím problémem je, že všechna data držíme stále na jednom místě v několika kopiích. Tento způsob neřeší vůbec škálovatelnost. Problém nastává v případě, že jsou data v jedné databázi příliš velká. Pak vysoká četnost dotazů může vyčerpat kapacitu CPU[31]. Také bychom se měli vyvarovat uložení dat, jejichž velikost značně přesahuje velikost RAM[32] paměti. V takovém případě také mohou nastat problémy v souvislosti s přístupem k datům. Sharding umožní rozmístit jednu databázi nebo kolekci mezi několik uzlů. MongoDB samozřejmě nabízí horizontální škálování pomocí shargingu. Tzv. *shared cluster* se skládá ze tří následujících komponent:

- **Shard** - obsahuje dílčí část dat a lze ho nasadit jako replica set
- **Mongos** - funguje jako směrovač dotazů a poskytuje rozhraní mezi klientskými aplikacemi a shard clusterem
- **Config server** - ukládají metadata a nastavení konfigurace clusteru.

MongoDB sharduje data na úrovni kolekce a distribuuje je mezi jednotlivými shardy v clusteru. Následující obrázek znázorňuje interakci komponent v rámci shard clusteru. Jsou zde zobrazeny jednotlivé instance `mongos` serverů, se kterými komunikují klientské aplikace a také shard clustery, které jsou nasazeny jako replica set. Nechybí zde ani config server, který si uchovává veškeré informace o konfiguraci clusteru. [2]

[31] Centrální procesorová jednotka
[32] Random Access Memory

Obrázek 2.5: Interakce komponent v shard clusteru [2]



2.3.7 Zabezpečení

MongoDB poskytuje různé mechanismy zabezpečení jako například autentizaci, řízení přístupu, šifrování databáze a další. Pomocí autentizace lze ověřit totožnost jednotlivých klientů. Pokud je MongoDB v režimu autorizace, pak je vyžadováno, aby se každý klient nejprve autentizoval. Do té doby nemá přístup k žádným datům v databázi. Ačkoliv jsou pojmy autentizace a autorizace úzce spojeny, význam mají zcela odlišný. Autentizace slouží k ověření totožnosti uživatele, naopak autorizace určuje, ke kterým operacím a zdrojům má autentizovaný klient přístup. Pro autentizaci musí uživatel zadat uživatelské jméno a heslo. To je možné pomocí příkazu `db.auth()`. MongoDB podporuje hned několik mechanismů autentizace. Jako výchozí je použit SCRAM[33], ale lze použít například i standard x.509.

MongoDB využívá řízení přístupu na základě rolí (RBAC[34]). Každému uživateli je určena jedna nebo více rolí, které určují, k jakým zdrojům a operacím bude mít přístup. Pokud uživateli nejsou přiřazeny žádné role, pak nemá přístup do systému. Ve výchozím režimu MongoDB není autorizace aktivní. Pro zapnutí režimu autorizace je třeba při spuštění MongoDB serveru použít příznak `--auth`.

Poslední věc, která stojí za zmínku, je, že MongoDB podporuje TLS/SSL[35] k šifrování veškerého síťového provozu. To zajišťuje, že data lze číst pouze v systému MongoDB nebo z přidruženého klienta. Od verze MongoDB 4.0 byla ukončena podpora pro šifrování TLS 1.0 na systémech, které mají k dispozici novější verzi TLS. [2]

[33] Salted Challenge Response Authentication Mechanism
[34] Role-Based Access Control
[35] Transport Layer Security/Secure Sockets Layer

2.4 Mongoeye

V předchozí sekci byla popsána validace dokumentů vůči schématu v databázovém systému MongoDB (viz 2.3.1.2). V praxi ale mnohdy nemáme definované schéma dokumentů a chtěli bychom ho vytvořit na základě dat, která máme uložena v databázi MongoDB. Typickým příkladem jsou digitalizované sbírky ÚČL. Ty vlastně představují dokumenty, které lze po konverzi do JSON formátu velmi snadno nahrát do jedné kolekce. Pro budoucí práci se sbírkami by se ale hodilo mít společné schéma sbírek.

Pro tyto účely byl vytvořen v roce 2017 analytický nástroj Mongoeye, který je vhodný pro analýzu schémat dokumentů uložených v databázi MongoDB a jejich dat. Tento nástroj poskytuje rychlý přehled o datech v kolekci databáze. V dnešní době se jedná o jeden z nejrychlejších analyzátorů schémat pro MongoDB. K dispozici je lokální analýza pomocí paralelního algoritmu, ale také vzdálená analýza pomocí agregačního rámce v MongoDB. Výhodou tohoto nástroje je i možnost uložení výstupu analýzy pomocí formátu JSON nebo YAML[36]. Mongoeye je pod licencí GPL-3.0 a k dispozici je také jako Javascript knihovna.

Nástroj Mongoeye je napsán v jazyce Go. Jedná se binární soubor, který je spustitelný z příkazové řádky. Pro spuštění je vyžadována instalace Go 1.8. Syntaxe příkazu je následovná:

```
mongoeye [host] database collection [flags],
```

kde volitelná položka `host` je adresa, na které běží MongoDB, `database` resp. `collection` představuje konkrétní databázi resp. kolekci a `flags` jsou další volitelné možnosti. Výchozím výstupem tohoto příkazu je tabulka, která má tři sloupce. První sloupec obsahuje jméno klíče a jeho datové typy, v dalších je pak počet výskytů klíče a jeho datových typů ve všech dokumentech v kolekci a také tato hodnota vyjádřená v procentech. V případě výstupu ve formátu JSON jsou k dispozici datové položky s následujícími názvy:

- **database** - název databáze
- **collection** - název kolekce
- **plan** - typ analýzy (lokální nebo vzdálená)
- **duration** - doba trvání analýzy
- **allDocs** - počet dokumentů v kolekci
- **analyzedDocs** - počet analyzovaných dokumentů
- **fieldsCount** - počet nalezených klíčů v dokumentech

[36] YAML Ain't Markup Language

- **fields** - analýza jednotlivých klíčů (obsahuje název položky, hloubku, počet výskytů, datové typy a počet výskytů jednotlivých datových typů).

V příkazu je možné uvést další volitelné možnosti analýzy. Při použití příznaku `--value` bude výstup obsahovat také minimální, maximální a průměrnou hodnotu datových položek numerického datového typu s daným klíčem. Pro datové typy `string`, `array` a `object` lze použít příznak `--length`, díky kterému bude výstup obsahovat minimální, maximální a průměrnou hodnotu délky těchto položek. Dále lze použít příznaky `--count-unique` pro výpis počtu unikátních hodnot pro daný klíč a `--most-freq N` nebo `--least-freq N` pro výpis nejčastějších či nejméně častých výskytů hodnot pro daný klíč. Jednou z dalších volitelných možností výstupu je výpis hodnot pro histogram, k čemuž lze použít přepínač `--value-hist` nebo jiné přepínače pro speciální histogramy.

Kromě možností pro výstup jsou také k dispozici možnosti pro rozsah analyzovaného obsahu. Například pomocí příznaku `--match` lze definovat regulární výraz, kterému musí odpovídat analyzovaná data. Dále je k dispozici příznak `--sample`, kterým lze určit, jak bude vypadat vzorek dat pro analýzu. Jako hodnotu tohoto příznaku lze použít `all` pro všechny dokumenty v kolekci, `first:N` pro N prvních dokumentů v kolekci, `last:N` pro N posledních dokumentů v kolekci a `random:N` pro N náhodně vybraných dokumentů, kde $N > 1$. Tyto příznaky slouží pro selekci dokumentů, pro projekci je možné použít příznak `--project`. Velmi důležitý příznak je `--depth`, kterým lze nastavit, do jaké hloubky se bude provádět analýza u vnořených dokumentů.

Možností pro výstup nebo vzorek dat je samozřejmě daleko více, v této sekci však byly uvedeny jen nejdůležitější z nich. Nástroj také nabízí další možnosti pro připojení k databázi, autentizaci nebo jiné možnosti pro nastavení barev výstupu. [30]

Kapitola 3

Analýza a návrh řešení

Tato kapitola pojednává o návrhu datového úložiště a aplikačního serveru, který bude vyhovovat požadavkům ÚČL, které již byly zmíněny v první kapitole (viz 1.2). Jedním z požadavků je také vytvoření společného schématu pro všechny digitalizované sbírky. První sekce této kapitoly se tedy zabývá analýzou dat.

V dalších sekcích je popsán návrh architektury, datového úložiště a také aplikačního serveru. Poslední sekce je věnována technologiím, pomocí kterých bude řešení realizováno.

3.1 Analýza dat

Jak již bylo několikrát zmíněno, ÚČL má k dispozici 495 naskenovaných děl a 1700 digitalizovaných děl. K naskenovaným dílům jsou k dispozici tři typy souborů. Prvním typem souborů jsou obrázky ve formátu JPG obsahující naskenovanou stránku sbírky. V rámci jedné knihy mají obrázky stejné rozměry, dvě různé knihy však mohou mít odlišnou šířku. Výška obrázků je vždy 800 pixelů. Druhým typem souborů jsou miniatury těchto obrázků. Ty jsou přesně 8x menší než originály (jejich výška je tedy vždy 100 pixelů). Dále jsou k dispozici soubory ve formátu CSV obsahující mapování názvů souborů na čísla stránek.

ÚČL požaduje, aby těmto souborům nebyla věnována přílišná pozornost. Daleko větší důraz klade na digitalizované sbírky ve formátu XML, jejichž vytvoření trvalo několik let. Běžní čtenáři by měli především pracovat s těmito sbírkami. K těmto souborům nebyly poskytnuty ze strany ÚČL žádné informace. Struktury jednotlivých souborů se mohou navzájem lišit, jelikož na digitalizaci sbírek pracovalo velmi mnoho lidí.

Právě z těchto důvodů je potřeba analyzovat tato díla, zjistit, z jakých elementů se skládají, a pokusit se nalézt společné schéma. Jelikož je formát XML velmi snadno převeditelný na formát JSON, byly všechny soubory nej-

prve převedeny do formátu JSON a následně byly nahrány do kolekce jedné databáze MongoDB. Provedená transformace je přiblížena v následující kapitole (viz 4.2.1). Pro analýzu struktury dokumentů byl použit již zmíněný nástroj Mongoeye (viz 2.4).

3.1.1 Obsah sbírek

U každého dokumentu v kolekci bylo zjištěno, z jakých různých datových položek se skládá. Datové položky v JSON formátu lze přirovnat k XML elementům v původních dokumentech. Každý dokument má stromovou strukturu, kde vnitřní uzly představují objekty či pole objektů, listy tvoří datové položky jednoduchých datových typů jako například `integer` nebo `string`. Kořen stromu tvoří vždy datová položka s názvem `dilo`. Maximální hloubka stromu je 13.

V následujících tabulkách jsou uvedeny všechny datové položky sbírek s různým názvem. U každé z nich jsou základní informace jako název, význam a možné datové typy. U objektů a polí objektů je také uvedeno, ze kterých další datových položek se mohou skládat. Vnořené objekty jsou analogické k původním vnořeným XML elementům.

Tabulka 3.1: Datové položky ve sbírkách (1. část)

Id	Název	Význam	Datové typy	Obsah
1	dilo	dílo	objekt	2, 21
2	hlavicka	hlavičkové údaje	objekt	3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
3	autor	autor díla/koment.	objekt	52, 53, 58
4	autormota	autor mota v knize	objekt	52, 53, 58
5	edicnipoznamka	ediční poznámky	objekt	52, 53, 54, 55, 58
6	format	formát knihy	objekt	52, 58
7	komentare	komentáře k dílu	objekt	8
8	komentar	komentář	objekt, pole	3, 9, 23, 39, 44, 53, 54, 56, 57, 58, 59
9	datum	datum	objekt	58
10	misto	místo vydání knihy	objekt	52, 58
11	moto	moto v knize	objekt	52, 53, 58

Tabulka 3.2: Datové položky ve sbírkách (2. část)

Id	Název	Význam	Datové typy	Obsah
12	podtitul	podtitul díla	objekt, pole	8, 52, 53, 54, 58
13	popis	popis díla	objekt	52, 53, 58
14	rok	rok vydání knihy	objekt	52, 53, 58
15	stran	počet stran	objekt	53, 58
16	titul	titul díla	objekt	52, 53, 58
17	venovani	věnování	objekt, pole	8, 23, 44, 52, 53, 54, 56, 58
18	vydani	vydání knihy	objekt	52, 53, 58
19	vydavatel	vydavatel knihy	objekt	52, 53, 58
20	zdroj-signatura	zdroj-signatura	objekt	52, 53, 58
21	text	obsah díla	objekt	17, 22, 24, 27, 28, 31, 35, 36, 43, 45, 46, 50, 51, 52
22	doprovod	doprovod	objekt, pole	8, 23, 39, 44, 53, 54, 56, 58
23	verze	verze	objekt, pole	8, 39, 53, 54, 56, 57, 58
24	kniha	kniha	objekt	17, 22, 25, 26, 28, 29, 35, 36, 42, 45, 46, 47, 50, 51, 52, 57
25	basen	básně	objekt, pole	12, 17, 22, 26, 27, 28, 29, 30, 31, 34, 35, 36, 37, 40, 41, 42, 43, 45, 46, 49, 50, 52, 57
26	fales-titulbasen	faleš. - titul básně	objekt, pole	8, 23, 56
27	moto-autor	autor mota	objekt, pole	8, 23, 53, 54, 58

Tabulka 3.3: Datové položky ve sbírkách (3. část)

Id	Název	Význam	Datové typy	Obsah
28	moto-text	text mota	objekt, pole	8, 23, 53, 54, 58
29	nadpis	nadpis	objekt	8, 23, 52, 53, 56, 57, 58
30	nadpisvbasni	nadpis v básni	objekt, pole	8, 23, 56
31	obsah	obsah	objekt, pole	32, 33
32	vlevo	obsah vlevo	objekt	8, 23, 53, 54, 56, 58
33	vpravo	obsah vpravo	objekt	8, 23, 53, 54, 55, 56, 58
34	oddilbasne	oddíl básně	objekt, pole	12, 17, 22, 26, 27, 28, 29, 30, 31, 35, 36, 37, 40, 41, 42, 43, 45, 46, 49, 50, 52, 57
35	spisovatel	spisovatel	objekt, pole	8, 23, 56, 58
36	strana	strana	objekt, pole	8, 23, 56, 57, 58
37	strofa	sloka	objekt, pole	17, 22, 26, 27, 30, 35, 36, 38, 40, 42, 45, 46, 50, 57
38	v	verš	objekt, pole	8, 23, 39, 44, 52, 53, 54, 56, 57, 58
39	prolozeno	proloženo	objekt, pole	8, 23, 53, 54, 56, 58
40	zakladnitext	základní text	objekt, pole	8, 23, 39, 44, 53, 54, 56, 58
41	podtitulbasne	podtitul básně	objekt, pole	8, 23, 56

Tabulka 3.4: Datové položky ve sbírkách (4. část)

Id	Název	Význam	Datové typy	Obsah
42	poznpodcarou	poznámka pod čarou	objekt, pole	8, 23, 39, 44, 53, 54, 56, 58
43	sifra	speciální zápis	objekt	54
44	sup	horní index	objekt, pole	56, 58
45	tiraz	tiráž	objekt, pole	8, 23, 39, 44, 53, 54, 56, 58
46	titulsbirka	titul sbírky	objekt, pole	8, 52, 56, 58
47	oddil	oddíl	objekt, pole	12, 17, 22, 25, 26, 27, 28, 29, 30, 31, 36, 40, 42, 43, 45, 46, 48, 49, 52, 57
48	pododdil	pododdíl	objekt, pole	12, 17, 22, 25, 27, 28, 29, 36, 52, 57
49	fales-tituloddil	faleš. - titul oddílu	objekt, pole	56
50	podtitulsbirky	podtitul sbírky	objekt, pole	8, 23, 54, 56
51	sbirka	sbírka básní	objekt, pole	12, 17, 22, 25, 26, 27, 28, 29, 30, 31, 35, 36, 40, 42, 43, 45, 46, 47, 49, 50, 52, 57

Výše uvedené datové položky se týkají především textového obsahu sbírek. Každá sbírka je rozdělena na dvě hlavní části představující hlavičkové údaje knihy (název knih, autor knihy, rok vydání atd.) a samotný textový obsah knihy (oddíly, básně, oddíly básní, verše atd.). V tabulkách je u každé datové položky uvedeno, jaké další položky může obsahovat. Ne každá sbírka obsahuje všechny tyto položky, tabulky značí pouze možný výskyt.

Uvedené významy jednotlivých názvů položek jsou platné téměř u všech

datových položek. Jedinou výjimku tvoří položka `autor`, která může představovat autora díla, ale také ji může obsahovat položka `komentar`. V takovém případě se nejedná o autora díla, ale o autora komentáře. Jelikož se s touto datovou položkou v rámci práce nepracuje, nebyla nijak přejmenována.

Kromě tohoto typu položek obsahují sbírky datové položky týkající se stylů a formátování textu. Ty jsou uvedeny v následující tabulce.

Tabulka 3.5: Datové položky stylu ve sbírkách

Id	Název	Význam	Datové typy	Obsah
52	br	nové řádky	objekt, pole	58
53	nbsp	nezlomitelné mezery	objekt, pole	58
54	i	kurzíva	objekt, pole	8, 23, 52, 53, 55, 56, 58
55	tab	tabulátory	objekt, pole	58
56	b	tučný text	objekt, pole	8, 23, 39, 44, 52, 53, 54, 58

V následující tabulce jsou datové položky, které byly v JSON formátu vytvořeny i přesto, že v původních sbírkách nebyl XML element se stejným názvem. Jedná se například o položky `_attributes` představující atributy u XML elementů a `_text` pro hodnoty jednoduchých elementů, které neobsahují další vnořené elementy nebo pole. Obdobně jako u položky `autor` má položka `text` také dvojí význam. Jelikož se s touto položkou pracuje, byla přejmenována na `text_`. Poslední nově vytvořenou položkou je `_id`, kterou vytváří databázový systém MongoDB automaticky jako unikátní identifikátor dokumentů.

Tabulka 3.6: Dodatečně vytvořené datové položky ve sbírkách

Id	Název	Význam	Datové typy	Obsah
57	_attributes	vlastnosti	objekt	61, 62, 63, 64, 65, 66, 67
58	_text	textový obsah	pole, řetezec, číslo	
59	text_	textový obsah	objekt	58
60	_id	unikátní identifikátor	ObjectId	

V poslední tabulce jsou uvedeny atributy, které se nacházejí v XML elementech v původních datech.

Tabulka 3.7: Atributy ve sbírkách

Id	Název	Význam	Datové typy
61	id	identifikátor	řetězec
62	typ	typ	řetězec
63	incipit	začátek	řetězec
64	prazdny	prázdný	řetězec
65	netistena	strana netištěna	řetězec
66	odsazeny	odsazeno	řetězec
67	naskenovana	strana naskenována	řetězec

Více datových položek se v dokumentech zatím nenachází. Není však vyloučeno, že během následujících let nebude nutné použít položku, která doposud v digitalizovaných sbírkách uvedena není.

3.1.2 Společné schéma

Analýza v předchozí sekci detailně popisuje strukturu sbírek. Je z ní patrné, které datové položky se mohou v digitalizovaném díle vyskytnout, jaký může být obsah těchto položek a jaké datové typy mohou položky představovat. Analýza však odkryla i celou řadu problémů. Jedním z problémů je nejednoznačnost datových typů, kdy v některých případech je datová položka například objekt a v jiných pole. Jiným problémem je například to, že analýza nestačí ke striktnímu omezení obsahu jednotlivých datových položek. Například u datové položky `vydani` analýza neobjevila žádnou sbírku, ve které by tato položka obsahovala položku `komentar`. V praxi je logicky možné, že k vydání knihy budou chtít zaměstnanci ÚČL napsat nějaký komentář. Takových příkladů lze uvést opravdu mnoho.

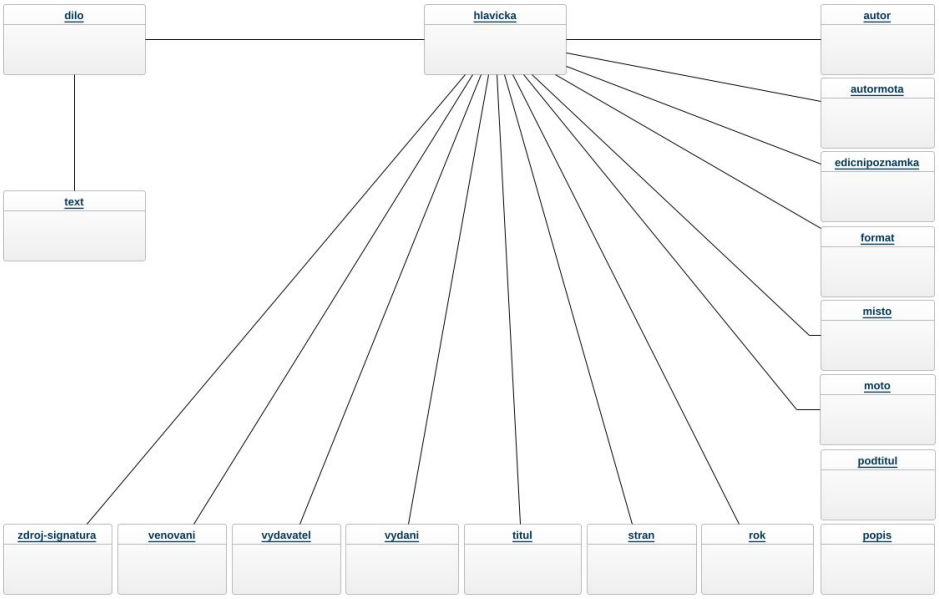
A právě proto vznikají dva pohledy na společné schéma. V prvním případě schéma popisuje strukturu, kterou lze najít ve všech sbírkách. V druhém případě společné schéma popisuje možnou strukturu na základě všech sbírek.

Snahou ÚČL je sjednotit schéma dokumentů a určit tak základ všech sbírek. K tomu je vytvořeno validační schéma. To popisuje ty položky, které musí nutně obsahovat všechny dokumenty. Pomocí nástroje Mongoeye lze tyto datové položky snadno určit. Mongoeye totiž u každé položky a jejích datových typů v dokumentu určí pravděpodobnost výskytu. Jedná se tedy o ty položky, u kterých je šance výskytu 100%. Schéma nevyloučí možnost výskytu nových datových položek. Bude možné ho využít i v databázi MongoDB pro validaci dokumentů vůči schématu při vkládání a úpravě sbírek. Při vytváření nových sbírek bude v budoucnu nutné respektovat toto schéma.

Na následujícím obrázku je znázorněna společná struktura všech sbírek básní. Každá sbírka tedy obsahuje položku `dilo`, která se pak skládá z částí `text` a `hlavicka`. Skladbu položek v části textu sbírky nelze nijak omezit. Naopak hlavičkové údaje musí vždy obsahovat minimálně 15 základních da-

tových položek. Schéma bylo převedeno do JSON Schema, kde každá datová položka je datového typu `object`. Toto JSON Schema lze nalézt na přiloženém DVD disku ve složce (`/JSONSchemas`).

Obrázek 3.1: Validační model [3]



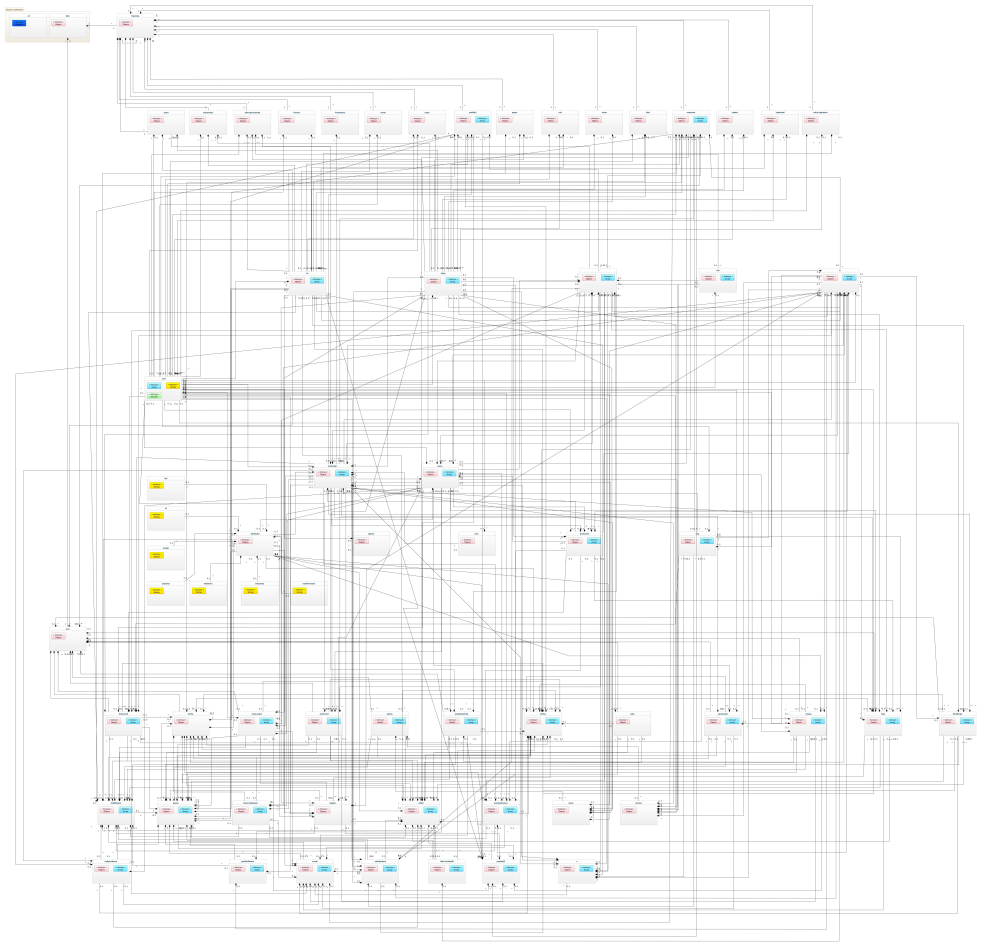
Naopak druhé schéma popisuje detailní strukturu děl a vychází z analýzy v předchozí sekci, která se týkala 1700 digitalizovaných sbírek ÚČL. Toto schéma obsahuje stejný základ jako předchozí schéma. Navíc však obsahuje všechny datové položky, které se mohou ve sbírce vyskytnout a také popisuje vazby mezi nimi. Toto schéma není vhodné využít pro validaci dokumentů. Pro ÚČL je však důležité, jelikož popisuje všechna existující díla a na jeho základě je možné snadněji vymýšlet možné úpravy struktury.

Na obrázku níže je zachyceno schéma pomocí objektového diagramu, kde objekty představují jednotlivé datové položky. Mezi jednotlivými objekty jsou příslušné vazby 1:1 a 1:N. V horní části schématu jsou převážně hlavičkové údaje, v dolní pak položky týkající se textu. Z obrázku je patrné, že schéma je velice rozsáhlé a v této podobě z něj nelze nic vyčíst. Jedná se pouze o ukázku, obrázek v plné kvalitě lze nalézt na přiloženém DVD disku ve složce `/Schema`.

Obrázek představuje pouze grafickou podobu modelu. Samotné schéma bylo vytvořeno pomocí konceptu JSON Schema, které je taktéž k dispozici na přiloženém DVD disku. Jelikož se datové položky ve struktuře děl velmi často opakují, byly popsány pomocí `definitions`. V případech více datových typů bylo využito klíčové slovo `anyOf`. Toto schéma lze například využít pro úpravu

samotných sbírek. Existují totiž nástroje, které jsou schopny vygenerovat formulář na úpravu JSON dokumentu právě na základě JSON Schema.

Obrázek 3.2: Komplexní společné schéma [3]



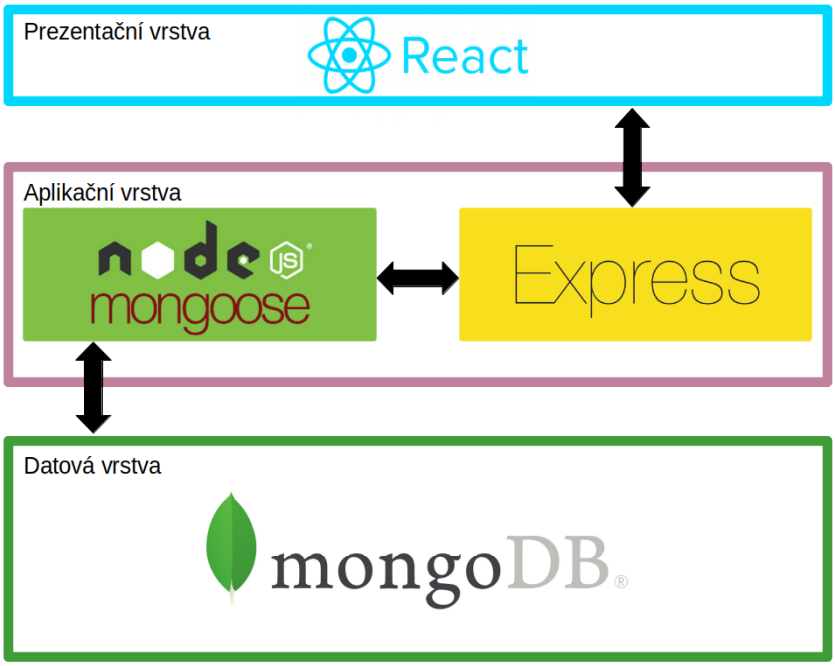
3.2 Architektura

Při návrhu aplikace je potřeba zvolit vhodnou architekturu, která bude splňovat veškeré požadavky na aplikaci. ÚČL specifikovalo pouze dva nové požadavky týkající se architektury (viz 1.2). Aplikace musí mít datové úložiště pro uložení sbírek a musí být snadno rozšiřitelná. Zároveň z předchozího řešení vyplývá, že se musí jednat o webovou aplikaci.

Všechny tyto požadavky splňuje vícevrstvá architektura skládající se z několika vzájemně vyloučených vrstev, které spolu komunikují pomocí definovaného rozhraní. Vrstvy je možné libovolně přidávat nebo měnit, aniž by to

mělo dopad na fungovaní celé aplikace.

Obrázek 3.3: Architektura aplikace



Pro řešení ÚČL je vhodná třívrstvá architektura zobrazená na předchozím obrázku. Třívrstvá architektura je v dnešní době nejpoužívanější pro implementaci webových aplikací. První vrstvou této architektury je vrstva prezentační, která obsahuje grafické uživatelské rozhraní. Pomocí této vrstvy budou moci uživatelé zobrazovat informace o jednotlivých sbírkách, provádět registraci či přihlášení, případně si budou moci prohlížet své vytvořené seznamy. Tato vrstva kontroluje uživatelské vstupy, avšak neprovádí žádné výpočty ani zpracování dat. Pro tuto vrstvu byla zvolena technologie ReactJS.

Druhou část aplikace představuje aplikační vrstva. Tato vrstva bývá často označována jako tzv. prostřední vrstva (*middleware*) či aplikační server a provádí operace a výpočty nad vstupními požadavky a daty. Prezentační vrstva komunikuje s aplikační pomocí komunikačního protokolu, kterým může být například protokol HTTP[37]. Pomocí tohoto protokolu si mohou i předávat data. Pro implementaci této vrstvy byly zvoleny technologie Express.js, Node.js a Mongoose.

Nyní zbývá doplnit pouze datovou vrstvu, se kterou komunikuje aplikační vrstva. Tato vrstva je v praxi tvořena databázovým systémem, který uchovává

[37]Hypertext Transfer Protocol

data, zaručuje jejich konzistenci a zpřístupňuje data aplikační vrstvě. Jedná se o nejnižší vrstvu architektury. Pro její implementaci byla použita technologie MongoDB.

Zmíněné technologie tvoří dohromady koncept zvaný *MERN Stack*, který byl navržen pro jednoduchý a plynulý vývoj webových aplikací. Tyto technologie jsou popsány v jedné z následujících sekcí (viz 3.5). Tato práce se zabývá pouze implementací backendové části aplikace, proto nebude implementována prezentační vrstva. Detailní analýzou a návrhem uživatelského rozhraní v rámci prezentační vrstvy se zabývá Filip Hladej ve své bakalářské práci.

3.3 Návrh datového úložiště

Jedním z nejdůležitějších požadavků ÚČL je implementace nového datového úložiště, do kterého bude možné uložit 1700 digitalizovaných sbírek. Jak již bylo zmíněno, pro analýzu dat byly sbírky převedeny z formátu XML do formátu JSON a následně byly tyto JSON dokumenty nahrány do jedné kolekce databáze MongoDB. Tento způsob uložení knih je velmi jednoduchý a efektivní. JSON dokumenty lze navíc snadno převést do původního formátu XML.

Pro potřeby ÚČL však jedna kolekce se sbírkami básní nestačí. Jedním z dalších požadavků ÚČL je ukládání starších verzí sbírek. Pokud například dojde k úpravě nebo ke smazání sbírky, měla by být uložena předchozí verze. Dále by aplikace měla poskytovat vytvoření uživatelských účtů pro jednotlivé uživatele. Ti by pak měli mít možnost pracovat se samotnými sbírkami básní, ale také by jim mělo být umožněno vytvořit si vlastní seznam se sbírkami. S tímto seznamem pak dále budou moci pracovat a budou moci analyzovat sbírky v tomto seznamu. Analýza děl vyžaduje častou práci se samotným textem sbírek, jelikož aplikace by měla umožnit například vypsat abecední či frekvenční slovník pro sbírky v uloženém seznamu.

V následující sekci jsou popsány kolekce, které bude databáze obsahovat na základě výše zmíněných požadavků.

3.3.1 Kolekce v databázi

Jak již bylo uvedeno, jedna kolekce v databázi slouží pro uložení samotných sbírek básní. Tato kolekce je nazvána `books` a obsahuje datové položky s následujícím názvem:

- **_id** - unikátní identifikátor knihy vygenerovaný systémem MongoDB
- **dilo** - vnořený objekt obsahující celou knihu.

Dále je potřeba vytvořit kolekci `users`, která bude uchovávat informace o jednotlivých uživatelích. U nich je potřeba evidovat tyto datové položky:

- **_id** - email uživatele
- **firstName** - křestní jméno uživatele
- **lastName** - příjmení uživatele
- **role** - číselná hodnota určující roli uživatele (1 - Čtenář, 2 - Editor, 3 - Redaktor)
- **password** - zahašované heslo uživatele.

Pro seznamy knih, které si uživatelé budou vytvářet, je vytvořena kolekce `bookLists`. V té jsou evidovány tyto položky:

- **_id** - unikátní identifikátor seznamu vytvořený databázovým systémem MongoDB
- **books** - seznam identifikátorů knih
- **name** - název seznamu
- **userId** - identifikátor (email) uživatele.

Pro ukládání starších verzí knih je vytvořena kolekce `bookVersions`. Do té jsou ukládány struktury dokumentů z kolekce `books`. Datové položky jsou tedy následující:

- **_id** - unikátní identifikátor verze knihy vytvořený systémem MongoDB
- **bookId** - identifikátor knihy
- **dilo** - vnořený objekt obsahující celou knihu před úpravou/smazáním
- **expirationDate** - datum a čas změny/smazání.

Jelikož je v aplikaci vyžadována častá práce s celým textem knihy a získání textu z dokumentů uložených v kolekci `books` není úplně triviální, je vytvořena ještě jedna kolekce s názvem `bookTexts` pro samotné texty knih. Kromě textu jsou do kolekce uloženy i základní statistické údaje. Struktura dokumentů v této kolekci je následující:

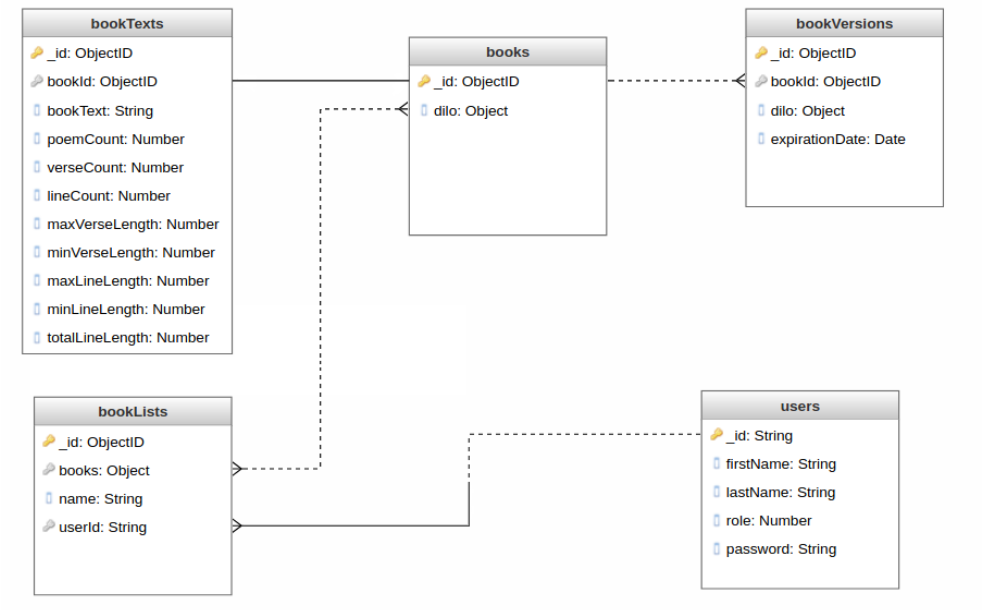
- **_id** - unikátní identifikátor textu knihy vytvořený systémem MongoDB
- **bookId** - identifikátor knihy
- **bookText** - text knihy
- **poemCount** - počet básní v knize
- **verseCount** - počet slok v knize

- **lineCount** - počet veršů v knize
- **maxVerseLength** - maximální počet veršů ve sloce
- **minVerseLength** - minimální počet veršů ve sloce
- **maxLineLength** - maximální počet slov ve verši
- **minLineLength** - minimální počet slov ve verši
- **totalLineLength** - celkový počet slov ve verších

Navržené kolekce obsahují jednu zvláštnost, která na první pohled může působit dojmem špatného návrhu databáze. Všechny datové položky mají název v anglickém jazyce, avšak jedna položka obsahuje název `dilo`, který je v jazyce českém. Tento název je v češtině ponechán úmyslně, jelikož přesně odpovídá XML elementu v původních dokumentech ÚČL.

Na následujícím obrázku jsou znázorněny jednotlivé kolekce a jejich datové položky. Pro každou kolekci je nutné vytvořit JSON Schema, které bude validovat vkládané či upravované dokumenty. Toto schéma bude kontrolovat uvedené datové typy položek dokumentů. Kromě toho by také mělo kontrolovat formát položky `_id` u uživatele, zda odpovídá emailové adrese.

Obrázek 3.4: Databázový model [3]



3.3.2 Indexy

U každé kolekce budou využívány dotazy pro vkládání, úpravu či mazání dokumentů. Nejčastěji však budou používány dotazy pro čtení dokumentů. Kromě dotazů pro získání seznamů všech dokumentů v kolekcích a vyhledávání dokumentů podle primárního klíče `_id`, bude třeba provádět selekci záznamů i na základě jiných datových položek. V kolekci `users` bude třeba filtrovat uživatele i podle hesla, v kolekci `bookLists` podle datové položky `userId` a v kolekcích `booksTexts` a `bookVersions` zase podle datové položky `bookId` případně `expirationDate`. V kolekci `bookTexts` budou využívány dotazy pro práci s textem sbírek básní.

Pro efektivní provádění těchto dotazů bude nutné vytvořit indexy nad některými datovými položkami. Vytvoření indexů výrazně sníží počet dokumentů, které musí databázový systém MongoDB projít, aby mohl správně vyhodnotit daný dotaz. Nad položkami `_id` vytváří systém indexy automaticky. V tabulce níže je uveden seznam všech indexů, které bude třeba v databázi vytvořit.

Tabulka 3.8: Indexy v databázi

Název kolekce	**Název klíče**	**Typ indexu**
users	password	Single Field Index
bookTexts	bookId	Single Field Index
bookTexts	bookText	Text Index
bookVersions	bookId	Single Field Index
bookVersions	expirationDate	Single Field Index
bookLists	userId	Single Field Index

3.4 Návrh API

Aby mohli klienti číst data a pracovat s nimi, potřebují rozhraní API, které použijí pro interakci s aplikací. Většina moderních webových aplikací takové rozhraní zveřejňuje. Dobře navržené rozhraní API musí být platformě nezávislé. To znamená, že by toto rozhraní měl být schopný využít jakýkoliv klient nezávisle na samotné implementaci rozhraní API. Dále musí umožňovat vývoj webových služeb, které poskytuje. Rozhraní by tedy mělo mít možnost přidávat nové funkce, aniž by bylo potřeba měnit implementaci klientů. Všechny funkce by měly být zjistitelné, aby je mohl klient bez problému využít.

V dnešní době je k dispozici více variant pro implementaci rozhraní API. Nejznámější z nich jsou SOAP[38] a REST[39]. Pro tuto práci byl zvolen REST, který představuje architektonický přístup k navrhování webových služeb. Sa-

[38]Simple Object Access Protocol
[39]Representational state transfer

motný REST je nezávislý na aplikačním protokolu. Většina API je však implementována pomocí protokolu HTTP, takové rozhraní také nazýváme RESTful API. Rozhraní REST je použitelné pro jednotný a snadný přístup ke zdrojům, kterými jsou většinou data. Každý zdroj má vlastní identifikátor URI[40] a může mít libovolnou reprezentaci. Klient tak nepracuje přímo se zdrojem, ale pouze s jeho reprezentací. Pro práci se zdroji poskytuje REST čtyři základní CRUD operace. U RESTful API se jedná o HTTP metody GET, POST, DELETE a PUT. REST také splňuje princip zvaný HATEOAS[41], tedy stav aplikace je určen pomocí URL a další možné stavy lze získat pomocí odkazů. [31]

HTTP je protokol, který slouží pro komunikaci mezi klientem a serverem. Klienta si lze představit například jako webový prohlížeč, který posílá webovému serveru požadavky. HTTP požadavek musí mít specifikovanou HTTP metodu, URL a také může obsahovat hlavičky. V následující ukázce je uveden jednoduchý příklad HTTP požadavku, který přistupuje ke zdroji `/books` na serveru `ucl.cz`.

```
GET /books
Host: ucl.cz
```

Server odesílá klientovi HTTP odpověď, která se skládá ze stavového kódu, hlaviček a těla. Stavový kód je číslo, které určuje, zda se požadavek podařilo splnit, nebo nikoliv. Následující tabulka obsahuje základní rozdělení stavových kódů. [32]

Tabulka 3.9: Stavové kódy v protokolu HTTP

Stavový kód	**Sémantika**
1xx	Informační charakter zprávy
2xx	Úspěch při zpracování požadavku
3xx	Přesměrování klienta
4xx	Chyba na straně klienta
5xx	Chyba na straně serveru

V ukázce níže je uveden příklad HTTP odpovědi, která značí úspěšné zpracování požadavku.

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    author: "Jan Neruda",
    title: "Hřbitovní kvítí"
}
```

[40] Uniform Resource Identifier
[41] Hypermedia as the Engine of Application State

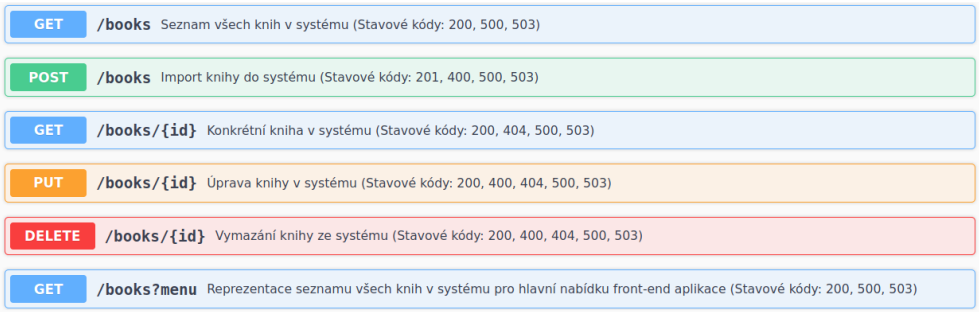
Kromě výše zmíněných principů umožňuje REST například cachování odpovědí pomocí hlaviček `Last-Modified` a `ETag` nebo umožňuje řešit tzv. *concurrency control* představující souběžný přístup k datům. Tyto principy však nejsou v této práci rozebrány, jelikož jejich implementace bude provedena později.

V následujících sekcích jsou popsány jednotlivé zdroje a metody, které lze nad těmito zdroji použít. Tento návrh API byl vytvořen na základě požadavků frontendu aplikace.

3.4.1 Zdroje pro knihy

Tato sekce obsahuje návrh API pro zdroj `/books`. Kromě klasických operací pro manipulaci s daty vyžaduje frontend metodu, která vrátí seznam knih ve speciálním formátu, který bude využitelný pro hlavní nabídku knih v uživatelském rozhraní. Návrh API pro zdroj `/books` vypadá následovně:

Obrázek 3.5: REST metody nad zdroji pro knihy



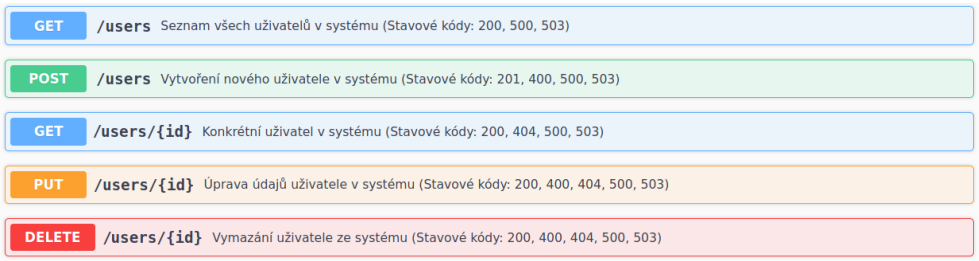
Na obrázku je na levé straně uvedena HTTP metoda, kterou je nutné použít, dále URI, popis a stavové kódy. Požadavek s metodou POST na zdroj `/books` slouží pro import XML sbírek. V těle požadavku je nutné poslat celý obsah sbírky XML formátu. Při zpracování této metody by mělo dojít k transformaci dat a jejich následnému uložení do databáze. Požadavek PUT také musí obsahovat ve svém těle data s upravenou knihou. Ty už jsou však ve formátu JSON. V některých URI si můžeme všimnout tzv. *path* parametru `id`. Jedná se o unikátní identifikátor zdroje. U poslední metody GET je tzv. *query* parametr `menu`. Tato metoda je podobná jako GET metoda bez query parametru, akorát slouží k získání dat ve speciálním formátu, který potřebuje frontend aplikace pro hlavní nabídku knih.

Jelikož by navrhnuté REST API mělo splňovat princip HATEOAS, vrací metody kromě požadované reprezentace zdroje také tzv. linky obsahující metody, které lze s danými zdroji dále provést. Toto platí i pro navrhnuté API v následujících sekcích. K tomuto tématu je uvedeno více v jedné z následujících sekcí (viz 4.2.3).

3.4.2 Zdroje pro uživatele

Tato sekce obsahuje návrh API pro zdroj `/users`. Návrh obsahuje klasické CRUD operace, tedy vkládání, modifikaci, mazání a získání uživatelů. Frontend aplikace nevyžaduje žádné další speciální operace. Po registraci dojde k vytvoření nového uživatele, následně je vyžadován buď seznam uživatelů, nebo jeden konkrétní uživatel. Počítá se s tím, že uživatel bude chtít své údaje změnit, případně bude požadovat odstranění údajů z databáze. Návrh je zobrazen na následujícím obrázku:

Obrázek 3.6: REST metody nad zdroji pro uživatele



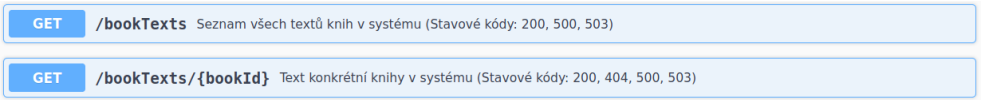
U metod POST a PUT na zdroj `/users` je třeba do těla požadavku vložit data vytvářeného či upravovaného uživatele. U některých metod je součástí URI path parametr `id`, který představuje unikátní identifikátor zdroje.

3.4.3 Zdroje pro texty a verze knih

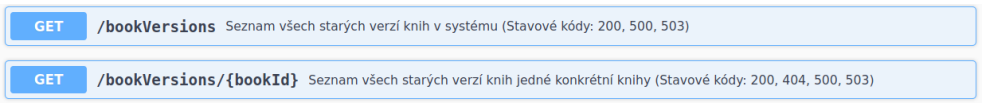
Tato sekce obsahuje návrh API pro texty a starší verze knih. Zde oproti předchozím zdrojům nejsou potřeba metody pro vytváření, modifikaci či mazání zdrojů. Text knihy je vytvářen automaticky při importu knihy. Obdobně funguje i úprava či smazání tohoto zdroje. Klient by neměl mít možnost smazat či upravit samotný text knihy. Má možnost pouze změnit původní zdroj knihy, čímž dojde k aktualizaci i tohoto zdroje pro text knihy.

Starší verze knihy je vytvořena vždy při úpravě či mazání aktuální verze v kolekci `books`. Uložené starší verze není možné nijak upravit ani smazat. V případě, že je vyžadováno vymazání textu nebo starší verze knihy z databáze manuálně, musí tento krok provést administrátor databáze. Návrh API tedy obsahuje pouze metody GET pro čtení zdrojů:

Obrázek 3.7: REST metody nad zdroji pro texty knih



Obrázek 3.8: REST metody nad zdroji pro verze knih



U dvou metod si můžeme všimnout path parametru `bookId`. Nejedná se o identifikátor konkrétních textů či verzí knih, nýbrž o identifikátor zdroje pro knihu.

3.4.4 Zdroje pro seznamy knih

Další API, které je potřeba navrhnout, se týká zdroje `bookLists`. Seznamy knih se vážou přímo ke konkrétnímu uživateli (čtenáři), proto lze se seznamy pracovat pouze v rámci jednoho uživatele. Zde frontend aplikace vyžaduje metody pro vytvoření, úpravu, smazání a zobrazení seznamů. U daných seznamům je také třeba vytvářet abecední či frekvenční slovníky, generovat různé statistiky, vyhledávat fráze fulltextově nebo kontextově a také je třeba generovat souhrnný text všech knih v seznamu. Návrh API proto vypadá následovně:

Obrázek 3.9: REST metody nad zdroji pro seznamy knih

Při použití POST a PUT metod je třeba vložit do těla požadavku data obsahující konkrétní seznam knih. V URI lze najít path parametry `userId` představující identifikátor konkrétního uživatele a `bookListId`, což je identifi-

kátor seznamu knih. Pro získání abecedního, resp. frekvenčního slovníku k danému seznamu knih lze použít query parametr `alphabeticDictionary`, resp. `frequencyDictionary`. Dále lze použít query parametry `stats` pro statistiky seznamu, `fulltext` pro fulltextové vyhledávání, `context` pro kontextové vyhledávání a `allTexts` pro získání všech textů knih v seznamu. Hodnotou parametrů `fulltext` a `context` je vyhledávaná fráze. Výsledkem fulltextového a kontextového vyhledávání je seznam identifikátorů knih v seznamu, ve kterých se nachází hledaná fráze a počet výskytů.

Jelikož patří seznamy knih k uživatelům, musí dojít v případě odstranění uživatele i k odstranění jeho seznamů. Dále seznamy obsahují referenci na knihu. V případě odstranění knihy z databáze se nabízí hned několik scénářů.

1. Smazaná kniha je odstraněna ze všech seznamů, ve kterých se vyskytuje. Čtenář je upozorněn, že některá z uložených knih již není v seznamu.

2. Smazaná kniha v seznamu zůstává, ale čtenář s ní nemůže pracovat. Ze seznamu jí musí odstranit sám čtenář.

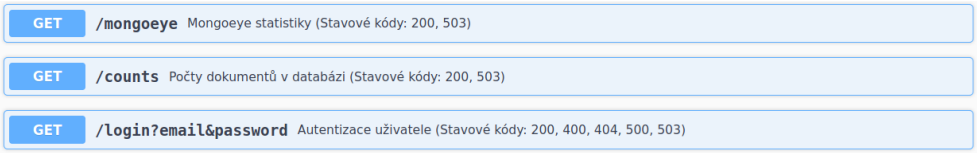
3. Při smazání knihy nedojde ke smazání zdroje se samotným textem. Čtenář je upozorněn, že kniha v jeho seznamu byla smazána, může s ní však dále pracovat. Systém pravidelně kontroluje, zda je reference na smazanou knihu v nějakém seznamu. Pokud v žádném seznamu není, je smazán i zdroj pro text knihy.

O tom, která z těchto variant bude v praxi použita musí rozhodnout ÚČL. V rámci této práce je implementována první varianta.

3.4.5 Ostatní zdroje

Následující API slouží pouze pro účely analýzy a testování. GET metoda na zdroj `/counts` vrací počty dokumentů v jednotlivých kolekcích. GET metoda na zdroj `/mongoeye` vrací kompletní analýzu kolekce knihy a je velmi důležitá v případě jakéhokoliv rozšíření API pro zdroj knih. Poslední metoda na zdroj `/login` slouží k autentizaci uživatele. Z bezpečnostních důvodů nevrací žádná z metod nad zdrojem `/users` hash hesla uživatele. K zjištění, zda jsou uživatelské údaje správné, lze použít právě tuto metodu.

Obrázek 3.10: REST metody nad ostatními zdroji



3.4.6 Stavové kódy

Po zpracování HTTP požadavku odesílá server odpověď, ve které je obsažen mimo jiné i stavový kód, který upřesňuje, jak byl požadavek vyřízen. V následující tabulce je seznam použitých stavových kódů.

Tabulka 3.10: Sémantika použitých stavových kódů

Stavový kód	Název	Použití
200	OK	Úspěšné provedení požadavku
201	Created	Úspěšné vytvoření objektu
400	Bad Request	Chybně zapsaný požadavek
404	Not Found	Nelze najít daný zdroj
500	Internal Server Error	Neočekávaná chyba serveru
503	Service Unavailable	Chyba připojení k databázi

3.5 Použité technologie

Tato sekce krátce pojednává o použitých technologiích. V sekci týkající se architektury (viz 3.2) byla nastíněna kolekce softwaru pro vývoj webových aplikací MERN Stack. MERN je zkratka skládající se ze čtyř technologií, jmenovitě MongoDB, Express.js, React.js a Node.js. Je tedy jisté, že hlavním programovacím jazykem, který bude použit pro implementaci řešení, je Javascript. Systém MongoDB v této kapitole již nebude popisován, jelikož mu byla věnována celá sekce (viz 2.3).

3.5.1 Javascript

Javascript je objektově orientovaný skriptovací jazyk, který je svou syntaxí velmi podobný jazykům C++ nebo Java. Funkčně a principiálně se však jedná o naprosto odlišný programovací jazyk. Dá se využít jak na straně klienta, tak na straně serveru. Pro implementaci řešení byl použit pro frontend i pro backend aplikace. Javascript je multiplatformní jazyk, a lze jej použít na jakémkoliv operačním systému. Další vlastností tohoto programovacího jazyka je to, že je *case sensitive*. Při psaní kódu je tedy třeba dávat pozor na psaní malých a velkých písmen.

Javascript je v dnešní době jeden z nejvyužívanějších programovacích jazyků pro tvorbu webových aplikací. Pro tvorbu těchto aplikací existuje mnoho knihoven a frameworků, mezi které patří například React nebo Angular. Výhodou tohoto jazyka je i to, že je podporován téměř všemi prohlížeči. V každém prohlížeči se však může chovat trochu odlišně. Například ve starším webovém prohlížeči Internet Explorer jsou podporovány jiné metody než u prohlížečů Google Chrome nebo Firefox. [33]

3.5.2 React.js

React.js je javascriptová open-source knihovna od společnosti Facebook, která umožňuje tvorbu uživatelského rozhraní. Aplikace vytvořené v Reactu se skládají z HTML[42] elementů se zapouzdřenou funkcionalitou, kterým se říká komponenty (*components*). Každá komponenta má své vlastnosti (*props*) a svůj vnitřní stav (*state*). U těchto komponent lze snadno předpovídat jejich chování a lze je také velmi dobře ladit. Z těchto důvodu je tato knihovna velmi populární.

V praxi se React používá i v kombinaci s jinými knihovnami. Vždy se ale stará o vykreslení uživatelského rozhraní. Často je používán pro tvorbu SPA[43] nebo pro vykreslování webových stránek na straně serveru. Také je nutné zmínit *React Native* pro tvorbu nativních mobilních aplikací pomocí Javascriptu a Reactu. [34]

K vytvoření React webové aplikace je vyžadován Node.js a NPM[44]. Jak již bylo řečeno, tato technologie byla vybrána pro implementaci frontendu aplikace a podrobněji se jí zabývá bakalářská práce Filipa Hladeje.

3.5.3 Node.js

Pro implementaci aplikačního serveru webové aplikace byly vybrány technologie Express.js a Node.js. Technologie Node.js je prostředí, které umožňuje spustit kód i mimo internetový prohlížeč a používá se primárně pro tvorbu aplikačních serverů. Node.js má schopnost obsloužit několik připojených klientů najednou, proto je v dnešní době velmi oblíbený pro tvoru aplikačních serverů. Pro psaní kódu se používá převážně čistý Javascript. Můžeme ale využít jakékoliv Node.js knihovny, které lze do aplikace nainstalovat pomocí NPM. Nejnovější verze Node.js také nabízí všechny funkcionality z Javascript ES6.

Node.js server je tvořen tzv. *Event Loop*, do které přicházejí veškeré požadavky ze strany klienta. Tyto požadavky představují události a jsou postupně přiřazovány dostupným vláknům aplikace. V případě čtení souborů či databáze jsou volání těchto operací také zařazeny do Event Loop a je zde použit koncept tzv. *Nonblocking I/O*. Dá se tedy říci, že vše je řešeno pomocí této „smyčky událostí“.

Součástí instalace Node.js je také balíčkový systém NPM, který lze snadno ovládat z příkazové řádky. Pomocí něj lze instalovat jakékoliv Node.js knihovny a závislosti. Aktuálně lze použít více než půl milionu knihoven a balíčků. [35]

[42] Hypertext Markup Language
[43] Single Page Application
[44] Node Package Manager

3.5.4 Express.js

Express.js je minimalistický a flexibilní Node.js framework pro tvorbu webových aplikací, který poskytuje robustní sadu funkcí pro webové a mobilní aplikace. Pomocí této technologie lze velmi snadno implementovat navrhnuté REST API, jelikož framework nabízí nesčetné množství nástrojů pro práci s HTTP a pro implementaci middlewaru. Express.js sice poskytuje funkce pouze pro tuto oblast, nicméně umožňuje použít i veškeré Node.js funkcionality. Využívá se ve známých konceptech MERN, MEAN[45] nebo MEVN[46], tedy v kombinaci s databázovým systémem MongoDB, Node.js a některým z javascriptových frameworků pro tvorbu uživatelského rozhraní. [36]

3.5.5 Mongoose

V navrhnuté architektuře máme aplikační a datovou vrstvu a tyto vrstvy spolu musejí nějak komunikovat. Přesně k těmto účelům slouží Node.js ODM[47] knihovna Mongoose. Pomocí této knihovny lze spravovat vztahy mezi daty, validovat schémata dat nebo třeba převádět data z databáze MongoDB do javascriptových objektů a naopak. Mongoose pracuje v asynchronním prostředí a podporuje tzv. *Callbacks* a *Promises* v Javascriptu.

Tuto knihovnu lze naistalovat standardně pomocí NPM. Následně je nutné vytvořit připojení k databázi MongoDB pomocí příkazu `mongoose.connect`. Mongoose podporuje všechny příkazy jako databázový systém MongoDB. Některé příkazy však mohou mít nepatrně odlišnou syntaxi. [37]

[45] technologie MongoDB, Express.js, Angular.js a Node.js
[46] technologie MongoDB, Express.js, Vue.js a Node.js
[47] Object Data Modeling

Kapitola 4

Implementace řešení

Tato kapitola obsahuje popis samotné implementace řešení pro potřeby ÚČL. Je zde rozebrána implementace datového úložiště a REST API. V další sekci je zmíněna dokumentace implementovaného REST API a její možnosti využití. V posledních sekcích je popsáno nasazení aplikace na virtuální server a také jsou rozebrány možnosti budoucího rozšíření aplikace.

4.1 Vytvoření datového úložiště

Aby bylo možné používat databázi MongoDB, musí běžet tzv. daemon proces `mongod`. Tento proces běží primárně na TCP[48] portu 27017 a zpracovává požadavky klientů, spravuje přístup k datům a provádí potřebné operace na pozadí. V případě potřeby distribuce dat pomocí shardingu lze využít instance `mongos`, které poskytují rozhraní mezi klientskými aplikacemi a shardovaným clusterem.

4.1.1 Příprava kolekcí

V kapitole týkající se návrhu řešení byly popsány jednotlivé kolekce, které je potřeba vytvořit (viz 3.3). Samotné vytvoření kolekcí bylo realizováno v konzoli `mongo` pomocí příkazů v následujícím tvaru:

```
db.createCollection( <collectionName>, {
    "validator": {
        "$jsonSchema": <JSONSchema>
    },
    "validationAction": "error",
    "validationLevel": "strict"
} )
```

[48] Transmission Control Protocol

Za `<collectionName>` byla postupně dosazena jednotlivá jména kolekcí, konkrétně tedy `books`, `users`, `bookTexts`, `bookVersions` a `bookLists`. Položka `<JSONSchema>` pak představuje konkrétní JSON Schema pro danou kolekci. Jednotlivá schémata lze nalézt na přiloženém DVD disku ve složce s názvem `/JSONSchemas`. Každé JSON Schema obsahuje veškerá validační pravidla a může tak sloužit k validaci vkládaných a upravovaných dokumentů. Datové položky `validationAction` a `validationLevel` zajistí, že nebude možné vložit nový či upravený dokument, který nebude splňovat všechna validační pravidla ve schématu. Pokud se klient pokusí nahrát dokument, který není validní, vrátí systém MongoDB chybovou hlášku s kódem 121.

Dále je potřeba v jednotlivých kolekcích nad určitými klíči vytvořit indexy. Ty jsou vytvořeny následujícím způsobem:

```
db.users.createIndex( { "password": 1 } );
db.bookTexts.createIndex( { "bookId": 1 } );
db.bookTexts.createIndex(
    {"bookText" : "text"},
    {"default_language": "none"}
);
db.bookVersions.createIndex( { "bookId": 1 } );
db.bookVersions.createIndex( { "expirationDate": 1 } );
db.bookLists.createIndex( { "userId": 1 } );
```

Všechny indexy jsou tedy typu *Single Field Index*. Jedinou výjimku tvoří index vytvořený v kolekci `bookTexts` nad klíčem `bookText`. Ten představuje *Text Index*. U textových indexů je jako výchozí nastaven anglický jazyk, český jazyk MongoDB nepodporuje. Pro rychlejší práci s diakritikou je lepší nastavit hodnotu datové položky `default_language` na `none`.

4.1.2 Používané dotazy

V předchozí sekci bylo popsáno, jak byla databáze vytvořena. Takto vytvořená databáze je připravena k použití. V této sekci jsou popsány dotazy, které implementované řešení využívá pro manipulaci s daty.

Jelikož po vytvoření neobsahuje databáze žádná data, jsou potřeba dotazy pro vkládání dokumentů. Tyto dotazy jsou používány pro každou kolekci a mají stejnou syntaxi jako v následující ukázce. Položka `<collection>` představuje název kolekce a `<data>` jsou vkládaná data.

```
db.<collection>.create( <data> )
```

Téměř všechna vkládaná data neobsahují identifikátor. Databázový systém tedy vytvoří `_id` datového typu `ObjectId`. Výjimku tvoří kolekce `users`, kde položka `_id` představuje email uživatele.

Data uložená v databázi je potřeba velmi často číst. V následující ukázce jsou čtyři dotazy, kdy první slouží k získání všech dat z dané kolekce. Velmi často však potřebujeme, aby záznamy obsahovaly pouze některé datové položky. Pomocí druhého dotazu v ukázce získáme pouze název knihy. Pro získání konkrétních dokumentů je použita syntaxe ze třetího uvedeného dotazu. Zde jsou vybrány pouze ty dokumenty, které mají uvedený identifikátor s hodnotou `<_id>`. Kritérií selekce může být libovolný počet. Příkladem je čtvrtý dotaz, který se používá například pro autentizaci uživatele.

```
db.<collection>.find( {} );
db.books.find({},{ "dilo.hlavicka.titul._text": 1 });

db.<collection>.find( { _id: <_id> } );
db.users.find( {
    "_id": <email>,
    "password": <password>
} );
```

Pro seznamy knih je mnohdy potřeba získat informace o knihách. V seznamech jsou však uloženy pouze reference na podmnožinu knih. Pro tyto podmnožiny jsou sestaveny dotazy, které pro selekci používají operátor `$or`. V následujícím dotazu představuje `<books>` pole selekčních kritérií ve tvaru `{"_id": <id>}`.

```
db.books.find(
    {
        "$or": <books>
    },
    {
        "dilo.hlavicka.titul._text": 1,
        "dilo.hlavicka.autor._text": 1
    }
)
```

Dokumenty je také třeba upravovat a mazat. U dotazů pro úpravu a mazání se vždy používá selekční kritérium pro výběr konkrétního dokumentu. Při úpravě je třeba navíc vložit ty datové položky, které je třeba upravit. K tomu se využívá operátor `$set`.

```
db.<collection>( { "_id": <_id> } );
db.<collection>( { "_id": <_id> }, { "$set": <data> });
```

Implementované řešení se neobejde ani bez agregačních dotazů. Tyto dotazy využívají koncept *Aggregation Pipeline* a slouží pro fulltextové vyhledávání a pro získání abecedních či frekvečních slovníků. Agregační dotaz pro

fulltextové vyhledávání se skládá ze 2 částí. V první části jsou vybrány pouze ty knihy, které jsou obsaženy v konkrétním seznamu. Ve druhé části je provedena projekce, díky které je výsledkem pouze seznam identifikátorů knih a počet výskytů hledané fráze.

```
db.bookTexts.aggregate([
    {
        "$match": {
            "$or": <books>
        }
    },
    {
        "$project": {
            "_id": "$bookId",
            "count": {
                "$subtract": [
                    {
                        "$size": {
                            "$split": [ "$bookText", phrase ]
                        }
                    },
                    1
                ]
            }
        }
    }
])
```

Agregovaný dotaz pro sestavení abecedního slovníku se skládá z poněkud více částí. V první části jsou opět vybrány pouze ty knihy, které jsou obsaženy v konkrétním seznamu. Následně jsou všechna velká písmena v textech knih převedena na malá písmena. Hned poté jsou z textů odstraněny všechny symboly tak, aby bylo možné provést rozdělení na jednotlivá slova. V ukázce níže je zobrazeno pouze odstranění symbolu pro nový řádek. Odstranění ostatních symbolů je v ukázce nahrazeno položkou `<otherReplacements>`. Dále jsou texty rozděleny a seskupeny podle jednotlivých slov. Počítá se s tím, že slova jsou oddělena mezerami. U každého slova je spočítán počet jeho výskytů ve všech textech. Ze seznamu slov jsou odstraněna ta slova, která se neskládají pouze ze znaků české abecedy. Díky tomu dojde například k odstranění prázdných řetězců, které mohou vzniknout v případě, že se v textu vyskytnou dvě mezery za sebou. V předposledním kroku je provedena projekce, která zajistí, aby se ve výsledku vyskytovala pouze samotná slova a jejich počet výskytů. Na závěr je pole seřazeno podle abecedy. Dotaz je zobrazen v ukázce na další straně.

```
db.bookTexts.aggregate([
    { "$match": { "$or": <books> } },
    {
        "$project":{
            "bookText": {
                "$toLower": "$bookText"
            }
        }
    },
    {
        "$project": {
            "bookText": {
                "$replaceAll": {
                    "input": "$bookText",
                    "find": "\n",
                    "replacement": " "
                }
            }
        }
    },
    <otherReplacements>,
    {
        "$project": {
            "bookText" : {
                "$split": ["$bookText", " "]
            }
        }
    },
    { "$unwind": "$bookText" },
    {
        "$group": {
            "_id":  "$bookText",
            "count" : {
                "$sum" : 1
            }
        }
    },
    {
        "$addFields": {
            "czechWord": {
                "$regexMatch": {
                    "input": "$_id",
                    "regex": /^[<czechChars>]+$/
                }
```

```
                }
            }
        },
        { "$match": { "czechWord": true } },
        { "$project": { "_id" : "$_id", "count": "$count" } },
        { "$sort": { "_id" : 1 } }
    ])
```

Stejný dotaz je možné použít i pro získání frekvenčního slovníku. V posledním kroku je však nutné seřadit slova sestupně podle jejich počtů výskytů v textech.

Díky vytvoření indexů jsou všechny zmíněné dotazy maximálně optimalizovány a umožňují rychlou a efektivní práci s uloženými daty. Díky indexům nemusí MongoDB procházet při selekci všechny dokumenty sekvenčně a může využít B-stromy. Indexy také usnadňují práci s texty knih.

4.1.3 Chybové kódy

Je třeba počítat s tím, že během manipulace s daty může dojít k určitým chybám. V této sekci jsou popsány možné chyby, které mohou nastat při provádění dotazů. Chyby způsobené nesprávnou syntaxí dotazů zde nebudou uvažovány.

Nejprve se podívejme na dotazy pro čtení dat. V aktuální konfiguraci databáze nenastávají žádné chyby při čtení dat. Pokud je v selekci uvedeno kritérium, kterému nevyhovuje žádný dokument, pak je výsledkem prázdné pole. Pokud je v projekci obsažen neexistující název datové položky, pak je tato položka ignorována. Obdobně to funguje i při mazání dokumentů. V případě, že se pokusíme smazat dokument, který neexistuje, databáze jednoduše nesmaže žádný záznam.

Během vkládání a úprav už však nějaké chyby nastat mohou. První chyba nastane v případě, že se klient pokusí vložit dokument, který nevyhovuje validačnímu schématu. V takovém případě MongoDB dokument neuloží a vrátí chybu s kódem 121 (*Document failed validation*). Druhá chyba může nastat v situaci, kdy se klient pokusí vložit dokument s položkou `_id`, kterou už má v kolekci jiný dokument. V takovém případě vrací MongoDB chybu s kódem 11000 (*E11000 duplicate key error collection*). Další chyba s kódem 10334 (*BSONObj size is invalid*) je vrácena v případě, že je velikost vkládaného dokumentu větší než 16 MB. Při úpravě dat je třeba se vyhnout modifikaci datové položky `_id`. V opačném případě zareaguje MongoDB chybou s kódem 66 (*Immutable field '_id'*).

Uvedené chyby nejsou jediné, které mohou nastat. Jedná se však o chyby, které v současné implementaci mohou vzniknout nejčastěji. Další chyby s kódy 67 a 68 mohou vzniknou během vytváření indexů. Se všemi chybami systému MongoDB implementace aplikace počítá a v případě jejich vzniknu odpoví aplikační server odpovídající chybovou hláškou.

4.2 Zdrojový kód aplikačního serveru

V této sekci jsou rozebrány implementační detaily samotného aplikačního serveru, který obsahuje REST API, které je zpřístupněno frontendu aplikace. V další části jsou popsány možnosti konfigurace serveru. V poslední části této sekce jsou uvedeny knihovny, které byly využity pro implementaci řešení. Implementované řešení lze nalézt na přiloženém DVD disku ve složce s názvem `BookApp`. Tato složka obsahuje mimo jiné i adresář `src/main`, který obsahuje veškeré zdrojové kódy aplikačního serveru. Dále se zde vyskytuje také adresář `src/test`, ve kterém jsou všechny dostupné jednotkové testy.

4.2.1 REST API

V předchozí kapitole bylo navrhnuto REST rozhraní umožňující frontendu pracovat s daty (viz 3.4). Implementace se drží tohoto návrhu a nabízí téměř všechny navrhnuté metody. Jedinou výjimku představuje REST metoda pro kontextové vyhledávání v dílech, jelikož ÚČL řádně nedefinovalo, jak se má tato metoda chovat. Proto tato metoda nebyla implementována.

Jádrem celého rozhraní je javascriptová knihovna Express.js, která nabízí veškeré prostředky pro HTTP komunikaci mezi klientem a serverem. Toto rozhraní bylo vytvořeno pomocí příkazu `var app = express()`. Následně pomocí metody `app.listen(<port>)` bylo nastaveno, na jakém TCP portu má server poslouchat příchozí požadavky. Jako výchozí hodnota TCP portu byla zvolena hodnota 8888. Kromě nastavení portu bylo také třeba povolit veškeré HTTP požadavky metody GET, PUT, POST a DELETE.

Komunikaci aplikace s databází zajišťuje druhá velmi důležitá knihovna Mongoose. Aby bylo možné s databází komunikovat, je nutné při spuštění aplikace vytvořit připojení k databázovému systému MongoDB k databázi `books`:

```
var mongo = require("mongoose");
var db = mongo.connect("mongodb://127.0.0.1:27017/books",
  function(err, response){
    if( err ){
        console.log('Failed to connect to ' + db);
    }
    else{
        console.log('Connected to ' + db, ' + ', response);
    }
});
```

Pro práci s jednotlivými kolekcemi je třeba definovat Mongoose Schema. Každé schéma se mapuje na kolekci MongoDB a definuje strukturu jednotlivých dokumentů v této kolekci. U každé položky ve schématu je třeba určit její `SchemaType`. Oproti klasickým datovým typům v JSON Schema lze použít

typ `SchemaType.Mixed`, jehož hodnota může být jakéhokoliv datového typu. Nad jednotlivými položkami lze také definovat indexy. Toto schéma slouží převážně pro mapování objektů z kolekcí MongoDB na objekty Javascriptu. Je možné toto schéma použít i pro validaci dokumentů. Současná implementace však k validaci dokumentů používá validátory obsahující JSON Schema. Ty lze najít ve zdrojových kódech ve složce `validators`.

Po definici Mongoose schématu je třeba vytvořit také Mongoose Models. Ty představují efektivní konstruktory zkompilované z definic jednotlivých schémat. Instancí těchto modelů je pak samotný dokument. Pomocí modelů tedy čteme a vytváříme dokumenty v databázi MongoDB. Následující ukázka obsahuje vytvoření Mongoose schématu a modelu pro dokumenty v kolekci `users`. Všechny definice modelů lze nalézt ve zdrojových kódech ve složce `db/dbModels`.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var userSchema = new Schema(
    {
        _id: Schema.Types.String,
        firstName: Schema.Types.String,
        lastName: Schema.Types.String,
        role: Schema.Types.Number,
        password: Schema.Types.String
    }
);
userSchema.index({ password: 1 }, { background: false });
var userModel = mongoose.model('user', userSchema, 'users');
userModel.schema.options.autoIndex = true;
```

Samotné REST API se nachází ve zdrojových kódech ve složce `server`. Zde v souboru `server.js` jsou definována API podle návrhu v předchozí kapitole. Například API metoda pro získání seznamu všech uživatelů je zobrazena v ukázce níže.

```
app.get("/users", function(req,res){
    if(mongo.connection.readyState === 1){
        userModel.find({}, function(err, data){
            if(err){
                customLogger(winston, err, 'server-1006');
                res.sendStatus(500);
            }
            else{
                res.status(200).send(data);
            }
```

```
            });
        }
        else{
            res.status(503).send('Unable to access database.');
        }
    });
```

U každého implementovaného API je uvedena HTTP metoda a URI. V případě, že klient odešle HTTP požadavek s odpovídající metodou na uvedený zdroj, pak je tento požadavek uložen do proměnné `req` a je provedena zobrazená funkce. V každé funkci je nejprve kontrolováno, zda je připojení k databázi MongoDB stále aktivní. V opačném případě je na klienta odeslána odpověď se stavovým kódem 503 (*Service Unavailable*). Pokud je ale připojení aktivní, pak je pomocí Mongoose modelu definován dotaz pro získání dat nebo provedení potřebné operace. Pokud dotaz skončí očekávanou chybou vzniklou na straně klienta, pak je klientovi odeslána odpověď se stavovým kódem 400 (*Bad Request*) případně 404 (*Not Found*). Pokud při dotazu nastane neočekávaná chyba, pak je klientovi zaslána odpověď, která má stavový kód s hodnotou 500 (*Internal Server Error*). Pokud je dotaz úspěšně dokončen, obdrží klient odpověď se stavovým kódem 200 (*OK*) nebo 201 (*Created*). Pokud dojde k jakékoliv chybě, je tato skutečnost zalogována, aby ji bylo možné později opravit. Důležité je zmínit, že Mongoose Model provádí jednotlivé dotazy asynchronně. Pokud tedy v rámci jedné metody realizujeme dva dotazy, na jejichž základě provádíme výpočet, musíme zajistit, aby byl výpočet zahájen až po získání dat z obou dotazů. K tomu lze definovat asynchronní funkci a při jejím zavolání použít klíčové slovo `await`. Druhou možností je napsat příslušný kód do části funkce, která odpovídá úspěšnému získání dat.

U některých metod není proveden pouze samotný dotaz, ale jsou implementovány i nějaké další funkcionality. Za zmínku stojí nepochybně metoda pro vkládání knih. Ta očekává, že tělo HTTP požadavku bude obsahovat data z XML souboru se sbírkou. Klient tedy musí nejprve získat textový obsah souboru a ten následně zaslat v těle metody POST na URI `/books`. Metoda před samotným dotazem provede konverzi XML formátu do JSON formátu. Během této konverze jsou nejprve převedeny Unicode kódy znaků na klasické symboly. Následně je zkontrolováno, zda má XML formát správnou syntaxi. Při převodu do JSON formátu je kladen důraz na zachování primitivních datových typů. Z XML obsahu jsou však odstraněny XML komentáře, deklarace a instrukce. Pro XML atributy je vytvořena datová položka `"_attributes"`, pro obsah je vytvořena položka `_text`. Takto převedený XML obsah je připravený pro nahrání do databáze. Kromě XML souborů je také možné nahrát rovnou soubory ve formátu JSON. JSON dokument je do databáze vložen pouze v případě, že je validní vůči schématu v databázi. V opačném případě je klientovi vrácena odpověď se stavovým kódem 400. Metoda počítá s tím, že vkládaná data neobsahují položku `_id`. Pokud tato datová položka v souboru

je a v databázi již existuje dokument s touto datovou položkou stejné hodnoty, pak klient také dostane odpověď se stavovým kódem 400.

Některé implementované metody provádí více dotazů najednou. Například HTTP metody POST, PUT a DELETE na zdroj `/books` pracují také s textem v kolekci `bookTexts`. Při vytváření nové knihy je vytvořen a uložen také text této knihy. Při úpravě či mazání se tyto změny také dotknou i samotného textu knihy. Velmi podobné je to u metody DELETE na zdroj `/users`. Uživatel si vytvoří své vlastní seznamy knih, se kterými pracuje. Pokud dojde k vymazání uživatele z databáze, pak jsou společně s ním vymazány také jeho vytvořené seznamy knih.

Velmi užitečná je také metoda GET na zdroj `mongoeye`. Ta provádí analýzu sbírek básní v kolekci `books`. Součástí HTTP odpovědi je objekt, který se skládá ze 4 částí. První částí je `MongoEyeSchema` obsahující podrobnou analýzu datových položek v jednotlivých sbírkách. U každé datové položky jsou uvedeny vyskytující se datové typy a počty výskytů. Druhou část objektu představuje datová položka `ObjectPropertiesAnalysis`, která vychází z předchozí datové položky a pouze zestručňuje její obsah. Na jejím základě lze velmi efektivně generovat JSON Schema knih. V JSON Schema je dobré dát pozor na výskyt rekurze. K tomu slouží položka `RecursionsAnalysis` obsahující detekci rekurzí u jednotlivých datových položek. Poslední část tvoří datová položka `CommonBookSchema`. Ta obsahuje vygenerované společné JSON Schema pro knihy v kolekci `books`. Toto schéma využívá frontend pro úpravu knih v nástroji `react-jsonschema-form`. Na základě JSON Schema je nástroj schopný vygenerovat formulář pro vytváření či úpravu JSON souborů.

Jak je uvedeno v návrhu, REST API nabízí speciální metody pro práci se seznamy knih. Kromě generování abecedního či frekvenčního slovníku, spojování textů všech knih a vyhledávání frází v textech nabízí rozhraní také generování statistik. Tyto statistiky obsahují počty a délky básní, strof, veršů a slov. U délek jsou uvedeny minimální, průměrné a maximální hodnoty. Dále zde nalezneme počty 1 až N písmenných slov včetně jejich příkladů, které se v knihách vyskytují.

Pro některé metody byly implementovány podpůrné funkce usnadňující práci s daty. Tyto metody lze najít ve zdrojových kódech ve složkách `analysis`, `poem` a `utilities`. Dále zdrojové kódy obsahují složku `config` pro konfiguraci, `links` pro generování linků v HTTP odpovědích a `winston` pro nastavení logování.

4.2.2 Získání textů knih

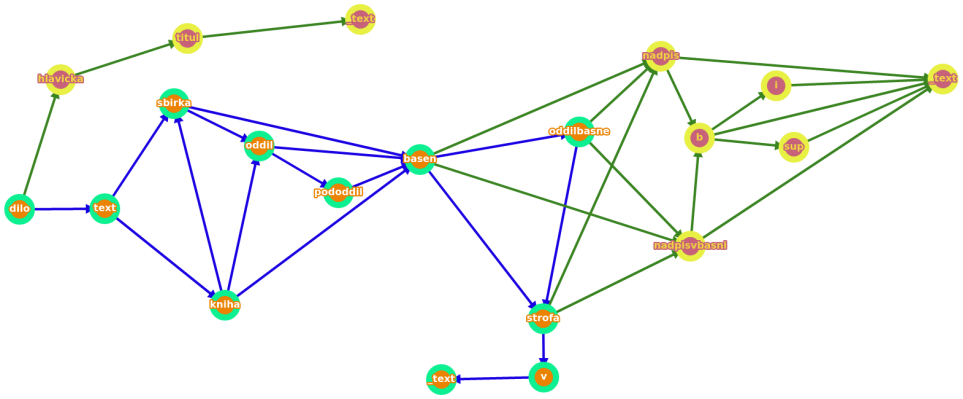
Jak bylo zmíněno v předchozí sekci, při vkládání a úpravách knih je z knih extrahován text, který je uložen do samostatné kolekce s texty knih. Tento text je zobrazován ve frontendu v uživatelském rozhraní v případě, že si jej chce uživatel přečíst. Také jsou nad ním vytvořeny indexy pro efektivnější vyhledávání a vytváření slovníků.

Získání samotného textu však není úplně triviální záležitost vzhledem k odlišným strukturám jednotlivých sbírek. Pro zjednodušení je uvažováno, že text obsahuje nadpis s názvem celého díla. Následují jednotlivé básně ve sbírce obasující nadpis básně a jednotlivé sloky, které se skládají z veršů. Získání názvu díla je velmi snadné, jelikož každá kniha obsahuje hlavičkové údaje, ve kterých se povinně nachází datová položka `titul`. Cesta k textům jednotlivých básní už tak jednoznačná není.

A právě pro tyto účely byla využita analýza pomocí nástroje mongoeye, která je obsažena i ve výstupu metody GET na zdroj `/mognoeye`. Součástí tohoto výstupu je datová položka `ObjectPropertiesAnalysis`, díky které lze se sbírkami pracovat jako s grafem. Sbírky mají stromovou strukturu, kdy kořenem tohoto stromu je datová položka `dilo`. V `ObjectPropertiesAnalysis` lze najít mimo jiné hrany mezi jednotlivými uzly v grafu.

Pro získání textu knih je tedy nejprve potřeba nalézt všechny cesty z kořene stromu (položka `dilo`) do uzlů s názvem díla (položka `titul`), verši (položka `v`) a nadpisy básní slok či oddílů básní (položky `nadpisvbasni` nebo `nadpis`). Nalezení všech cest k těmto položkám bylo provedeno pomocí iterativního prohledávání do hloubky. Na následujícím obrázku je znázorněn výsledek tohoto prohledávání.

Obrázek 4.1: Grafová struktura textu knih



Implementovaný algoritmus pro získání textů z knih vychází z grafu, který je na obrázku uvedeném výše. Algoritmus počítá s tím, že potřebné texty se mohou vyskytnout na libovolné z uvedených cest. Implementaci této funkcionality lze nalézt ve zdrojových kódech v adresáři `poem` v souboru s názvem `bookTextExtractor.js`.

4.2.3 Dokumentace Swagger

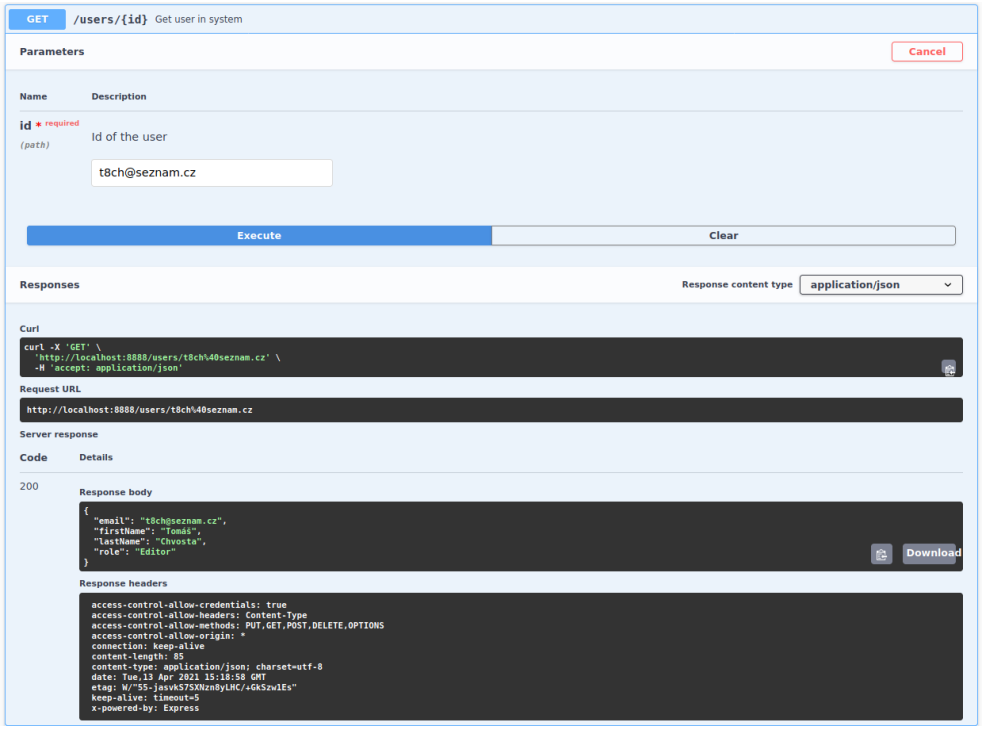
Swagger UI je open-source framework, který slouží pro dokumentaci RESTful web API. Tento nástroj umožňuje vizualizovat a testovat REST API. Pomocí

něj byla vytvořena dokumentace pro implementované rozhraní REST. Zdrojový kód této dokumentace se nachází v souboru `swagger.json`.

Dokumentaci je možné najít na URL `/api-docs`. Zde je uveden seznam a krátký popis všech zdrojů a u každého zdroje jsou popsány všechny dostupné metody. U každé metody je uvedeno URI a krátké vysvětlení k čemu metoda slouží. Po rozkliknutí metody je zobrazen detailní popis metody obsahující veškeré vstupní parametry včetně ukázkových dat, ukázky HTTP odpovědí pro všechny možné stavové kódy a také jsou uvedeny odkazy, které jsou součástí odpovědi.

Kromě zobrazení popisu je také možné metodu vyzkoušet. Uživatelské rozhraní umožňuje zadat všechny potřebné parametry a metodu provést. Také je vygenerován odpovídající příkaz pro nástroj Curl, který je dostupný na příkazové řádce.

Obrázek 4.2: Ukázka nástroje Swagger



4.2.4 Konfigurace aplikačního serveru

Pro základní nastavení parametrů aplikačního serveru slouží ve zdrojových kódech složka `config`, ve které se nachází soubor `config.js`. V tomto souboru lze nastavit všechny parametry aplikace. Jedním z nich je například TCP port na kterém server poslouchá. Výchozí hodnota tohoto portu je 8888. Kromě

toho lze nastavit adresu MongoDB serveru. Nevzniká problém ani v případě, že databáze běží na vzdáleném serveru. Dále je možné nastavit cestu ke složce, do které se mají ukládat všechny výstupní soubory. Aplikace také loguje chyby, které nastaly a veškeré přijaté HTTP požadavky. V souboru `config.js` je možné zvolit, do kterého souboru se budou logy zapisovat.

Výše zmíněné parametry nejsou jediné, které lze v konfiguračním souboru nastavit. Obecně lze v souboru nastavit veškeré konstanty a názvy.

4.2.5 Využité knihovny

Jazyk Javascript obsahuje rozsáhlou sadu základních knihoven, nicméně pro některé implementované funkcionality byly využity externí knihovny. Instalace knihoven byla provedena pomocí NPM. V následujícím seznamu jsou uvedeny všechny knihovny, které byly pro implementaci řešení použity.

body-parser (verze 1.19.0) knihovna usnadňující parsování dat v těle požadavku dostupných v `req.body`

cookie-parser (verze 1.4.5) knihovna umožňující správu cookies (tuto funkcionalitu aplikace zatím nemá, v budoucnu však bude implementována, proto je knihovna připravena k použití)

express (verze 4.17.1) webový framework určený pro vytváření webových aplikací a API

fast-xml-parser (verze 3.18.0) knihovna, pomocí které lze mimo jiné rychle a efektivně validovat XML dokumenty

mongoeye (verze 0.0.1) knihovna umožňující podrobnou analýzu dokumentů v kolekcí databáze MongoDB

mongoose (verze 5.11.14) knihovna umožňující modelování objektů v databázi MongoDB a práci s nimi

morgan (verze 1.10.0) knihovna pro logování příchozích HTTP požadavků

swagger-ui-express (verze 4.1.6) knihovna generující uživatelské rozhraní s dokumentací implementovaného RESTful API a umožňující testování implementovaných metod

winston (verze 3.3.3) knihovna umožňující jednoduché a univerzální logování s podporou využití libovolného datového úložiště

xml-js (verze 1.6.11) knihovna nabízející snadný vzájemný převod formátů XML a JSON a také konfiguraci tohoto převodu

4.3 Nasazení aplikace

Implementovanou aplikaci je možné spustit lokálně nebo ji lze nasadit na vzdálený server. V případě lokálního spuštění je třeba nejprve spustit deamon proces `mongod` a jako jeho parametr uvést cestu k vytvořené databázi. Příkaz je uveden níže.

```
./mongod --dbpath <dbpath>.
```

Tento proces poslouchá na TCP portu 27017. Pokud je potřeba změnit tento port, je možné použít příznak `--port`. Nyní je třeba spustit samotný aplikační server. Před spuštěním je nutné nainstalovat všechny potřebné závislosti. Ve složce `BookApp` se nachází soubor `package.json`, ve kterém jsou uvedeny všechny závislosti. Instalaci závislostí lze provést pomocí následujícího příkazu.

```
npm install
```

Tento příkaz nainstaluje všechny knihovny, které jsou uvedeny v souboru `package.json`. Všechny přidané knihovny se po úspěšném provedení příkazu nacházejí ve složce `node_modules`. Nyní už zbývá pouze nastavit všechny parametry v souboru `src/config/config.js`, především cestu složky pro ukládání výstupů a cestu k souboru pro zapisování logů. Poté už nic nebrání spuštění aplikace pomocí příkazu uvedeného níže. Zda aplikace běží správně lze zjistit ve webovém prohlížeči zadáním URL `http://localhost:<port>/api-docs`, kde `<port>` je příslušný TCP port uvedený v souboru `config.js`. Na této URL by měla být dostupná dokumentace poskytovaného REST API.

```
npm start
```

V případě nasazení aplikace na vzdálený server je možné využít stejný postup. Pokud však dojde k restartu vzdáleného serveru, je třeba znovu spustit databázi i aplikaci. Stejně tak v případě pádu databáze či aplikace v důsledku nějaké neočekávané chyby. Proto je třeba zajistit, aby systém neustále hlídal, zda jsou oba procesy spuštěny. V operačním systému Linux je možné vytvořit skript (například `mongo.sh`), který obsahuje příkaz na spuštění deamon procesu `mongod`. Dále je potřeba vytvořit tzv. *Service Unit File* (například `mongo.service`), jehož obsah může vypadat následovně:

```
[Unit]
Description=MongoDB server

Wants=network.target
After=syslog.target network-online.target
```

```
[Service]
Type=simple
ExecStart=/usr/local/bin/mongo.sh
Restart=on-failure
RestartSec=10
KillMode=process

[Install]
WantedBy=multi-user.target
```

Název souboru musí obsahovat příponu `.services`. V tomto souboru je mimo jiné specifikováno, který skript má být spuštěn jako služba a kdy má dojít k restartu služby. Konkrétně v této ukázce je nastaveno, že má dojít k restartu v případě nějaké chyby, což je kontrolováno každých 10 sekund. Tento soubor je třeba přesunout do složky `/etc/systemd/system/` a nastavit mu příslušná oprávnění. Poté už stačí pouze spustit službu pomocí příkazů uvedených níže.

```
sudo systemctl daemon-reload
sudo systemctl enable mongo
sudo systemctl start mongo
```

U aplikace je možné udělat totéž. Pro tyto účely však existuje tzv. *Process manager*, který lze nainstalovat pomocí NPM. Tento nástroj se postará o běh aplikace a v případě chyby ji restartuje. Aplikaci lze spustit pomocí níže uvedeného příkazu.

```
pm2 start server.js
```

Process manager však neřeší restart systému, proto je třeba provést podobnou proceduru jako u databáze MongoDB. To však nemusíme provádět manuálně a můžeme použít příkazy uvedené níže, které automaticky vytvoří a uloží skript i Service Unit File.

```
pm2 startup systemd
pm2 save
```

Následně stačí pouze spustit službu pomocí `systemctl start`. Tím máme zajištěno, že dojde ke spuštění Process manageru vždy po spuštění systému nebo po pádu Process manažeru, což je však nepravděpodobné. Pro kontrolu, zda služby běží, lze použít příkaz `systemctl status <serviceName>`. Pro kontrolu samotné aplikace slouží příkaz `pm2 info <appName>` případně `pm2 monit`. V případě, že je třeba zastavit službu nebo aplikaci, je možné použít příkazy `systemctl stop <serviceName>` nebo `pm2 stop <appName>`. Poslední příkaz, který se může hodit, je `pm2 restart <appName>` pro manuální restart běžící aplikace.

4.4 Rozšiřitelnost řešení

Současné řešení obsahuje nové datové úložiště a aplikační server, který zpřístupňuje REST API pro jednotlivé klienty. Datové úložiště se skládá z kolekcí, ve kterých jsou uloženy jednotlivé knihy, texty knih, verze knih, dále uživatelé a jejich seznamy. REST API umožňuje k těmto datům přistupovat a pracovat s nimi. Pro knihy, uživatele a jejich seznamy jsou k dispozici všechny CRUD operace. Verze a texty knih umožňuje API pouze číst, o vytváření a modifikaci dat se stará aplikační server automaticky. Pro potřeby frontendu jsou navíc k dispozici metody pro získání statistik, abecedních a frekvenčních slovníků k daným seznamům a také pro vyhledávání či spojení textů knih v seznamu. Dále je implementována metoda umožňující provádět analýzu děl uložených v kolekci knih. Kromě REST API je také součástí řešení uživatelské rozhraní, pomocí kterého lze implementované REST API testovat.

Implementované řešení počítá s tím, že do něj budou přidány další funkcionality. V případě potřeby rozšíření datového úložiště lze přidat další databázi či kolekci do existující databáze. Do dokumentů kolekce lze přidat libovolné datové položky, aniž by to způsobilo nějaké problémy. Je však nutné zachovat všechny povinné datové položky tak, aby dokumenty splňovaly definované validační schéma. Pokud v budoucnu nastane nějaká změna v těchto položkách, musí se změna promítnout i ve schématu. Zároveň je třeba dát pozor, aby změna nijak neovlivnila chování současných metod v REST API. V kolekcích je také možné vytvářet nové indexy nad danými datovými položkami. Při změně existujících indexů je opět nutné prověřit, zda se tato změna negativně neprojevila na existujících funkcionalitách v REST API.

Do REST API je možné přidat libovolné metody navíc. Jednotlivé metody jsou na sobě nezávislé, nová metoda tedy nebude mít na chování současných metod žádný vliv. Počítá se například s tím, že v REST API vzniknou nové metody na práci s verzemi knih. Pokud dojde ke změně v současných metodách, je třeba zajistit, aby změna neovlivnila implementaci klientů, kteří danou metodu využívají.

Kapitola 5

Testování a vyhodnocení

Součástí této práce je důkladné otestování implementovaného řešení. Tato kapitola popisuje proces testování jednotlivých funkcionalit řešení a také zhodnocení tohoto procesu. Testování lze dělit dle několika kritérií. V následujících sekcích jsou popsány jednotkové testy, dále testy pro samotné REST API, performance testy a krátce také akceptační testy.

5.1 Testování jednotek

Jednotkové testy (Unit testy) přicházejí na řadu jako první. Jejich cílem je otestovat pouze jednu danou konkrétní jednotku. V ideálním případě by měl být každý testovaný případ nezávislý na ostatních. Tyto testy by měly být automatizované a měly by ověřit opakující se scénáře za pomoci krátkých programů.

Tyto testy neslouží pro otestování samotného REST API. Jsou vytvořeny pro každou implementovanou podpůrnou funkci, kterou REST API využívá pro zpracování dat a výpočty nad těmito daty. Ve složce se zdrojovými kódy `src` se nacházejí dva adresáře `main` a `test`. Struktura adresáře `main` byla popsán v předchozí kapitole (viz 4.2) a obsahuje celé implementované řešení. Soubory v této části mají příponu `.js`. Struktura adresáře `test` je téměř shodná. Složky a soubory v tomto adresáři mají naprosto shodné názvy jako v adresáři `main`. Jediné, v čem se liší, jsou přípony souborů, zde mají příponu `.ts`. Pro každý soubor s příponou `.js` je tedy vytvořen odpovídající soubor s příponou `.ts`, který obsahuje jednotkové testy pro funkce a metody implementované v souborech s příponou `.js`. Pomocí těchto testů nejsou testovány REST metody a metody přistupující k databázi.

Pro jednotkové testy je použit testovací framework Mocha, který běží na NodeJS. Tento nástroj sériově spouští jednotlivé testy a vytváří reporty. Lze ho jednoduše nainstalovat pomocí NPM. V souboru `package.json` byl přidán kód zobrazený v ukázce níže, který zajistí, že nástroj Mocha rekurzivně pro-

jde adresář `src/test` a jeho podadresáře a provede všechny testy obsažené v souborech s příponou `.ts`.

```
"scripts": {
    "test": "mocha src/test/**/*.ts"
}
```

Každý z testů slouží k otestování právě jedné metody. Jeden test může obsahovat klidně více scénářů. Příklad testu pro metodu `getTextFromBook` z `bookTextExtractor.js` je zobrazen v následující ukázce. Tento test zkusí získat text knihy z testovacího objektu a kontroluje, zda je výsledek shodný s očekávanou hodnotou.

```
var fName = 'Poem (bookTextExtractor): getTextFromBook'
describe(fName , function(){
    it('works' , function(){
        assert.strictEqual(
            bookTextExtractor( testObjectWithBook ),
            expectedText
        );
    });
});
```

Výstupem testu je report obsahující informaci o tom, zda testovací scénář `works` proběhl bez chyby či nikoliv. Pokud nastane chyba, obsahuje report také podrobný popis příčiny neúspěchu testu.

Všechny testy je možné najednou spustit pomocí příkazu `npm test`. V případě potřeby je možné nastavit, aby se testy spouštěly automaticky. Díky tomu je neustále kontrolováno, zda některé změny v implementaci nezpůsobí chyby v existujících funkcích a metodách. Celkem bylo pro implementované metody vytvořeno zhruba 30 jednotkových testů. Počítá se s tím, že při implementaci nových funkcí vzniknou zároveň pro tyto funkce odpovídající jednotkové testy.

5.2 Manuální testování

V předchozí sekci jsou popsány testy, které jsou spouštěny automaticky a kontrolují jednotlivé funkce a metody, které jsou implementovány. Tyto testy by měly sloužit pouze k testům jednotek, které by na sobě měly být nezávislé. Pro testování REST API však chceme využít jiný druh testů. Jednotlivé metody v REST rozhraní jsou sice samy o sobě také nezávislé, avšak v praxi jsou velmi často používány v nějakém kontextu a jsou tedy závislé i na jiných metodách. Například pokud se v implementované aplikaci rozhodne uživatel přidat knihu do nějakého seznamu, pak je nutnou podmínkou pro provedení této

operace také vytvoření příslušného seznamu knih případně vytvoření uživatelského účtu. Z těchto důvodů je vhodnější vytvořit testovací scénaře, pomocí kterých bude možné implementované REST API manuálně otestovat.

Tyto scénáře byly vytvořeny pro všech 5 základních zdrojů. Konkrétně se tedy jedná o scénáře, které ověří funkčnost metod pro práci s knihami, uživateli a jejich seznamy knih. Také je třeba ověřit, že dochází ke správnému vytváření textů a starších verzí knih. Společně se scénáři je třeba vytvořit i testovací data, na kterých bude řešení otestováno. Pro samotné testy je použit framework Swagger-UI zmíněný v jedné z předchozích sekcí (viz 4.2.3). Ten umožňuje velmi elegantně zadat všechny potřebné parametry pro každou metodu, kterou lze poté spustit přímo z webového prohlížeče.

Nejprve bylo testováno API pro práci s knihami. Testování probíhalo podle následujícího scénáře.

Import knihy

Test 1: Pokus o import knihy, která není ve formátu XML. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 2: Pokus o import knihy, která je ve formátu XML, ale neobsahuje autora. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 3: Pokus o import knihy, která má všechny náležitosti. (Očekává se, že bude kniha úspěšně nahrána.)

Zobrazení vytvořené knihy

Test 4: Pokus o získání neexistující knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 5: Pokus o získání nově vytvořené knihy. (Očekává se, že odpověď bude obsahovat data knihy.)

Zobrazení seznamu všech knih

Test 6: Pokus o získání seznamu všech knih. (Očekává se, že odpověď bude obsahovat seznam knih.)

Test 7: Pokus o získání seznamu knih v reprezentaci pro knihovnu frontendu. (Očekává se, že odpověď bude obsahovat seznam knih.)

Úprava knihy

Test 8: Pokus o úpravu neexistující knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 9: Pokus o nevalidní úpravu knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 10: Pokus o validní úpravu knihy. (Očekává se, že kniha bude upravena.)

Test 11: Kontrola upravené knihy. (Očekává se, že odpověď bude obsahovat upravená data.)

Vymazání knihy

Test 12: Pokus o vymazání neexistující knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 13: Pokus o vymazání existující knihy. (Očekává se, že kniha bude vymazána.)

Test 14: Kontrola vymazané knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Při testování API pro knihy nebyl zjištěný žádný problém v implementovaném řešení. Drobný problém se vyskytl u nástroje Swagger-UI. V případě, že odpověď obsahuje větší množství dat, pak trvá velmi dlouho zobrazení výstupu. Následující testovací scénář slouží k otestování API pro práci s uživateli. Do tohoto scénáře je přidána i metoda pro autentizaci uživatele.

Vytvoření uživatele

Test 1: Pokus o vytvoření uživatele s chybějícím křestním jménem. (Očekává se, že odpověď bude obsahovat chybovou hlášku a uživatel nebude vytvořen.)

Test 2: Pokus o vytvoření uživatele, u něhož je email zadán ve špatném formátu. (Očekává se, že odpověď bude obsahovat chybovou hlášku a uživatel nebude vytvořen.)

Test 3: Pokus o vytvoření uživatele, u kterého jsou všechny údaje zadány správně. (Očekává se, že uživatel bude vytvořen.)

Test 4: Pokus o vytvoření stejného uživatele, jako v předchozím kroku. (Očekává se, že odpověď bude obsahovat chybovou hlášku a uživatel nebude znovu vytvořen.)

Zobrazení vytvořené knihy

Test 5: Pokus o získání dat uživatele s chybně zadaným emailem. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 6: Pokus o získání dat uživatele vytvořeného v testu 3. (Očekává se, že odpověď bude obsahovat data uživatele.)

Zobrazení seznamu všech uživatelů

Test 7: Pokus o získání dat všech uživatelů. (Očekává se, že odpověď bude obsahovat data všech uživatelů.)

Autentizace uživatele

Test 8: Pokus o autentizaci uživatele s nesprávně zadaný heslem. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 9: Pokus o autentizaci uživatele s nesprávně zadaným emailem. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 10: Pokus o autentizace uživatele se správnými přihlašovacími údaji. (Očekává se, že autentizace proběhne bez chyby.)

Úprava uživatele

Test 11: Pokus o úpravu dat neexistujícího uživatele. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 12: Pokus o úpravu emailu uživatele. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 13: Pokus o úpravu jména uživatele. (Očekává se, že uživatelská data budou upravena.)

Test 14: Kontrola upraveného uživatele. (Očekává se, že odpověď bude obsahovat nová data uživatele.)

Vymazání uživatele

Test 15: Pokus o vymazání neexistujícího uživatele. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 16: Pokus o vymazání existujícího uživatele. (Očekává se, že uživatel bude odstraněn.)

Test 17: Kontrola smazaného uživatele. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Během testování API pro práci s uživateli byla nalezena chyba v testu číslo 12. Tato chyba však nemá vliv na fungování přidružených klientů. Při pokusu upravit email uživatele byla očekávána odpověď s chybovou hláškou obsahující informaci, že email nelze upravit. Odpověď sice obsahovala chybovou hlášku, v té však byla informaci, že upravovaný uživatel neexistuje. Při bližším přezkoumání chyby bylo zjištěno, že systém MongoDB nezareagoval chybou, ale informací, že nebyl upraven žádný dokument. Aplikace proto mylně usoudila, že upravovaný uživatel nebyl nalezen. Tato chyba byla opravena a již se v implementovaném řešení nenachází.

Následující scénář se týká samotných textů knih. Zde API obsahuje pouze metody pro čtení, nikoliv pro zápis. Operace zápisu jsou prováděny v rámci metod pro knihy, díky čemuž je zajištěna konzistence dat v databázi. Scénář vypadá následovně.

Vytvoření nového textu knihy

Test 1: Pokus o vložení nové knihy. (Očekává se, že bude zároveň vytvořen i text knihy.)

Test 2: Kontrola textu knihy. (Očekává se, že odpověď bude obsahovat text knihy.)

Úprava textu knihy

Test 3: Pokus o úpravu knihy. (Očekává se, že bude zároveň upraven i text knihy.)

Test 4: Kontrola textu upravené knihy. (Očekává se, že odpověď bude obsahovat upravený text knihy.)

Vymazání textu knihy

Test 5: Pokus o vymazání knihy. (Očekává se, že bude zároveň vymazán i text knihy.)

Test 6: Kontrola textu vymazané knihy. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Při testování API pro texty knihy nebyla nalezena žádná chyba v implementovaném řešení. Další testování se týká verzí knih. Ty jsou vytvářeny vždy při úpravě či mazání knihy. Testovací scénář pro verze knih je zobrazen níže.

Vytvoření verze knihy

Test 1: Vytvoření nové knihy.

Test 2: Pokus o získání staré verze. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 3: Pokus o úpravu knihy. (Očekává se, že bude zároveň vytvořena stará verze knihy.)

Test 4: Kontrola staré verze knihy. (Očekává se, že odpověď bude obsahovat starou verzi knihy.)

Test 5: Pokus o vymazání knihy. (Očekává se, že bude zároveň vytvořena stará verze knihy.)

Test 6: Kontrola staré verze knihy. (Očekává se, že odpověď bude obsahovat starou verzi knihy.)

Při tomto testování byl objeven drobný nedostatek. Čas vytvoření jednotlivých verzí je o 2 hodiny opožděn. Tento problém je způsoben časovým posunem v Javascriptu u objektu typu `Date`. Ten vrací o 2 hodiny pozdější čas. Tento nedostatek byl z řešení odstraněn.

Poslední testovací scénář, který zbývá zmínit, se týká seznamů knih. Kromě klasické práce se seznamy knih je také třeba ověřit, zda při odstranění uživatele mizí z databáze také všechny jeho seznamy knih. Scénář vypadá následovně.

Vytvoření seznamu knih

Test 1: Vytvoření uživatele.

Test 2: Pokus o vytvoření seznamu neexistujícímu uživateli. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 3: Pokus o vytvoření nového seznamu. (Očekává se, že bude seznam úspěšně vytvořen.)

Zobrazení prázdného seznamu

Test 4: Pokus o získání seznamu, ve kterém nejsou žádné knihy. (Očekává se, že odpověď bude obsahovat seznam bez knih.

Úprava seznamu

Test 5: Pokus o nevalidní úpravu seznamu knih. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 6: Pokus o přidání knih do seznamu. (Očekává se, že seznam knih bude upraven.)

Zobrazení konkrétního seznamu

Test 7: Pokus o zobrazení neexistujícího seznamu. (Očekává se, že odpověď nebude obsahovat žádný seznam.)

Test 8: Pokus o získání upraveného seznamu knih. (Očekává se, že odpověď bude obsahovat data se seznamem.)

Zobrazení všech seznamů

Test 9: Pokus o získání seznamu všech seznamů uživatele. (Očekává se, že odpověď bude obsahovat data se seznamy.)

Práce se seznamy

Test 10: Pokus o zobrazení abecedního slovníku. (Očekává se, že odpověď bude obsahovat abecední slovník pro dané knihy.)

Test 11: Pokus o zobrazení frekvenčního slovníku. (Očekává se, že odpověď bude obsahovat frekvenční slovník pro dané knihy.)

Test 12: Pokus o zobrazení statistik. (Očekává se, že odpověď bude obsahovat statistiky pro dané knihy.)

Test 13: Pokus o fulltextové vyhledávání jednoho znaku. (Očekává se, že odpověď bude obsahovat výskyty hledané fráze.)

Test 14: Pokus o fulltextové vyhledávání fráze. (Očekává se, že odpověď bude obsahovat výskyty hledané fráze.)

Test 15: Pokus o fulltextové vyhledávání nevyskytující se fráze. (Očekává se, že odpověď bude obsahovat výskyty hledané fráze.)

Test 16: Pokus o zobrazení všech textů knih. (Očekává se, že odpověď bude obsahovat texty daných knih.)

Vymazání seznamu

Test 17: Pokus o vymazání neexistujícího seznamu. (Očekává se, že odpověď bude obsahovat chybovou hlášku.)

Test 18: Pokus o vymazání existujícího seznamu. (Očekává se, že seznam knih bude vymazán.)

Test 19: Kontrola seznamů uživatele. (Očekává se, že odpověď nebude obsahovat smazaný seznam.)

Test 20: Vytvoření nového seznamu.

Test 21: Kontrola seznamů uživatele. (Očekává se, že odpověď bude obsahovat nový seznam.)

Test 22: Vymazání uživatele.

Test 23: Kontrola vytvořeného seznamu. (Očekává se, že odpověď nebude obsahovat žádný seznam.)

Poslední testování neodhalilo žádnou podstatnější chybu s výjimkou nejednotnosti struktury výstupních dat. Tento nedostatek byl také opraven. Všech 66 testů lze nalézt na přiloženém DVD disku ve složce `RESTapiTesting`. Zde se nachází složky s jednotlivými testovacími scénáři. V těchto složkách se kromě testovacích scénářů nachází také složka `data` obsahující sadu testovacích dat pro jednotlivé testy, složka `tests` se screenshoty prováděných testů a také složka `results`, která obsahuje screenshoty s výsledky testů. Některé

testy nedovolil Swagger provést. Jednalo se pouze o jeden test, při kterém bylo testováno, zda aplikace zareaguje na nevalidní data chybovou hláškou. Chybná data však byla rozpoznána nástrojem Swagger, který kvůli tomu test neprovedl. V takových případech byl pro provedení testu použit nástroj Curl.

5.3 Výkonnostní testování

O testování jednotlivých funkcionalit se starají testy v předchozích dvou sekcích. Je však potřeba otestovat řešení z pohledu výkonu. V tomto testování se zaměříme na operace, které trvají nejdéle. Jedná se především o hromadné operace s knihami, a tedy i seznamy knih.

Předpoklad je takový, že nejnáročnější operace z pohledu času bude import všech sbírek do databáze. Těchto sbírek je zatím 1700, ale je možné, že v budoucnu ÚČL zpracuje další sbírky. Při vkládání jsou sbírky nejprve převedeny z XML do JSON formátu, následně je vytvořen samotný text, který je společně s knihou nahrán do databáze. Vložení do databáze může být navíc zpomaleno vytvořenými indexy. Mezi další náročnější operace s knihami patří mazání knih z databáze. Samotné vymazání knihy je velmi rychlé, avšak společně s vymazáním je také potřeba uložit mazanou verzi knihy a vymazat text knihy, nad kterým jsou vytvořeny indexy. Poslední hromadnou operací nad knihami je získání seznamu všech knih. Vzhledem k velkému objemu dat může být tato operace také časově náročná. Zde by však nemělo dojít k žádnému zpomalení kvůli vytvořeným indexům. Pomocí testování je třeba zjistit, jak dlouho tyto operace trvají a jak velký vliv má na dobu běhu vytvoření indexů.

Sada testovacích dat je tedy tvořena 1700 digitalizovanými sbírkami ÚČL. Z těch jsou náhodně vybrány sbírky, ze kterých jsou vytvořeny testovací sady obsahující 100, 250, 500, 1000 a 1700 sbírek. Testování proběhlo na stroji s operačním systémem Debian 10, dvoujádrovým procesorem Intel(R) Core(TM) i7-6500U CPU o frekvenci 2,5 GHz a RAM pamětí o velikosti 8 GB.

Naměřené hodnoty slouží pro srovnání rychlosti růstu času dle počtu sbírek a použití indexace databáze. Hodnoty uvedené v tabulkách nepředstavují výsledek jednoho konkrétního měření. Pro každou operaci a příslušnou testovací sadu bylo měření provedeno celkem 5krát. Z těchto měření byl následně vypočítán aritmetický průměr. Alternativně bylo možné provést měření více a použít medián z naměřených hodnot. Tato procedura slouží k eliminaci anomálií, které mohou vzniknout během měření.

Pro samotné měření času nebyl použit žádný speciální nástroj nýbrž logovací nástroj Winston. Při zahájení dané operace byl zalogován přesný čas s přesností na milisekundy. Stejně tak byl zalogován čas ukončení dané operace, který představuje okamžik odeslání výsledných dat pomocí HTTP odpovědi. Rozdíl těchto dvou časů představuje dobu běhu měřené operace. Následující tabulky obsahují naměřenou dobu běhu operací u jednotlivých testovacích sad dat.

Tabulka 5.1: Doba běhu operací pro knihy (bez indexace)

	100	**250**	**500**	**1000**	**1700**
Vkládání knih	5,418 s	15,013 s	30,804 s	62,876 s	98,779 s
Mazání knih	2,287 s	6,378 s	11,541 s	23,639 s	39,146 s
Získání všech knih	1,301 s	3,902 s	6,413 s	14,541 s	26,146 s
Knihy pro knihovnu	0,014 s	0,060 s	0,064 s	0,120 s	0,154 s

Tabulka 5.2: Doba běhu operací pro knihy (s indexací)

	100	**250**	**500**	**1000**	**1700**
Vkládání knih	5,551 s	15,729 s	31,301 s	71,899 s	139,643 s
Mazání knih	2,377 s	6,952 s	14,666 s	30,347 s	57,642 s
Získání všech knih	1,369 s	3,805 s	6,422 s	14,953 s	28,168 s
Knihy pro knihovnu	0,013 s	0,058 s	0,062 s	0,132 s	0,150 s

Testování potvrdilo hypotézu, že nejnáročnější operací je import knih. Nahrání všech 1700 knih do databáze trvalo téměř 140 vteřin. Vzhledem ke komplexitě operace vkládání jednotlivých knih je tato doba přijatelná. Druhou nejnáročnější operací z hlediska času je operace mazání knih. Vymazání 1700 knih zabralo téměř minutu. Na dobu běhu těchto dvou operací mají také vliv indexy. Rozdíl je však patrný až u testovací sady s 1000 sbírkami. U testovací sady se 1700 sbírkami je doba běhu při použití databáze s vytvořenými indexy zhruba o 40% vyšší než při použití databáze bez vytvořených indexů. Dá se předpokládat, že u větších testovacích sad by byl rozdíl markantnější. Jelikož však celkový počet digitalizovaných sbírek bude v budoucnu v řádech tisíců a operace vkládání a mazání sbírek se budou provádět minimálně, je tato doba běhu operací při použití databáze s vytvořenými indexy přijatelná.

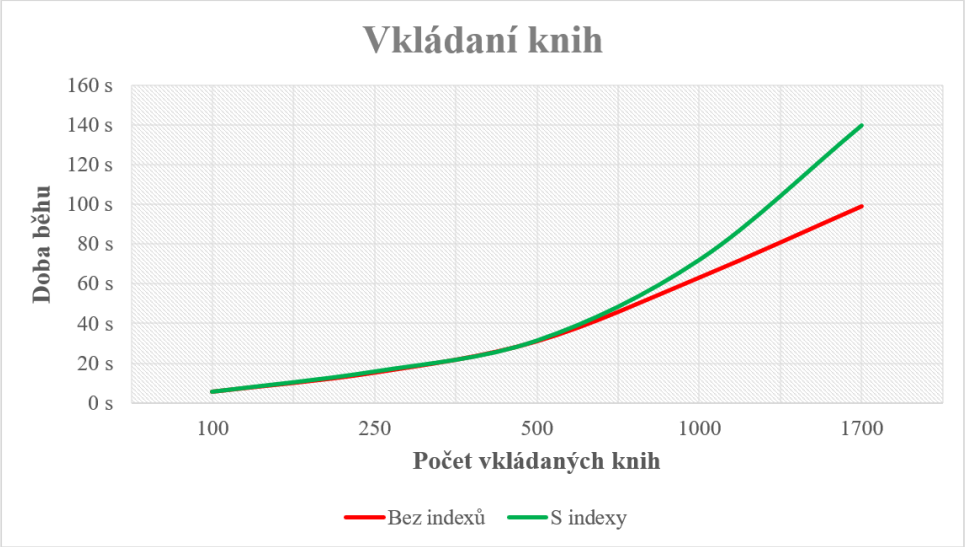
Další měření bylo prováděno na operaci získání seznamu všech dostupných děl v databázi. U této operace se předpokládá, že bude využívána častěji. Pro testovací sadu obsahující 1700 sbírek trvá získání všech dat zhruba 28 vteřin. Tato doba by pro uživatele nemusela být přijatelná, navíc většina prohlížečů není schopna takto velký objem dat zobrazit. Frontend aplikace tuto operaci nevyužívá. Místo toho používá metodu, která sice vrátí seznam všech knih, ty však obsahují pouze vybrané atributy. Tento seznam je následně zobrazen v uživatelském rozhraní v sekci „knihovna“. Proto v tabulkách výše přibyla ještě jedna operace pro tuto metodu. Z naměřených hodnot je vidět, že získání dat s podmnožinou atributů je velmi rychlé a doba běhu je pro frontend aplikace přijatelná.

Testování objevilo nedostatek v souvislosti s metodou pro získání všech knih. Co se týče doby běhu je implementace této metody nevyhovující. Z tohoto důvodu byla metoda změněna a nyní nevrací všechna data, ale pouze podmnožinu atributů knih, konkrétně autora, titul a rok vydání. Ke všem da-

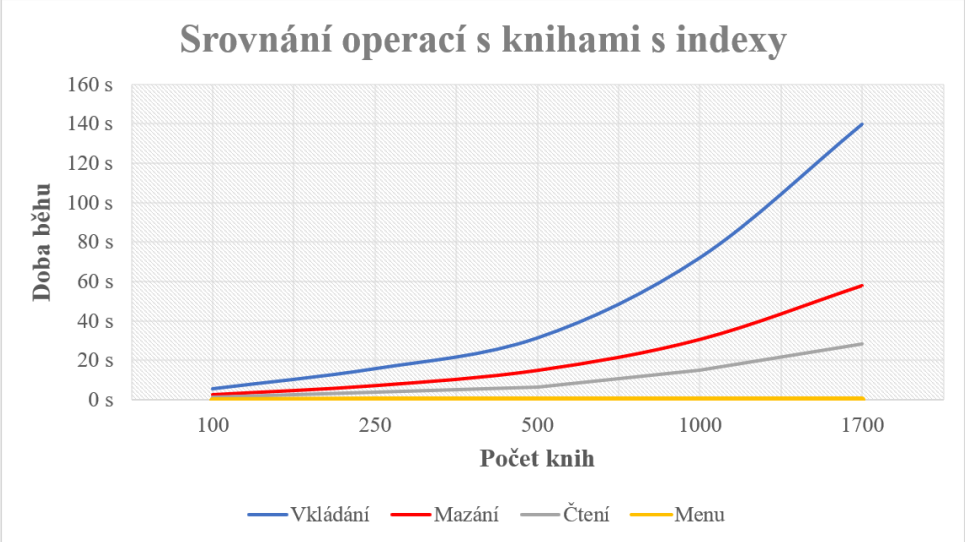
tům konkrétní knihy lze poté přistoupit pomocí HATEOAS linků, které jsou součástí HTTP odpovědi a jsou dostupné u každé knihy v seznamu.

V následujícím grafu jsou znázorněny naměřené hodnoty pro operaci vkládání knih. V dalším grafu lze nalézt srovnání všech operací při použití indexů. Další grafy lze nalézt v příloze C.

Obrázek 5.1: Doba běhu vkládání knih



Obrázek 5.2: Srovnání doby běhu operací s knihami



Kromě operací pro práci s knihami byly testovány také operace se seznamy knih. Jedná se především o vytváření abecedního a frekvenčního slovníku,

fulltextové vyhledávání a spojení všech textů knih v seznamu. U těchto operací se nepředpokládá, že by mělo vytvoření indexů významný vliv na dobu běhu. Texty knih jsou však velmi často rozděleny na základě nějaké fráze a je možné že databázový systém MongoDB využije textové indexy pro toto rozdělení textu. To je však hypotéza, kterou bude potřeba ověřit pomocí testování. Dále nás bude zajímat doba běhu jednotlivých operací.

V případě fulltextového vyhledávání je testováno více případů. První případ (1 nebo také FtA) představuje vyhledávání symbolu „a“, u kterého se předpokládá zvýšený počet výskytů. V druhém případě (2 nebo také FtS) je vyhledáváno slovo „srdce“, které se v textech vyskytuje často. Ve třetím případě (3 nebo také FtSSP) je vyhledávána fráze „srdce smutek provázel“, která se nachází pouze v jedné knize s názvem Písně tulákovy od Jiřího Karáska ze Lvovic. Poslední případ (4 nebo také FtIT) představuje vyhledávání fráze „informační technologie“, která se v textech nikde nevyskytuje.

V následujících tabulkách a grafech jsou zobrazeny naměřené hodnoty doby běhu jednotlivých operací:

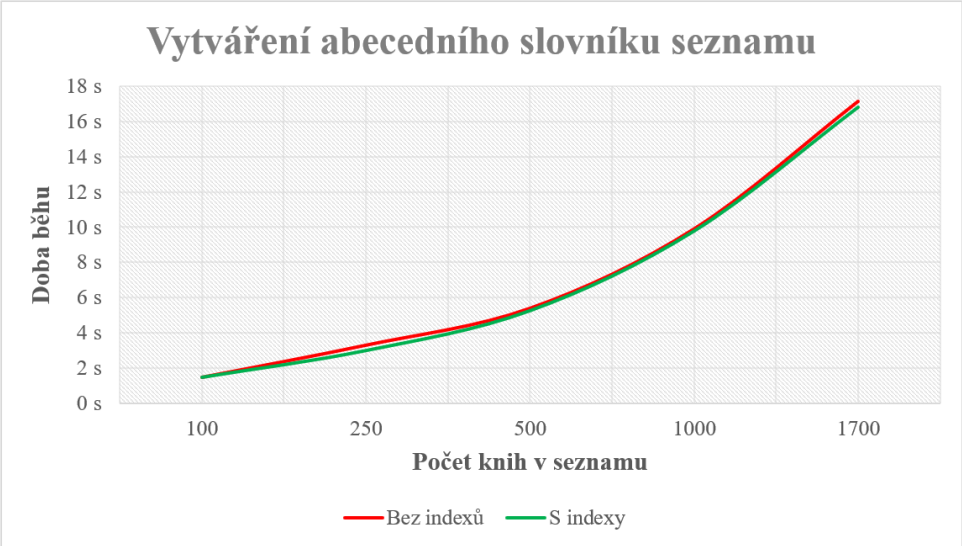
Tabulka 5.3: Doba běhu operací pro seznamy knih (bez indexace)

	100	**250**	**500**	**1000**	**1700**
Abecední slovník	1,461 s	3,283 s	5,377 s	9,880 s	17,117 s
Frekvenční slovník	1,415 s	3,393 s	5,316 s	10,002 s	16,877 s
Statistiky	1,654 s	3,308 s	5,428 s	11,021 s	18,909 s
Fulltextové vyhle. 1	0,063 s	0,198 s	0,508 s	1,496 s	3,865 s
Fulltextové vyhle. 2	0,043 s	0,125 s	0,415 s	1,308 s	3,547 s
Fulltextové vyhle. 3	0,040 s	0,130 s	0,392 s	1,279 s	3,499 s
Fulltextové vyhle. 4	0,036 s	0,145 s	0,378 s	1,292 s	3,631 s
Získání všech textů	0,111 s	0,281 s	0,512 s	1,119 s	2,178 s

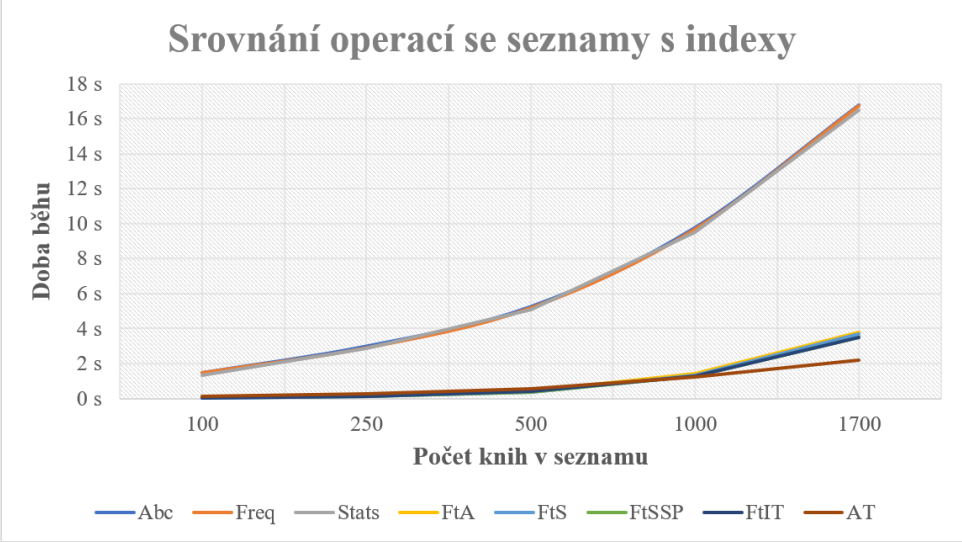
Tabulka 5.4: Doba běhu operací pro seznamy knih (s indexací)

	100	**250**	**500**	**1000**	**1700**
Abecední slovník	1,467 s	2,987 s	5,248 s	9,768 s	16,798 s
Frekvenční slovník	1,494 s	2,918 s	5,197 s	9,693 s	16,755 s
Statistiky	1,340 s	2,876 s	5,103 s	9,530 s	16,514 s
Fulltextové vyhle. 1	0,083 s	0,154 s	0,413 s	1,416 s	3,764 s
Fulltextové vyhle. 2	0,048 s	0,127 s	0,379 s	1,306 s	3,665 s
Fulltextové vyhle. 3	0,055 s	0,129 s	0,370 s	1,288 s	3,511 s
Fulltextové vyhle. 4	0,042 s	0,112 s	0,395 s	1,275 s	3,502 s
Získání všech textů	0,134 s	0,289 s	0,556 s	1,220 s	2,180 s

Obrázek 5.3: Doba běhu vytváření abecedního slovníku



Obrázek 5.4: Srovnání doby běhu operací se seznamy



Pro testování bylo nutné vytvořit sadu testovacích dat. K tomu byla implementována HTTP GET metoda na zdroj `/bookListTest/<N>`, která vrátí pole `books` obsahující N knih pro vytvoření seznamu knih. Tato metoda byla použita k získání polí obsahujících 100, 250, 500, 1000 a 1700 knih. Ty pak byly vloženy do 5 vytvořených seznamů. Metoda je v řešení pouze dočasně a je možné ji použít pouze pro účely testování. Z těchto důvodů se nenachází v dokumentaci Swagger.

Z naměřených hodnot je patrné, že použitím indexů je nepatrně snížen čas operace v případě vytváření slovníků a statistik nebo fulltextového vyhledávání. Časový rozdíl je však minimální, což je způsobeno tím, že většina úkonů v rámci operací indexy nevyužívá.

Fulltextové vyhledávání a získání všech textů knih ze seznamu jsou operace, které i pro velký počet dokumentů běží přijatelnou dobu. Vytváření abecedních slovníků a statistik je časově přijatelné, pokud je v seznamu maximálně 500 knih. V opačném případě může operace trvat i přes 10 vteřin. Například vytváření slovníků pro seznam obsahující 1700 knih trvá zhruba 17 vteřin. Pokud by tedy některý uživatel vytvořil seznam s více než 500 knihami, pak se může stát, že na odpověď serveru bude čekat klidně 5 až 20 sekund.

V této sekci byly zobrazeny grafy pouze pro některé naměřené hodnoty. Další grafy zobrazující srovnání hodnot z tabulek je možné nalézt v příloze C.

5.4 Systémové a akceptační testování

Předchozí části se týkaly testování pouze aplikační a datové vrstvy aplikace. Je třeba myslet na to, že aplikaci je také nutné testovat společně s frontendem aplikace jako funkční celek. K tomu slouží právě systémové testování. Toto testování ověřuje aplikaci z pohledu zákazníka. Jelikož vývoj celé aplikace probíhal v několika iteracích, ve kterých byly postupně implementovány jednotlivé funkcionality systému, došlo během každé iterace také k integraci všech částí aplikace a následně k integračnímu a systémovému testování. Během těchto testování bylo ověřeno, zda je zachována bezchybná komunikace jednotlivých komponent a zda se aplikace jako celek chová přesně tak, jak je očekáváno. Toto testování bylo prováděno se studentem Filipem Hladejem, který má na starosti implementaci prezentační vrstvy aplikace. Všechny chyby objevené tímto testování byly opraveny.

Po úspěšném proběhnutí všech předchozích testů je možné předat implementovanou aplikaci ÚČL. Ten pak provede akceptační testování, pomocí kterého je řešení testováno, zda odpovídá všem požadavkům. Pro účely tohoto testování byl vytvořen virtuální server v rámci výkonného fyzického serveru na Fakultě informačních technologií ČVUT. Na tomto virtuálním serveru běží operační systém Debian 10 a je k dispozici Intel Core Processor (Skylake) CPU s frekvencí 3 GHz a RAM paměť o velikosti 4 GB. Nasazení aplikace bylo provedeno způsobem, který je uveden v předchozí kapitole (viz 4.3). Následně byl vytvořen proxy server a aplikace je nyní dostupná a připravená na akceptační testování na veřejné adrese `https://ucl-poezie.fit.cvut.cz/`. Toto testování doposud neproběhlo z časových důvodů na straně ÚČL. Pokud budou během tohoto testování objeveny nějaké chyby, je potřeba provést v co nejkratším čase opravu a následně předat opravené řešení ÚČL k dalším testům.

Závěr

V diplomové práci jsem se zabýval vytvořením backendu webové aplikace pro Ústav pro českou literaturu Akademie věd České republiky, který se mi podařilo realizovat. Po důkladné analýze dat bylo vytvořeno společné schéma pro všechny digitalizované sbírky, které může ÚČL sloužit k pochopení dat digitalizovaných sbírek.

Implementované řešení nabízí nové datové úložiště, pomocí kterého lze uchovávat nejen samotné sbírky, ale také všechny informace o uživatelích včetně jejich uložených seznamů knih. Aplikační server poskytuje všechny základní operace pro práci s daty a navíc umožňuje také pokročilejší operace pro práci se seznamy knih. Společně s frontendem aplikace vyhovuje řešení téměř všem požadavkům ÚČL. Aplikace je připravena pro práci zaměstnanců ÚČL, ale mohou ji využít také běžní uživatelé jako čtenáři, studenti či literární badatelé.

V budoucnu je možné řešení libovolně rozšířit, ať už z pohledu datového úložiště nebo aplikačního serveru. Datové úložiště je implementováno tak, aby bylo možné přidat jakékoliv rozšíření v rámci kolekcí i databází. Očekává se, že v budoucnu vznikne nová kolekce, ve které budou uloženy naskenované obrázky knih. Aplikace je připravena na přidání autorizačních tokenů pro jednotlivé uživatele a také je připraveno asociativní pole pro ukládání cookies. REST API je možné libovolně rozšiřovat o nové metody, aniž by to mělo negativní vliv na přidružené klienty. Počítá se s tím, že v budoucnu budou přidány hlavně metody pro práci s verzemi knih.

Aplikace nyní běží na fakultním serveru. V nejbližší době dojde k přesunutí aplikace na server ÚČL.

Literatura

[1] Neo4j Console: Neo4j Console [online]. březen 2021, [cit. 2021-03-07]. Dostupné z: https://console.neo4j.org/

[2] MongoDB Inc.: The MongoDB 4.4 Manual [online]. Březen 2021, [cit. 2021-03-25]. Dostupné z: https://docs.mongodb.com/manual/

[3] GenMyModel: Class Diagram Online [online]. duben 2021, [cit. 2021-04-04]. Dostupné z: https://www.genmymodel.com/products/

[4] Redis Labs: Redis [online]. březen 2021, [cit. 2021-03-16]. Dostupné z: https://redis.io/

[5] Petr Bříza: Základy jazyka XPath [online]. duben 2004, [cit. 2021-03-23]. Dostupné z: https://www.interval.cz/clanky/zaklady-jazyka-xpath/

[6] Ústav pro českou literaturu AV ČR: O ústavu [online]. březen 2021, [cit. 2021-03-28]. Dostupné z: http://www.ucl.cas.cz/cs/

[7] Ústav pro českou literaturu AV ČR: Česká elektronická knihovna [online]. leden 2005, [cit. 2021-03-28]. Dostupné z: http://www.ceska-poezie.cz/cek/

[8] Jiří Kosek, Irena Holubová, Karel Minařík, David Novák: *Big Data a NoSQL databáze*. Grada Publishing, a.s., Praha, první vydání vydání, 2015, ISBN 978-80-247-5938-8.

[9] Martin Svoboda: Basic Principles (MIE-PDB.16:Advanced Database Systems) [online]. listopad 2020, [cit. 2021-03-20]. Dostupné z: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-08-Principles.pdf

[10] Linuxsoft.cz: Cassandra DB - I. [online]. červenec 2011, [cit. 2021-03-21]. Dostupné z: http://archiv.linuxsoft.cz/article.php?id_article=1850

[11] Palíšek, B. L.: Možnosti využití grafových databázív informačním systému internetového obchodu, diplomová práce. *Praha: České vysoké učení technické v Praze, Fakulta informačních technologií*, 2015.

[12] Neo4j, Inc.: Neo4j [online]. březen 2021, [cit. 2021-03-15]. Dostupné z: https://neo4j.com/

[13] Tomáš Kubica: NoSQL: vaše jednodušší, levnější a škálovatelnější databáze. srpen 2015, [cit. 2021-03-16]. Dostupné z: https://www.cloudsvet.cz/?p=243

[14] Vlastimil Klíma: Hašovací funkce, principy, příklady a kolize [online]. březen 2005, [cit. 2021-03-16]. Dostupné z: http://crypto-world.info/klima/2005/cryptofest_2005.htm

[15] Pavel Tišnovský: Databáze Redis (nejenom) pro vývojáře používající Python [online]. listopad 2018, [cit. 2021-03-16]. Dostupné z: https://www.root.cz/clanky/databaze-redis-nejenom-pro-vyvojare-pouzivajici-python/

[16] František Bártík: Cassandra DB - III. [online]. srpen 2011, [cit. 2021-03-17]. Dostupné z: http://text.linuxsoft.cz/article.php?id_article=1863

[17] Kohout, P.: Analýza databází vhodných do prostředí IOT, bakalářská práce. *Brno: Vysoké učení technické v Brně, Fakulta informačních technologií*, 2019.

[18] Apache Cassandra: Apache Cassandra Documentation v4.0-beta5 [online]. březen 2021, [cit. 2021-03-17]. Dostupné z: https://cassandra.apache.org/doc/latest/

[19] Microsoft: Nerelační data a NoSQL [online]. únor 2018, [cit. 2021-03-19]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/architecture/data-guide/big-data/non-relational-data

[20] Petr Valigura: Lekce 8 - B-stromy [online]. duben 2019, [cit. 2021-03-19]. Dostupné z: https://www.itnetwork.cz/navrh/algoritmy/algoritmy-vyhledavani/algoritmus-b-stromy

[21] Internet Engineering Task Force: XML Media Types, RFC 7303 [online]. červenec 2014, [cit. 2021-03-21]. Dostupné z: https://www.rfc-editor.org/rfc/rfc7303.txt

[22] Kosek, J.: *XML pro každého*. Grada Publishing, a.s., Praha, první vydání vydání, 2000, ISBN 80-7169-860-1.

[23] Jiří Kosek: Kapitola 3. XML schémata [online]. leden 2014, [cit. 2021-03-23]. Dostupné z: https://www.kosek.cz/xml/schema/wxs.html

[24] www.json.org: Introducing JSON [online]. březen 2021, [cit. 2021-03-23]. Dostupné z: https://www.json.org/json-en.html

[25] Martin Hassman: JSON : jednotný formát pro výměnu dat [online]. srpen 2008, [cit. 2021-03-23]. Dostupné z: https://zdrojak.cz/clanky/json-jednotny-format-pro-vymenu-dat/

[26] Michael Droettboom, Space Telescope Science Institute: JSON Schema Reference [online]. duben 2020, [cit. 2021-03-24]. Dostupné z: https://json-schema.org/understanding-json-schema/reference/index.html

[27] bsonspec.org: BSON specification [online]. březen 2021, [cit. 2021-03-24]. Dostupné z: http://bsonspec.org/

[28] Veronika Synková: Vybrané aplikace metadatového formátu TEI [online]. listopad 2007, [cit. 2021-03-24]. Dostupné z: https://ikaros.cz/vybrane-aplikace-metadatoveho-formatu-tei

[29] MongoDB Inc.: MongoDB Licensing [online]. Březen 2021, [cit. 2021-03-25]. Dostupné z: https://www.mongodb.com/community/licensing

[30] Michal Jurečko: Mongoeye [online]. červen 2007, [cit. 2021-03-27]. Dostupné z: https://github.com/mongoeye/mongoeye

[31] dragon119: Návrh webových rozhraní API [online]. leden 2018, [cit. 2021-04-06]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/architecture/best-practices/api-design

[32] Milan Jakel: Stavové kódy a hlášení v odpovědi protokolu HTTP [online]. květen 2002, [cit. 2021-04-06]. Dostupné z: https://www.interval.cz/clanky/stavove-kody-a-hlaseni-v-odpovedi-protokolu-http/

[33] Jiří Hrebenar: Úvod do JavaScriptu [online]. březen 2010, [cit. 2021-04-07]. Dostupné z: http://www.pestujemeweb.cz/obsah/javascript/javascript-uvod.php

[34] Jindřich Máca: Úvod do React [online]. březen 2019, [cit. 2021-04-08]. Dostupné z: https://www.itnetwork.cz/javascript/react/zaklady/uvod-do-react/

[35] Jindřich Máca: Úvod do Node.js [online]. duben 2018, [cit. 2021-04-08]. Dostupné z: https://www.itnetwork.cz/javascript/nodejs/uvod-do-nodejs

[36] expressjs.com: Express.js [online]. duben 2021, [cit. 2021-04-08]. Dostupné z: http://expressjs.com/

[37] npmjs.com: Mongoose [online]. duben 2021, [cit. 2021-04-08]. Dostupné z: https://www.npmjs.com/package/mongoose

Příloha A

Seznam použitých zkratek

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

BNF Backus–Naur form

BSON Binary JSON

CAP Consistency, Availability, Partition tolerance

CPU Central Processing Unit

CQL Cassandra Query Language

CRUD Create, Read, Update, Delete

CSV Comma Separated Values

DB Database

DDL Data definition language

DOM Document Object Model

DVD Digital Versatile Disc

E6S ECMAScript 6

FS File system

GNU GNU's Not Unix

GNU AGPL GNU Affero General Public License

GPL General Public License

HATEOAS Hypermedia as the Engine of Application State

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

I/O Input/Output

JPG Joint Photographic Group

JS JavaScript

JSON JavaScript Object Notation

MEAN MongoDB, Express.js, AngularJS, Node.js

MERN MongoDB, Express.js, React.js, Node.js

MEVN MongoDB, Express.js, Vue.js, Node.js

NPM Node.js Package Manager

ODM Object Document Mapping

ORM Object Relation Mapping

PDF Portable Document Format

RAM Random Access Memory

RBAC Role-based access control

RDF Resource Description Framework

REDIS Remote Dictionary Server

REST Representational State Transfer

SAX Simple API for XML

SCRAM Salted Challenge Response Authentication Mechanism

SGML Standard Generalized Markup Language

SOAP Simple Object Access Protocol

SPA Single Page App

SQL Structured Query Language

SSPL Server Side Public License

TCP/IP Transmission Control Protocol/Internet Protocol

TEI Text Encoding Initiative

TLS/SSL Transport Layer Security/Secure Sockets Layer

TTL Time to live

ÚČL AV ČR Ústav pro českou literaturu Akademie věd České republiky

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTC Coordinated Universal Time

UTF Unicode Transformation Format

W3C World Wide Web Consortium

XML Extensible markup language

XPATH XML Path Language

YAML YAML Ain't Markup Language

Příloha **B**

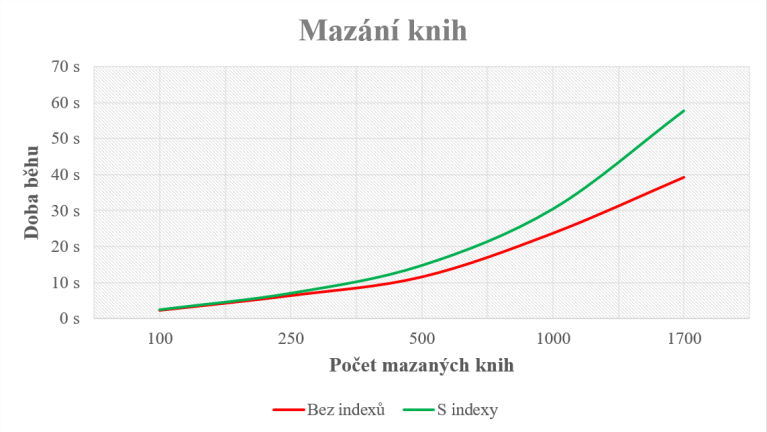
Obsah přiloženého DVD

- `BookApp` ..... adresář s implementovaným řešením backendu
  - `src` ..... adresář se zdrojovými kódy řešení
    - `main` ..... adresář s kódy
    - `test` ..... adresář s Unit testy
  - `package.json` ..... soubor obsahující všechny závislosti aplikace
  - `swagger.json` ..... soubor pro vytvoření dokumentace
- `JSONSchemas` ..... adresář s JSON schémata
- `MongoDB` ..... adresář se soubory týkající se databáze MongoDB
- `RESTapiTesting` ..... adresář s manuálním testováním REST API
- `Services` ..... adresář se službami
- `Schema` ..... adresář se schémata a modely
  - `bookDiagram.jpeg` ..... společné schéma knih
  - `bookValidationModel.jpeg` ..... validační schéma knih
  - `databaseModel.png` ..... databázové schéma
- `Text` ..... adresář s textem diplomové práce
  - `latexFiles` ..... zdrojová forma práce
    - `text.tex` ..... zdrojová forma práce ve formátu LaTeX
  - `text.pdf` ..... text práce ve formátu PDF
- `readme.txt` ..... stručný popis obsahu DVD

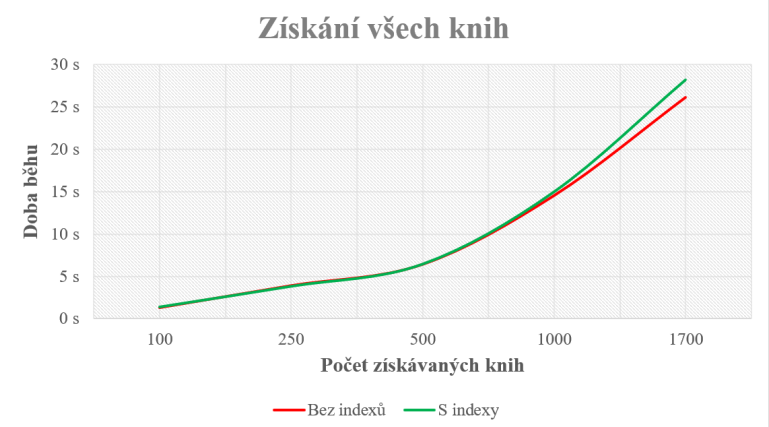
Příloha C

Výkonnostní testování - grafy

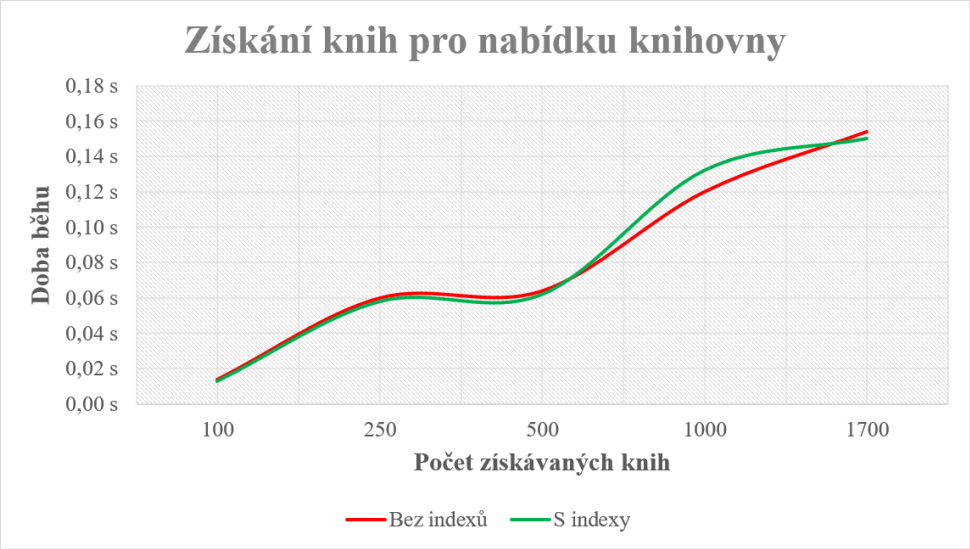
Obrázek C.1: Doba běhu mazání knih



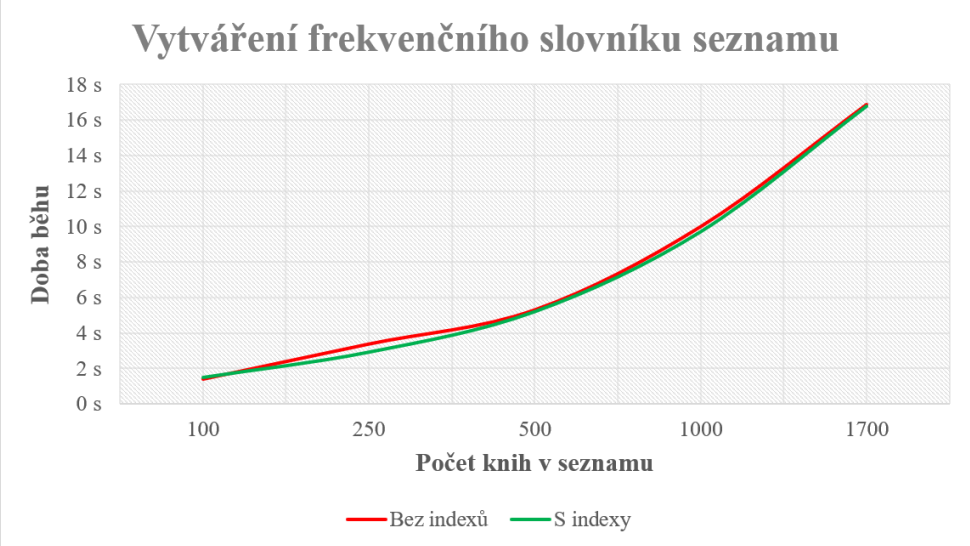
Obrázek C.2: Doba běhu získání knih



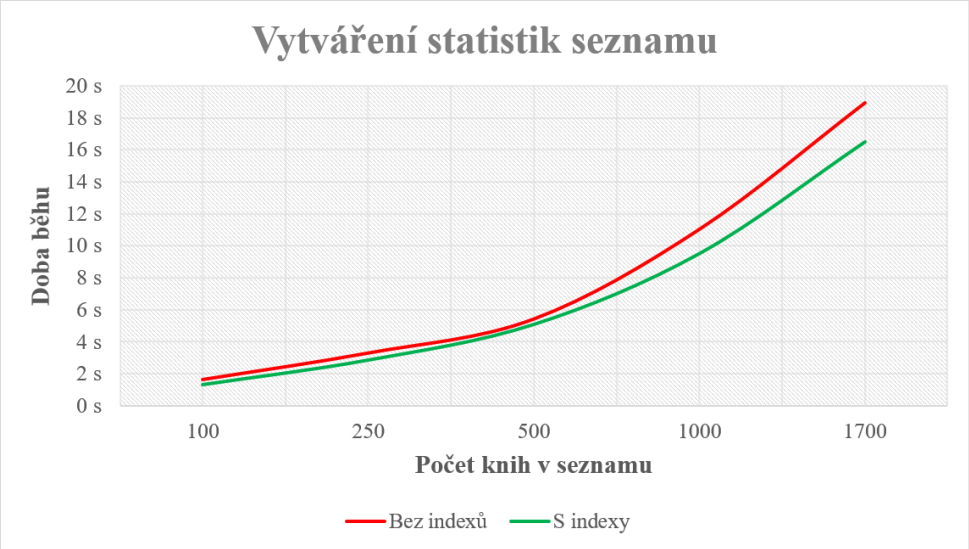
Obrázek C.3: Doba běhu získání knih pro knihovnu



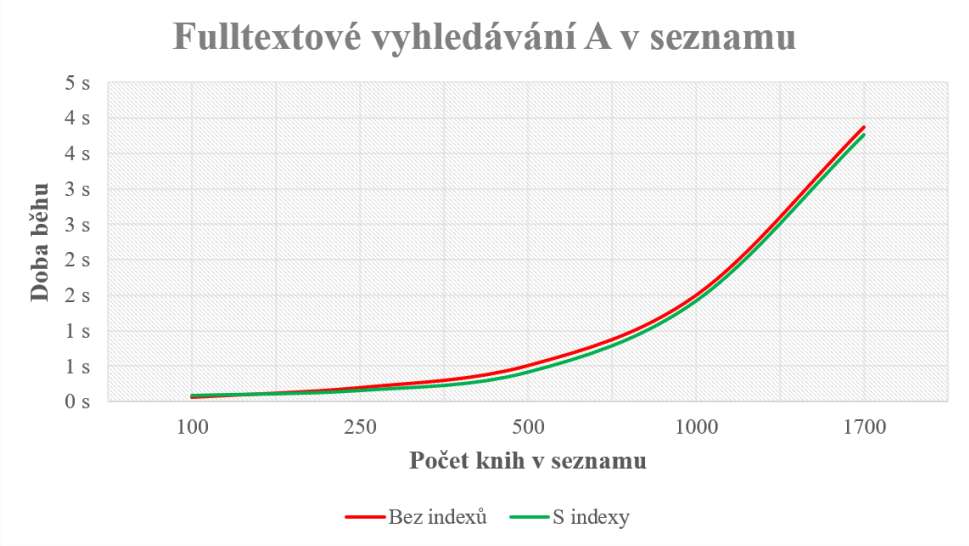
Obrázek C.4: Doba běhu vytváření frekvenčního slovníku



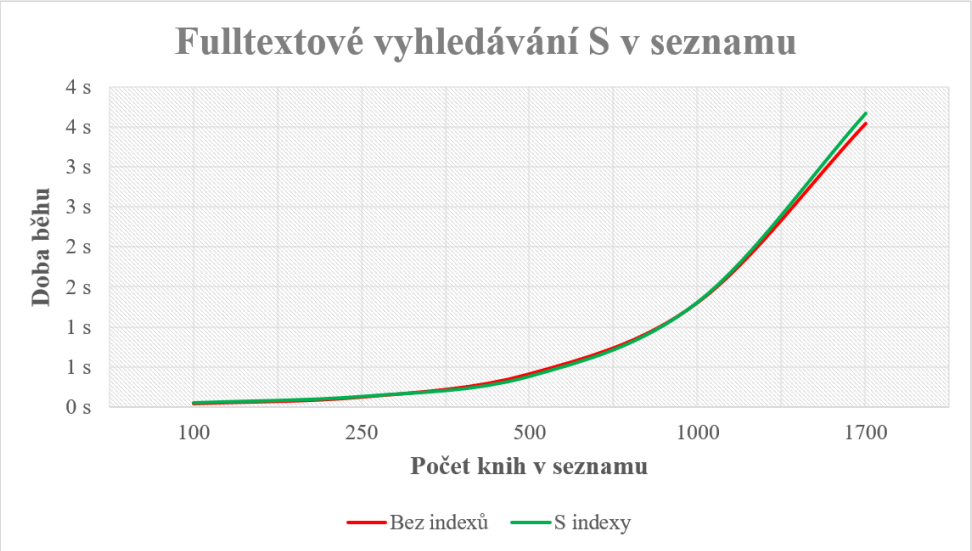
Obrázek C.5: Doba běhu vytváření statistik



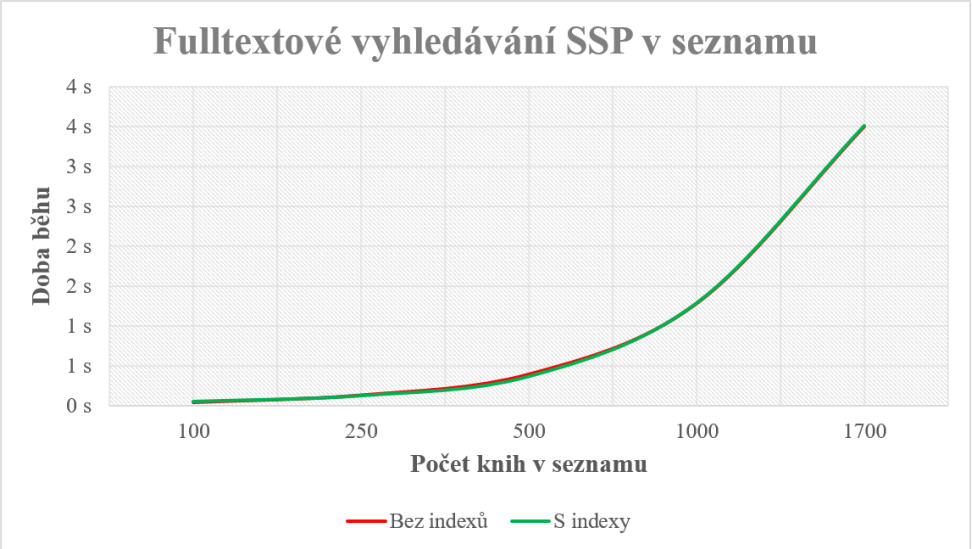
Obrázek C.6: Doba běhu fulltextového vyhledávání 1



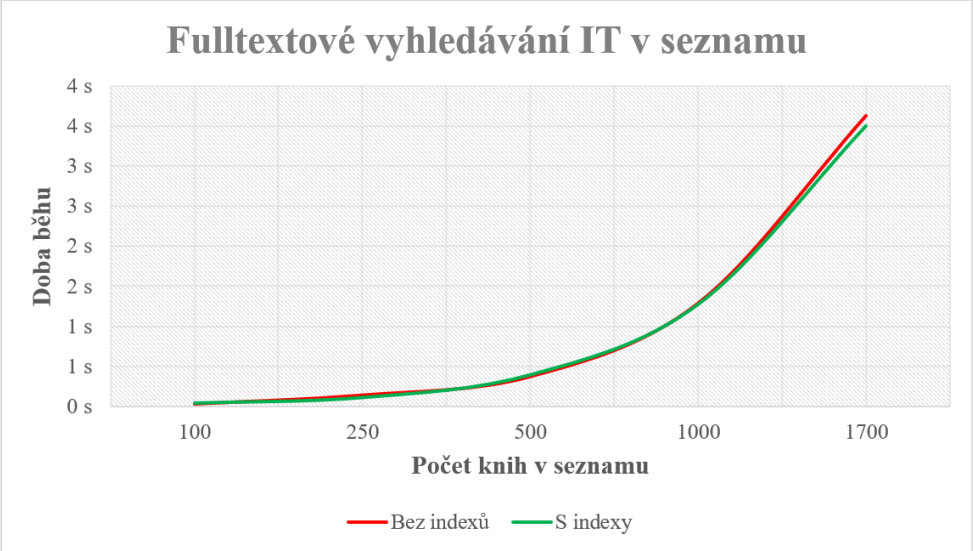
Obrázek C.7: Doba běhu fulltextového vyhledávání 2



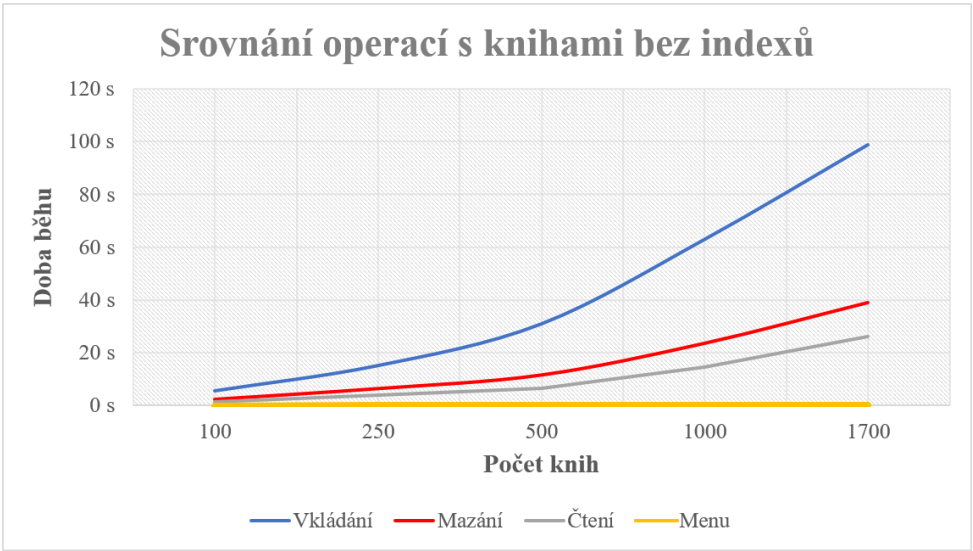
Obrázek C.8: Doba běhu fulltextového vyhledávání 3



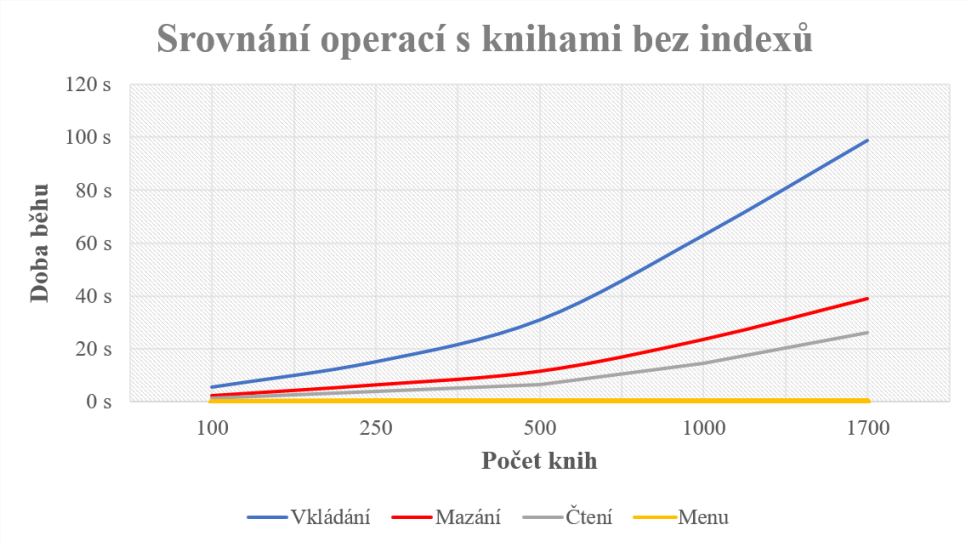
Obrázek C.9: Doba běhu fulltextového vyhledávání 4



Obrázek C.10: Doba běhu získání všech textů ze seznamu



Obrázek C.11: Srovnání doby běhu operací s knihami 2



Obrázek C.12: Srovnání doby běhu operací se seznamy 2