Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

NoSQL databáze MongoDB

Ivan Moroianu

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Ivan Moroianu

Informatika

Název práce

NoSQL databáze MongoDB

Název anglicky

NoSQL database MongoDB

Cíle práce

Cílem této práce je navrhnout, implementovat a otestovat aplikaci využívající NoSQL databázi MongoDB.

Metodika

V první části práce bude popis teoretických nástrojů a technik použitých v druhé praktické části s důrazem na MongoDB a jeho datový model ve srovnání s relační databázovou technologií. Budou stanoveny relevantní případy použití této nerelační databáze a bude vybrána a popsána příslušná sada nástrojů a technologií pro použití v praktické části. Druhá část práce bude obsahovat technickou dokumentaci vlastního příkladu včetně jeho testování. Budou dodržovány standardy softwarového inženýrství včetně UML.

Doporučený rozsah práce

40-60 stran

Klíčová slova

NoSQL; Dokumentově orientovaná databáze; MongoDB

Doporučené zdroje informací

Kyle Banker (2012) MongoDB in Action, ISBN 978-5-94074-831-1
MongoDB documentation, https://www.mongodb.com/docs/
Shakuntala Gupta Edward, Navin Sabharwal (2015) Practical MongoDB – Architecting, Developing, and Administering MongoDB, ISBN 978-1-4842-0647-8

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 12. 3. 2024

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 12. 3. 2024

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 12. 03. 2024

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "NoSQL databáze MongoDB" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2024 _____

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Vojtěchu Merunkovi, Ph.D. za vedení této bakalářské práce. Také bych rád poděkoval týmu Commerzbank OPENDBs za sdílení svých znalostí a zkušeností.

NoSQL databáze MongoDB

Abstrakt

Tato práce se věnuje popisu datového modelu MongoDB a jeho použití v praxi. V teoretické části jsou na základě porovnání s relačními databázemi identifikovány silné stránky dokumentově orientovaného datového modelu. Pro demonstraci výhod MongoDB oproti RDBMS je vybrána vhodná případová studie a sada nástrojů a technologií pro její realizaci. Po implementaci je aplikace otestována a jsou vyvozeny závěry ohledně použitelnosti této databáze.

Klíčová slova: dokumentově orientovaný datový model, NoSQL, MongoDB, kolekce, BSON, polymorfní schéma, PyMongo, logování, polostrukturovaná data, databáze.

NoSQL database MongoDB

Abstract

This thesis is dedicated to the description of the MongoDB data model and its practical use. In the theoretical part, the strengths of the document-oriented data model are identified based on a comparison with relational databases. A suitable case study and a set of tools and technologies for its implementation are selected to demonstrate the advantages of MongoDB over RDBMS. After implementation, the application is tested and conclusions are drawn regarding the usability of this database.

Keywords: document-oriented data model, NoSQL, MongoDB, collections, BSON, polymorphic schema, PyMongo, logging, semi-structured data, database.

Obsah

1 Úvod

Při práci s informacemi je často potřeba je uchovávat, přičemž množství a doba uchovávání mohou být neznámé. K řešení tohoto problému se používají databáze. Databáze jsou dnes nezbytnou součástí většiny moderních informačních a komunikačních systémů. Kromě objemu a doby uložení informací však vyvstávají otázky, v jaké formě by bylo racionální je ukládat pro optimální využití paměťového zařízení, usnadnění zápisu, modifikace, vyhledávání a mazání. Z tohoto důvodu se za dlouhou dobu vývoje databázových systémů objevilo velké množství paradigmat a datových modelů, z nichž každá má řadu individuálních výhod a nevýhod.

V současné době, kdy je k dispozici obrovské množství technologií, je důležité vědět, k čemu je vybraná technologie určena a na jaké konkrétní úkoly je zaměřena. Tím se lze vyhnout mnoha problémům spojeným s chybami při návrhu systému.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této práce je implementace aplikace využívající MongoDB. Zvolená případová studie a soubor technologií pro její realizaci by měly ukázat výhody dokumentově orientovaného datového modelu ve srovnání s klasickým relačním modelem.

2.2 Metodika

Teoretická část práce se bude věnovat popisu datového modelu MongoDB, jeho struktuře a vlastnostem. Důvody vzniku MongoDB budou přezkoumány v kontextu vývoje datových modelů. Tam, kde to bude vhodné, bude provedeno srovnání mezi dokumentově orientovaným modelem a relačním modelem. V praktické části práce bude podrobněji popsán jeden z případů použití uvedených v teoretické části a vybrán pro implementaci. Pro tento účel bude použita sada vhodných nástrojů a technologií. Realizace projektu bude popsána po krocích a bude otestována jeho finální verze.

3 Teoretická východiska

3.1 Datové modely a jejich rozvoj

Návrhy datových modelů se objevují již od 60. let 20. století, kdy první autoři publikovali své práce. V průběhu času se počet nových návrhů datových modelů začal jen zvyšovat. Michael Stonebraker a Joseph M. Hellerstein (1) rozdělili 35 let pokroku (časové období od 60. let do konce 90. let) do 9 "epoch":

- **Hierarchická epoc**ha (IMS): konec 60. a 70. léta 20. století;
- **Síťová epocha** (CODASYL): 70. léta 20. století;
- **Relační epocha**: 70. léta a začátek 80. let 20. století;
- **Epocha modelů entit a vztahů**: 70. léta 20. století;
- **Epocha rozšířených relačních modelů**: 80. léta 20. století;
- **Sémantická epocha**: konec 70. a 80. léta 20. století;
- **Objektově orientovaná epocha**: konec 80. a začátek 90. let 20. století;
- **Objektově-relační epocha**: konec 80. a začátek 90. let 20. století;
- **Polostrukturovaná epocha**: konec 90. let – současnost (minimálně do roku 2015).

Tato klasifikace nejen popisuje historii, ale také jasně ukazuje, jak každá následující kategorie datových modelů nabízí řešení omezení předchozího modelu, ale vytváří nová (1).

3.1.1 Před-relační éra

Systém IMS (IBM Information Management System) se objevil na trhu v roce 1968 a měl **hierarchický datový** model. Implementoval systém **typu záznamu**, což je kolekce pojmenovaných polí s přiřazenými **datovými typy**. Každá instance je nucena dodržovat datový formát uvedený v definici typu záznamu. Kromě toho musí některá podmnožina pojmenovaných polí jednoznačně specifikovat instanci záznamu, tj. musí být **klíčem**. A konečně, typy záznamů musí být uspořádány do stromu tak, aby každý typ záznamu (kromě kořene) měl jedinečný nadřazený typ záznamu. Celá databáze IMS je kolekcí instancí určitých typů záznamů kde, každá instance (kromě kořenových instancí) má jediného rodiče příslušného typu záznamu (1).

Hierarchický model narazil na dva problémy. Prvním je, že se informace opakují, což může vést k nekonzistenci. Druhým je, že stromově strukturované datové modely jsou velmi omezující a existence datové jednotky závisí na rodičích (1).

V roce 1969 zveřejnil výbor CODASYL (Committee on Data Systems Languages) svůj první report v sérii reportů s technologickými a jazykovými specifikacemi. CODASYL popsal první **síťový datový model** spolu s příslušným jazykem pro manipulaci s daty. Tento model uspořádal kolekci typů záznamů do sítě místo stromu. Instance záznamu tedy mohla mít více rodičů, nikoliv jednoho, jako je to v systému IMS (1).

Čas ukázal, že sítě jsou flexibilní, ale načítání a kopírování dat nebo oprava celých sítí v případě havárií je mnohem složitější než navrhování a správa hierarchií (1).

3.1.2 Relační model a jeho evoluce

Později v roce 1970 navrhl zaměstnanec IBM Edgar Codd svůj **relační model**. Zaměřil se na vytvoření systému, který by byl stejně flexibilní jako síťový datový model, ale poskytoval by nezávislost logických dat na fyzických datech (1) (2).

Ted Codd navrhl ukládat logická data do jednoduchých datových struktur – **tabulek**, a přistupovat k nim prostřednictvím vysokoúrovňového jazyka pro manipulaci s daty, čímž by se oddělil uživatel od fyzické struktury úložiště. Mezi tabulkami neměly být žádné ukazatele, takže vztahy jsou reprezentovány odpovídajícími datovými poli. Codd je však především matematik, takže metody a jazyky, které on navrhl (např. relační algebra a datový jazyk ALPHA) byly příliš složité na pochopení a používání (1).

Na konci 70. let si IMS již získal reputaci finančně úspěšného projektu. V této době již IBM vyvinula prototyp databáze založené na relačním modelu nazvaný "System R" a jazyk **SEQUEL** (Structured English Query Language), ale potenciál těchto technologií byl rozpoznán až v 80. letech (1).

V polovině 70. let minulého století navrhl Peter Chen **datový model entit a vztahů** (E-R) jako alternativu k relačnímu, síťovému a hierarchickému datovému modelu. Chen si představoval strukturu databáze jako soubor instancí entit, kde každá instance nebo objekt

může existovat nezávisle na ostatních entitách. Entity jsou charakterizovány atributy, které mohou být volitelné, povinné a/nebo jedinečné. Jeden nebo více povinných jedinečných atributů může tvořit tzv. klíč. Nejdůležitější součástí tohoto datového modelu jsou však **vztahy**. Vztahy mohou mít jednu z následujících forem: **1-to-1** (1:1), **1-to-many** (1:n), **many-to-1** (n:1) nebo **many-to-many** (m:n). Many-to-many nejsložitější typ vztahů může mít vlastní atributy (1).

Když byly entity a vztahy mezi nimi ve stadiu rozvinuté myšlenky, bylo nutné vytvořit pravidla pro **optimalizaci schématu** DB – **normalizaci** (1).

"Standardní přístup přívržence relací předpokládal, že návrh databáze se provede tak, že se sestaví počáteční soubor tabulek. Poté se na tento počáteční návrh aplikovala teorie normalizace." (1)

Vzhledem k tomu, že tento model byl také považován za podmnožinu Coddových dřívějších myšlenek, byla mu v průběhu desetiletí 70. let věnována jen malá pozornost (1).

Situace se změnila, když mladá společnost SDL (později Oracle corp.) uviděla v Coddově teorii správy dat komerční potenciál. Za základ si vzala projekt System R a Structured Query Language společnosti IBM (3). V roce 1979 vydala společnost Oracle, první komerční relační databázový program využívající jazyk **SQL**. V krátké době se stal tak úspěšným, že s ním IBM nedokázala držet krok (1) (4).

Poté začalo období aktivního vývoje a optimalizace technologií RDBMS. M. Sonebraker nazval toto období **"rozšířenou relační érou"** (Extended Relational Era) nebo érou **„ER++"** (1).

3.1.3 Éra post-relačních datových modelů

Souběžně s érou EP++ se objevil směr, který se snažil vyřešit problém relačních modelů, které nepropojují databázové entity s entitami popsanými v programech a aplikacích dostatečně dobře. "Post-relační datové modely" navržené Hammerem a McLeodem se jinak nazývaly **sémantické datové modely** (1) (5). Jejich cílem bylo poskytnout podrobnější a

smysluplnější zobrazení entit a vztahů mezi nimi. Logická vrstva tedy vycházela z fyzických dat a sémantická reprezentace měla být položena nad logickou vrstvou (5).

Později, v 80. letech 20. století široké využití objektově orientovaného programovacího paradigmatu vedlo k **objektově orientované** a později k **objektově-relační** éře. V tomto období se objevil koncept "perzistentních programovacích jazyků". Došlo k revizi způsobu interakce datových typů typických pro databáze a programovací jazyky. Také v této době byl vytvořen prototyp databáze "Postgres" (1).

Stonebraker a Hellerstein popsali vývoj datových modelů až do konce 90. let a poslední epochu nazvali **"polostrukturovanou"** (1).

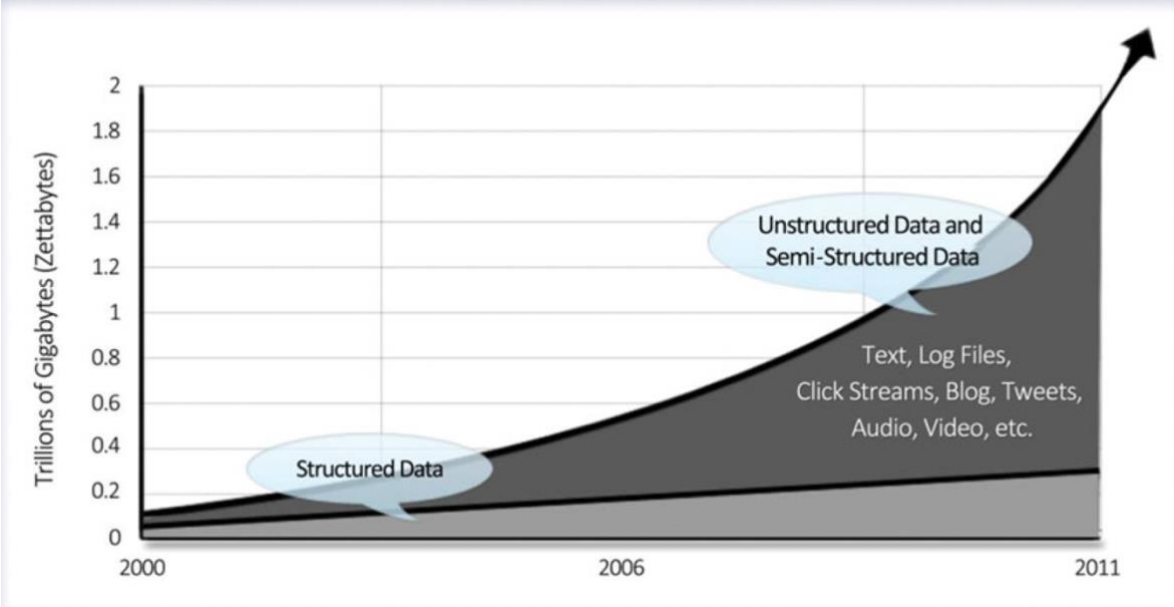
3.1.4 Polostrukturovaná epocha

Jestliže předchozí modely byly určeny k co nejpřesnějšímu popisu struktury dat, pak polostrukturované datové modely jsou určeny k ukládání dat s **nejasnou nebo předem neznámou strukturou** (1) (6).

"V případě semistrukturovaných dat informace, které jsou obvykle spojeny se schématem, jsou obsaženy v datech, která se někdy nazývají "samopopisná data".” (6)

Polostrukturované instance byly často nabízeny k uložení jako textové **dokumenty** (1). Nový způsob ukládání dat umožnil vývojářům přejít od principu **schéma na prvním místě** (schema first) k principu **schéma na posledním místě** (schema last) (1).

Významnou etapou tohoto období byl vznik formátu **XML** (7). Velkým impulsem pro vývoj technologií ukládání polostrukturovaných dat byl rozvoj internetu. Tradiční technologie RDBMS nemohly poskytnout vhodné řešení pro ukládání rostoucího množství blogů, fór, fotografií, videí atd. (8).



Obrázek 1- Poměr objemu strukturovaných a nestrukturovaných dat na internetu (8)

3.1.5 Závěr

Vývoj datových modelů prošel dlouhou cestou, ale zacyklil se. Návrhy polostrukturovaných datových modelů, které mají řešit omezení relačních schémat, jsou v některých ohledech podobné síťovému datovému modelu (1).

Vývoj datových modelů ukázal, že neexistuje univerzální model, který by řešil všechny problémy. Navíc se mezitím, co se diskový prostor zlevňuje, zvyšují náklady na správu a provoz komplexních DB systémů (9). Proto se dnes po dlouhé dominanci společností Oracle a Microsoft s jejich primárně relačními databázemi dostala mezi pět nejrozšířenějších DBMS dokumentově orientovaná databáze MongoDB (4) (10).

Rank			DBMS	Database Model
Feb 2024	Jan 2024	Feb 2023		
1.	1.	1.	Oracle	Relational, Multi-model
2.	2.	2.	MySQL	Relational, Multi-model
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model
4.	4.	4.	PostgreSQL	Relational, Multi-model
5.	5.	5.	MongoDB	Document, Multi-model

Obrázek 2 – žebříček technologií DB podle popularity na začátku roku 2024 (10)

3.2 MongoDB

„MongoDB je dokumentová databáze s požadovanou škálovatelností a flexibilitou a potřebným dotazováním a indexováním. ... dokumentový model MongoDB je pro vývojáře jednoduchý na zvládnutí a používání, a přitom poskytuje všechny možnosti potřebné pro splnění nejsložitějších požadavků v jakémkoli měřítku.“ (11)

3.2.1 Historie a příčiny vzniku

Dwight Merriman a Eliot Horowitz sestavili tým a v roce 2007 založili startup. Jeho cílem bylo vytvořit službu podobnou cloudu, která by poskytovala platformu pro vytváření, hostování a automatické škálování webových aplikací. Jinými slovy, chtěli vytvořit konkurenci pro produkty, jako je Google App Engine nebo Microsoft Azure. Vývojáři si uvědomili, že žádná open-source relační databáze neposkytuje škálovatelnost a flexibilitu, kterou potřebovali. Vedení startupu se rozhodlo, že je nutné vytvořit vlastní systém, odlišný od RDBMS, která by byla pružná, škálovatelná a snadno spravovatelná (8).

Obchodní model služby, kterou původně plánovali spustit, se zdál být neefektivní, nicméně tým se rozhodl pokračovat v práci na projektu databáze. V roce 2010 svět poprvé uviděl produkční verzi MongoDB (8).

3.2.2 MongoDB jako NoSQL databáze

S rostoucím počtem nerelačních datových modelů došlo k jejich kategorizaci a sloučení pod pojem „**NoSQL**” (9).

„Databáze NoSQL (neboli "nejen SQL") jsou netabulkové databáze a ukládají data jinak než relační tabulky. Databáze NoSQL existují v různých typech podle svého datového modelu. ... Poskytují flexibilní schémata a snadno se škálují při velkém množství dat a vysokém uživatelském zatížení.“ (12)

Existuje více způsobů klasifikace databází NoSQL, ale autoři dokumentace k MongoDB rozlišují 4 velké skupiny (12):

- Grafové databáze;
- Sloupcové databáze;

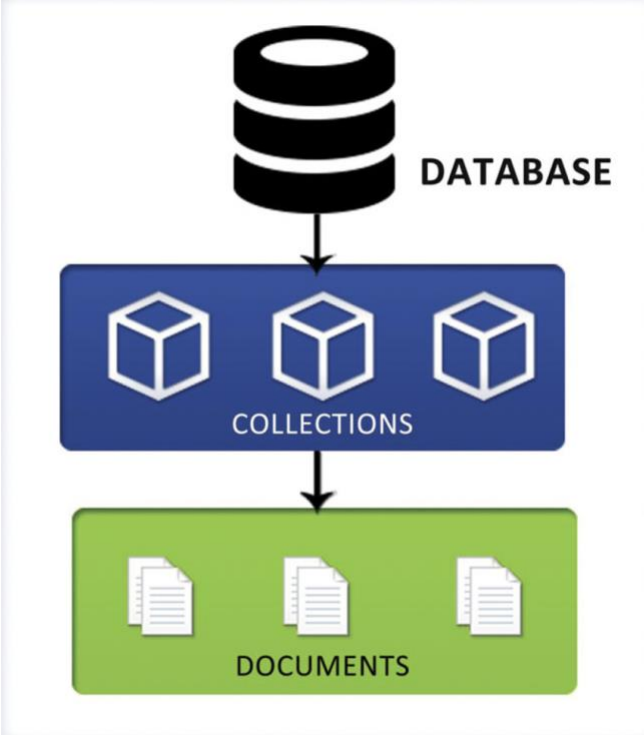
- Databáze typu „klíč-hodnota“;
- Dokumentové databáze.

Samotná MongoDB je klasifikována jako **dokumentová databáze**, kde se každý dokument skládá ze složek dvojic klíč-hodnota (12).

3.2.3 Koncepce databáze orientované na dokumenty

„Dokumentové databáze: umožňují ukládat a vyhledávat dokumenty v databázi, které mohou mít podobu XML, JSON, BSON (binární JSON) nebo jiných typů dokumentů se samopopisnou, hierarchickou stromovou strukturou dat, tvořenou mapami, kolekcemi a skalárními hodnotami.“ (13)

V RDBMS je obvyklá hierarchie, kdy v systému může existovat několik databází, databáze obsahují tabulky a při dodržení pravidel normalizace každý řádek tabulky je samostatným záznamem. V MongoDB hierarchie je podobná (8).



Obrázek 3 - Databázová struktura MongoDB (8)

I přes zásadně odlišné datové modely lze mezi MongoDB a jejími relačními konkurenty najít řadu paralel. Autoři dokumentace MongoDB tvrdí, že lze za odpovídající považovat řadu pojmů (viz Tabulka 1).

RDBMS	MongoDB
databáze	databáze
tabulka	kolekce
tabulkový řádek	dokument
tabulkový sloupec	pole dokumentu
primární klíč	Pole "_id"

Tabulka 1 - Terminologie relačních databází a související pojmy týkající se MongoDB (14)

3.2.4 Kolekce

Pokud jsou databáze v systému MongoDB obdobou databází v relačních systémech, pak o 1 úroveň níže jsou **kolekce**, nikoli **tabulky**.

„MongoDB ukládá dokumenty do kolekcí. Kolekce jsou obdobou tabulek v relačních databázích." (15)

Protože datové typy definované ve strukturách tabulek RDBMS jsou pevně dané, může vývojář do sloupce přidat pouze data určitého datového typu. Kolekce MongoDB nemají žádnou strukturu, a jejich účelem je logicky seskupovat dokumenty (16).

Různé zdroje popisují kolekce jako bezeschémové (16), s flexibilním (dynamickým) schématem nebo s polymorfním schématem (8).

„Dynamické schéma znamená, že dokumenty v rámci jedné kolekce mohou mít stejné nebo různé sady polí nebo struktury, a dokonce i společná pole mohou v různých dokumentech uchovávat různé typy hodnot. Ve způsobu ukládání dat v dokumentech kolekce není žádná rigidita." (8)

Kromě standardních kolekcí poskytuje MongoDB uživatelům:

- "capped" kolekce – kolekce s pevnou velikostí, které při přidání nahrazují nejstarší dokumenty nejnovějšími (17);
- kolekce "časových řad" – kolekce s povinným polem časového razítka, které umožňují efektivní dotazování na základě časových operací (18);

- "custom collation" kolekce – kolekce s uživatelsky definovanou sadou pravidel pro práci s daty, například jak porovnávat řetězce (19);
- "clusterové" kolekce – kolekce rozdělené do více částí, ale sjednocené jedním clusterovým klíčem (20).

Obvykle je velikost kolekce omezena místem dostupným na serveru a jeho souborovém systémem, ale díky shardingu (rozdělení mezi více serverů) může být jeji velikost až 32 terabajtů (21).

3.2.5 Dokumenty

„Představte si dokument jako řádek v SQL“ (8)

JSON (JavaScript Object Notation) je formát pro výměnu textových dat a odpovídající přípona souboru, které byly specifikovány na začátku roku 2000 jako alternativa k XML. Dnes jsou oba tyto formáty relevantní, ačkoli JSON je pro člověka snáze čitelný, váží méně a rychleji se zpracovává (22).

Vizuálně vypadá JSON jako neuspořádaný seznam dvojic klíč-hodnota. Klíč je od hodnoty oddělen dvojtečkou a celý objekt je uzavřen v kulatých závorkách. Pokud je v objektu mnoho párů, jsou odděleny čárkami (22).

Vývojáři MongoDB považovali JSON za vhodný formát pro svůj systém, ale on měl dvě nevýhody. První je malý počet podporovaných datových formátů (23):

- string;
- number;
- boolean;
- array;
- null;
- object (vnořený JSON objekt).

Druhým problémem je to, že neomezená délka objektů a jejich vlastností by mohla negativně ovlivnit výsledný výkon databázového systému. Tyto dva problémy vedly k potřebě vytvořit nový formát souborů (23).

BSON je zkratka "Binary JSON". Jestliže JSON kóduje data v řetězci UTF-8, BSON je kóduje v binární podobě. Tento formát byl navržen speciálně pro MongoDB, aby se zvýšila rychlost při zpracování dat a optimalizovalo využití diskového prostoru. V MongoDB se BSON používá také pro binární kódování příkazů a dotazů (23).

Co se týče počtu podporovaných datových typů, ten se oproti JSON také zvýšil. JSON například nepodporuje typ “Date“ (časový datový typ), nepodporuje binární data a nerozlišuje různé velikosti celých čísel (23) (24).

Podle dokumentace BSON k začátku roku 2024 Podporuje 21 datových typů, avšak 4 z nich jsou již zastaralé a nepoužívané (viz Tabulka 2).

Kód	**Datový typ**	**Zastaralý**
1	Double	
2	String	
3	Object	
4	Array	
5	Binary data	
6	Undefined	ano
7	ObjectId	
8	Boolean	
9	Date	
10	Null	
11	Regular Expression	
12	DBPointer	ano
13	JavaScript	
14	Symbol	ano
15	JavaScript code with scope	ano
16	32-bit integer	
17	Timestamp	
18	64-bit integer	
19	Decimal128	
-1	Min key	
127	Max key	

Tabulka 2 - Seznam datových typů podporovaných BSON (24)

Je důležité si uvědomit, že vzhledem k tomu, že BSON je binární formát, on není čitelný pro člověka a jakákoli textová reprezentace je ve skutečnosti adaptace JSON (viz Ukázka kódu 1) (23).

```
{"hello": "world"} →
\x16\x00\x00\x00          // total document size
\x02                      // 0x02 = type String
hello\x00                 // field name
\x06\x00\x00\x00world\x00  // field value
\x00                      // 0x00 = type EOO ('end of object')
```

Ukázka kódu 1 - Příklad JSON konvertovaného do BSON (23)

3.2.6 ObjectId a indexy

Pokud se podíváte na sedmý řádek v tabulce datových typů (viz Tabulka 2) podporovaných formátem BSON, uvidíte **ObjectId**.

ObjectId je speciální 12bajtový **datový typ** pro ukládání identifikátorů nebo indexů. Stejně jako **primární klíče v RDBMS** jsou indexy unikátní hodnoty. Používají se k identifikaci každého dokumentu v rámci kolekce a jejich hlavním účelem je podpora efektivních dotazů (16).

Defaultní index je vytvořen MongoDB. Je přiřazen povinnému poli "_id". Má následující strukturu (16):

1. První 4 bajty udávají čas, kdy byl objekt vytvořen. Čas se počítá v sekundách od 1. ledna 1970 (unixová epocha);
2. Další 3 bajty jsou generovány náhodně. Označují počítač;
3. Další 2 bajty jsou generovány společně s předchozími 3. Identifikují proces vytvoření objektu;
4. Poslední 3 bajty jsou lokální čítač, který se inkrementuje při každém generování nového ID.

Klíči však můžeme přiřadit vlastní hodnotu nebo můžeme kolekci indexovat podle jiného pole, pokud je zaručena jeho unikátnost (8).

MongoDB také umožňuje vytvářet složené indexy. Tyto indexové struktury udržují odkaz až na 32 klíčů v rámci dokumentu kolekce (21).

3.2.7 Struktura dokumentu jako schéma

S rozvojem polostrukturovaného datového modelu a později s vznikem dokumentově orientovaných databází přešli vývojáři od přístupu "schéma na prvním místě" k přístupu "schéma na posledním místě" (1).

MongoDB nemá explicitní schéma a data se popisují samy. To znamená, že informace jsou uloženy v dokumentech, které mohou mít různé sady polí s různými datovými typy pro každé pole. Praxe však ukazuje, že ve většině případů dokumenty v rámci jedné kolekce mají podobnou strukturu a liší se jen částečně (16).

> *„Teoreticky každý dokument v kolekci může mít zcela odlišnou strukturu; v praxi bude dokument kolekce relativně jednotný.“* (16)

V tomto případě je řeč o polymorfním schématu (8). Pokud si představíme kolekci "Users", kde každý dokument bude popisovat uživatele, pak potřebujeme sadu polí pro jeho definici.

```
{
  "_id": {
    "$oid": "65d5ecb53b7451cd5ce3ef62"
  },
  "first_name": "Jan",
  "last_name": "Novak",
  "telephone": [
    "111 111 111"
  ],
  "email": "aaa@seznam.cz",
  "address": "..."
}
```

Ukázka kódu 2 - Dokument s vyplněnou e-mailovou adresou (vlastní zpracování)

```
{
  "_id": {
    "$oid": "65d5ecb53b7451cd5ce3ef63"
  },
  "first_name": "Jana",
  "last_name": "Novakova",
  "telephone": [
    "222 222 222",
    "333 333 333"
  ],
  "address": "..."
}
```

Ukázka kódu 3 - Dokument bez pole pro e-mail (vlastní zpracování)

Ve výše uvedeném příkladu (viz Ukázka kódu 2 a 3) jsou uvedeny 2 záznamy z kolekce „Users“. Oba uživatelé mají společná pole "first_name", "last_name", "telephone" a "address". Ale druhý uživatel na rozdíl od prvního nemá pole "email" a pole "telephone" obsahuje několik telefonních čísel místo jednoho.

Tento flexibilní přístup k ukládání dat zbavuje vývojáře řady omezení a může také ušetřit spoustu času v počáteční fázi vývoje aplikace, kdy ještě není jasné, jaká data budou později nutná (8). Pokud však aplikace používající MongoDB prošla všemi fázemi vývoje a programátoři potřebují záruku, že uložená data budou v souladu se standardy, mohou použít validaci dokumentů (25).

„JSON schéma je slovník, který umožňuje anotovat a ověřovat dokumenty. Pomocí schématu JSON můžete specifikovat pravidla validace polí v lidsky čitelném formátu.“ (25)

MongoDB podporuje standard „JSON Schema Draft 4“. Díky validacím můžeme na dokumenty aplikovat různá omezení, a takhle zajistit integritu a konzistenci dat. Mezi nejčastější omezení patří (25):

- existence polí
- datové typy
- hodnoty polí atd.

```
{
  "$jsonSchema": {
    "bsonType": "object",
    "title": "User validation object",
    "required": [
      "first_name",
      "last_name",
      "telephone",
      "email"
    ]
  }
}
```

Ukázka kódu 4 - validační schéma JSON (vlastní zpracování)

Přidáním schématu jako ve výše uvedeném příkladu (viz Ukázka kódu 4) způsobíme při validaci druhého dokumentu chybu "Validation Error". První dokument projde validací, protože obsahuje všechna požadovaná pole.

3.2.8 Normalizace

„*V relačních databázích se při modelování dat obvykle postupuje tak, že se definují tabulky a postupně se odstraňuje redundance dat, aby se dosáhlo normální formy.“* (8)

Normalizace začíná vytvářením tabulek podle logiky a požadavků aplikace a následným postupným odstraňováním redundance, aby se dosáhlo co nejvyšší normální formy. Obvykle je akceptovatelná **3NF**. Třetí normální forma zakazuje přítomnost tranzitivních závislostí v rámci jedné tabulky. To lze vyřešit použitím **dalších tabulek k oddělení dat**. (26)

Předpokládejme, že existuje relační databáze s definovanými 2 tabulkami: "Book" a "Author". V tomto případě bychom nemuseli ukládat všechny informace o autorovi do popisu každé knihy. Stačilo by pouze uchovávat odkaz na druhou tabulku, kde by byly všechny informace uloženy v jediné exempláři. Normalizace odstraňuje redundanci a zjednodušuje proces aktualizace informací. Absence redundance šetří místo na disku a snadná aktualizace pomáhá udržovat informace konzistentní. (26) (8)

Potíže však nastanou, když chceme data **vytáhnout** (8). Aby mohl vývojář používající relační databázi shromáždit všechna data o knize do jedné celistvé sestavy, musí provést operaci **JOIN**. Pokud v databázi nejsou 2, ale 10 nebo dokonce 50 tabulek, proces extrakce dat se značně zpomalí a samotný SQL dotaz bude rozsáhlý a obtížně pochopitelný.

„Obecně platí, že každý RDBMS čte z disku a provádí vyhledávání, které zabere více než 99 % času stráveného čtením řádku. Operace JOIN je jednou z nejdražších operací v relační databázi. Pokud navíc nakonec potřebujete databázi škálovat na více serverů, nastává problém s generováním distribuovaného JOIN'u, což je složitá a obecně pomalá operace." (8)

Normální formy jsou souborem pravidel a technik pro efektivní návrh schémat obsahujících vztahy (26). Protože MongoDB není relační databáze, když mluvíme o procesu normalizace, jedná se o **nalezení optimální struktury dokumentů**. Ze stejného důvodu MongoDB nemá operátor JOIN, ale má dvě alternativy (27):

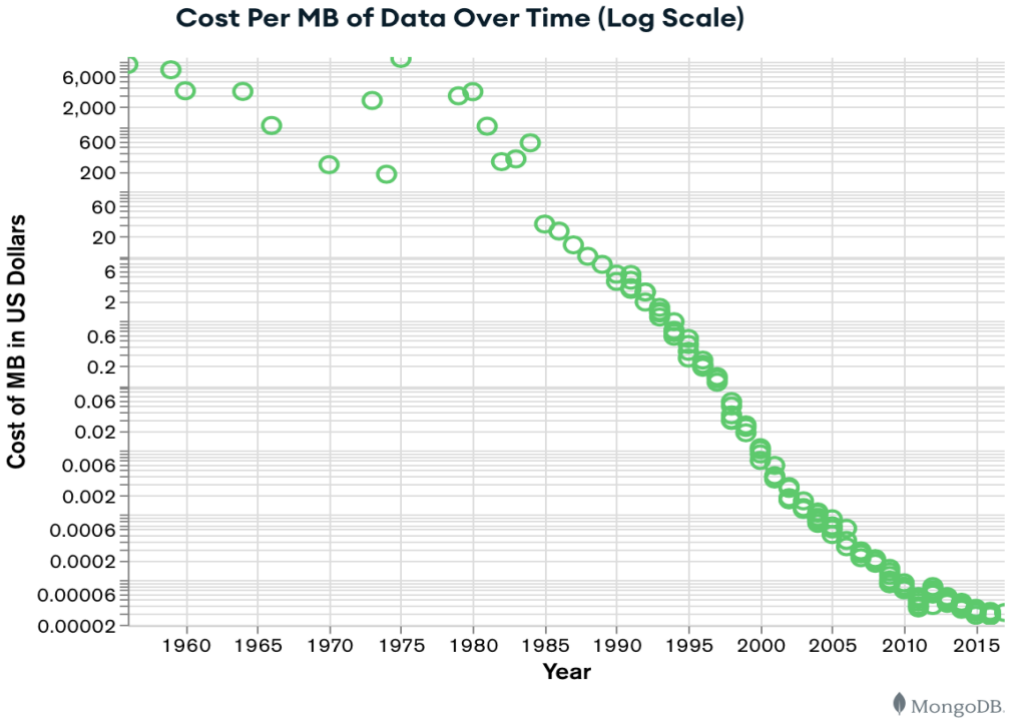
1. **Vložení;**
2. **Odkazování.**

3.2.9 Vložení vs. odkazování

Zatímco u RDBMS by vložení znamenalo porušení normálních forem, u MongoDB ono může mít pozitivní dopad na výkon (26) (8).

„Vkládání může být užitečné, pokud chcete načíst nějakou sadu dat a zobrazit ji na obrazovce, například stránku, která zobrazuje komentáře spojené s blogem; v tomto případě lze komentáře vložit do dokumentu blogu. Výhodou tohoto přístupu je, že vzhledem k tomu, že MongoDB ukládá dokumenty na disku spojitě, lze všechna související data načíst při jediném vyhledávání.“ (8)

V příkladu knih a autorů by "vnoření" znamenalo **zvýšení produktivity při extrakci dat za cenu duplikování** informací o autorovi. Ale již v roce 2017 byla cena jednoho megabajtu na trvalém úložišti přibližně 0,000032 amerického dolaru. Úspora místa dnes již není tak akutní potřebou jako před 50 lety (viz Obrázek 4) (12).



Obrázek 4 - Graf zobrazující pokles průměrné ceny za místo na trvalých nosičích *(12)*

Pokud však vznikne potřeba často aktualizovat vložená data, bude nutné vyhledávat každý výskyt, což představuje velký problém pro zachování konzistence dat. Další nevýhodou vložení je to, že při vyhledávání konkrétní knihy v databázi získáme všechny informace o jejím autorovi, i když to není nutné (viz Ukázka kódu 5) (8).

```
{
  "_id": {
    "$oid": "65d75e318344189eec0538c8"
  },
  "title": "Metamorphosis",
  "publication_year": 1915,
  "author": {
    "first_name": "Franz",
    "last_name": "Kafka",
    "year_of_birth": 1883,
    "year_of_death": 1924,
    "Citizenship": "Austrian",
    "Occupations": [
      "novelist",
      "short story writer",
      "insurance officer"
    ],
    "Style": "Modernism"
  },
  "pages": 44,
  "rating": 4.5,
  "price": 3.99
}
```


Ukázka kódu 5 - Dokument popisující knihu, obsahující vložený dokument s informacemi o autorovi (vlastní zpracování)

Alternativou k vloženým dokumentům je odkazování. Tato metoda přibližuje strukturu dokumentu normalizovaným relačním tabulkám (8) (27).

„Výsledkem odkazování jsou normalizované datové modely, protože data jsou rozdělena do více kolekcí a nejsou duplikována.“ (27)

Myšlenka odkazování spočívá v tom, že namísto ukládání všech souvisejících dat do jednoho dokumentu je lze rozdělit do **několika kolekcí**. Do dokumentu pak lze vložit **odkaz** na ID druhého souvisejícího dokumentu (viz Ukázka kódu 6 a 7) (27).


```
{
  "_id": {
    "$oid": "65d75c678344189eec0538c5"
  },
  "first_name": "Franz",
  "last_name": "Kafka",
  "year_of_birth": 1883,
  "year_of_death": 1924,
  "Citizenship": "Austrian",
  "Occupations": [
    "novelist",
    "short story writer",
    "insurance officer"
  ],
  "Style": "Modernism"
}
```


Ukázka kódu 6 - Dokument s informacemi o autorovi umístěný v oddělené kolekci (vlastní zpracování)


```
{
  "_id": {
    "$oid": "65d760238344189eec0538c9"
```

```
  },
  "title": "Metamorphosis",
  "publication_year": 1915,
  "author": "65d75c678344189eec0538c5",
  "pages": 44,
  "rating": 4.5,
  "price": 3.99
}
```

Ukázka kódu 7 - Dokument popisující knihu, který obsahuje odkaz na dokument s informacemi o autorovi z jiné kolekce (vlastní zpracování)

Pro získání všech informací o knize, včetně autora, je nyní nutný operátor **$lookup**, který provede 2 vyhledávání v databázi. První iterace je zaměřena na vyhledání knihy na základě zadaného filtru a druhá na vyhledání autora s odpovídajícím ID. Při práci s rozsáhlými dokumenty však může odkazování naopak urychlit proces zápisu na disk. Protože MongoDB zapisuje dokumentu atomicky, je jednodušší aktualizovat informace v malém odkazovaném dokumentu než přepisovat velký (8).

"Při řešení problému s návrhem schématu již neexistuje pevná cesta normalizovaného návrhu databáze, jako je tomu u relačních databází. V MongoDB závisí návrh schématu na problému, který se snažíte vyřešit." (8)

Vzhledem k tomu, že neexistují přesná pravidla, je třeba při navrhování schématu zohlednit následující faktory (27):

- typ vztahů mezi entitami;
- průměrná velikost dokumentu;
- typické dotazy a případy použití;
- očekávaná četnost operací zápisu a čtení.

„Obecně platí, že pokud je vzor dotazu vaší aplikace dobře známý a k datům se přistupuje pouze jedním způsobem, vložený přístup funguje dobře. Pokud se vaše aplikace může dotazovat na data mnoha různými způsoby nebo pokud nejste schopni předvídat vzory, podle kterých mohou být data dotazována, může být vhodnější "normalizovanější" přístup." (8)

3.2.10 Omezení

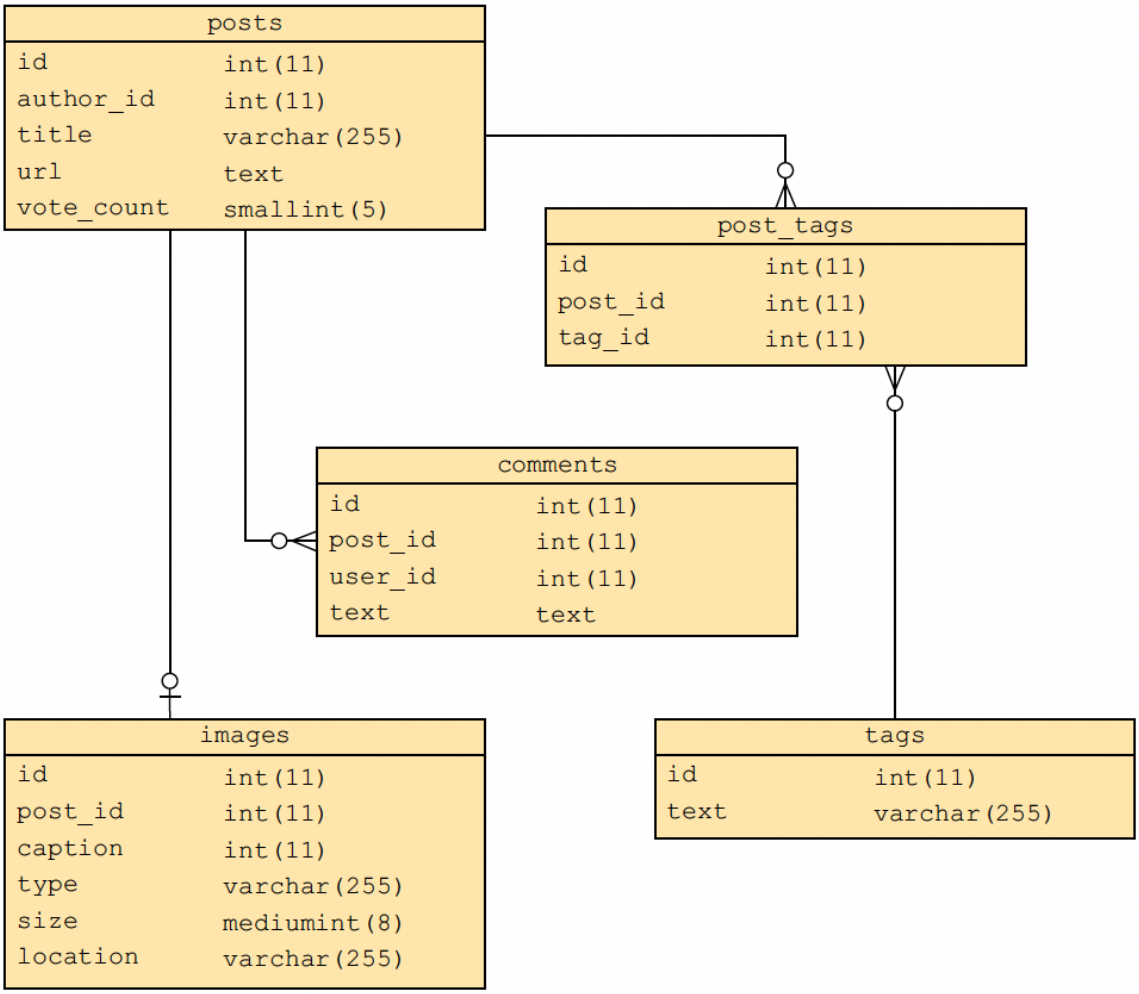
Aby MongoDB fungovala efektivně, je v její architektuře zakotvena řada omezení. Maximální velikost dokumentu JSON je obvykle omezena pouze parserem a serializátorem používaným systémem (22), ale maximální velikost dokumentu BSON je striktně omezena na 16 megabajtů. Pokud je nutné ukládat dokumenty nebo soubory přesahující tento limit, používá se technologie GridFS k jejich rozdělení na jednotky (tzv „chunks“). Toto opatření zabraňuje nadměrné spotřebě paměti RAM a také snižuje zatížení disku při čtení a zápisu. MongoDB také nepodporuje více než 100 úrovní vnoření dokumentů a polí (21).

Pole "_id" je rezervováno pro primární klíče. Hodnota klíče musí být v kontextu kolekce unikátní. Hodnotou "_id" nemůže být pole (array), může to však být dokument, pokud jeho klíče nemají jako první symbol znak "$". V případě potřeby dalšího indexování pro rychlejší vyhledávání, jedna kolekce podporuje až 64 indexů. Složený index může odkazovat až na 32 polí (21).

3.2.11 Přímé porovnání relačního schématu a struktury dokumentu

Flexibilní schéma je velkou výhodou pro vývojáře používající MongoDB ve srovnání s pevnými tabulkovými schématy používanými v relačních databázích. Tuto výhodu dobře demonstruje příklad, který uvedl Kyle Banker (16).

Pro uložení postu ze zpravodajské sítě do relační databáze je nutné nejprve vytvořit schéma.



Obrázek 5 - Schéma ukládání dat v relační databázi pro zpravodajský portál (16)

Výše uvedený ER (entity-relational) diagram (viz Obrázek 5) ukazuje, že každý post má ID, odkaz na autora (tabulka s autory je z příkladu pro zjednodušení vyřazena), název, odkaz URL a počet hlasů. Dále může mít každý post 0 nebo hodně komentářů, 0 nebo 1 obrázek a je také spojen s tabulkou tagů vazbou “m:n“.

Pro zobrazení kompletního postu v aplikaci, včetně všech jeho komentářů, fotografií a tagů, je tedy třeba zkombinovat informace ze 4 tabulek.

Pokud některý řádek v tabulce potřebuje další pole, musí vývojář tabulku explicitně změnit. Pokud se změní struktura stránky obsahující post, například zrušíme omezení počtu fotografií a přidáme "reakce", povede to k nutnosti změnit celé schéma (16).

Níže je uveden příklad, jak by mohl vypadat záznam v databázi, kdyby se jednalo o MongoDB (viz Ukázka kódu 8). Údaje jsou reprezentovány v notaci JavaScriptu.

```
{
   _id: ObjectID('4bd9e8e17cefd644108961bb'),
   title: 'Adventures in Databases',
   url: 'http://example.com/databases.txt',
   author: 'msmith',
   vote_count: 20,
   tags: [
      'databases',
      'mongodb',
      'indexing'
   ],
   image: {
      url: 'http://example.com/db.jpg',
      caption: 'A database.',
      type: 'jpg',
      size: 75381,
      data: 'Binary'
   },
   comments: [
      {
         user: 'bjones',
         text: 'Interesting article.'
      },
      {
         user: 'sverch',
         text: 'Color me skeptical!'
      }
   ]
}
```

Ukázka kódu 8 - Příklad dokumentu v MongoDB obsahujícího informace o postu (16)

Díky možnosti vytvářet vnořené dokumenty všechna data jsou uložena společně.

- Zde je pole "_id" primárním klíčem, který identifikuje post;
- Hodnoty atributů „title“, „url“, „author“ a „vote_count“ jsou uloženy jako hodnoty příslušných polí;
- Tagy jsou reprezentovány jako pole řetězců;
- Obrázek je vnořený dokument;
- Komentáře jsou reprezentovány jako pole objektů.

Při používání MongoDB by se vývojáři již neměli řídit zásadou "nejprve schéma", a jakékoli změny lze provést okamžitě a neovlivní již uložené dokumenty. Můžeme například snadno přidat pole "reactions" nebo nahradit hodnotu pole "image" polem vnořených dokumentů.

3.2.12 Jazyk databáze a CRUD operace

Většina moderních relačních databází používá jazyk SQL. SQL je vysokoúrovňový databázový jazyk, který umožňuje definici dat, dotazování dat, manipulaci s daty, řízení dat a transakcí (8). **MongoDB není s jazykem SQL kompatibilní** kvůli své dokumentově orientované povaze (28).

MongoDB umožňuje pracovat s databázemi a jejich obsahem třemi způsoby (29):

- prostřednictvím uživatelských rozhraní, jako jsou MongoDB Compass a MongoDB Atlas;
- prostřednictvím oficiálních knihoven a ovladačů k podporovaným programovacím jazykům a frameworkům;
- prostřednictvím **mongo shell.**

Mongo shell je rozhraní příkazového řádku (command line interface), které je součástí standardní distribuce MongoDB. Používá příkazy se **syntaxí podobnou JavaScriptu** pro manipulaci s daty a administrativní operace (30).

„JavaScriptový shell databáze MongoDB umožňuje snadno si hrát s daty a získat představu o dokumentech, kolekcích a specifickém dotazovacím jazyce databáze.“ (16)

Níže jsou uvedeny příkazy pro operace CRUD (Create, Read, Update, Delete).

Operace vytváření a vkládání:

- use db_name;
- db.createCollection(name, [options]);
- db.collection_name.insertOne(doc);
- db.collection_name.insertMany(doc1, doc2, …).

Příkaz "use" slouží k přepínání mezi databázemi, pokud však vložíme jméno, které ještě není definováno, vytvoří se nová databáze. Stejným způsobem "insertOne" a "insertMany" implicitně vytvoří kolekci, pokud není definován daný název.

Operace pro čtení dokumentů z kolekce:

- db.collection_name.find({…}).

Metoda ".find()" přijímá argument, který se nazývá **selektor**. Selektor je predikát podobný dokumentu, který se používá k porovnání se všemi existujícími dokumenty kolekce.

Operace pro aktualizaci nebo výměnu dříve vytvořených dokumentů:

- db.collection_name.updateOne({…}, {$set : …});
- db.collection_name.updateMany({…}, {$set : …}));
- db.collection_name.replaceOne({…}, doc).

Metody aktualizace dostávají kromě selektoru ještě druhý argument. Má stejnou podobu jako dokument, ale obsahuje klíčové slovo "$set", za kterým je uvedeno cílové pole a nová hodnota. V případě operace nahrazení je uveden celý nový dokument. Starý bude po dokončení odstraněn.

Operace vymazání:

- db.collection_name.deleteOne({…});
- db.collection.deleteMany({…}).

Metody pro odstranění stejně jako metoda pro čtení přijímají jako argument selektor.

3.2.13 Přímé porovnání SQL a příkazů v MongoDB shellu

V této části jsou uvedeny ukázkové příkazy SQL (dialekt PostgreSQL) a ekvivalentní metody jazyka JavaScript pro MongoDB shell.

```
CREATE DATABASE example;
USE example;
```

Ukázka kódu 9 - (vlastní zpracování)

```
use example
```

Ukázka kódu 10 - (vlastní zpracování)

```
CREATE TABLE books(
  id SERIAL PRIMARY KEY,
  title VARCHAR(255),
  publication_year INT,
  author VARCHAR(255),
  pages INT,
  rating REAL,
  price REAL
);

INSERT INTO books (title, publication_year, author, pages, rating, price)
VALUES
  ("The Metamorphosis", 1915, "Franz Kafka", 44, 4.5, 3.99),
  ("Crime and Punishment", 1886, "Fyodor Dostoevsky", 430, 4.6, 6.99),
  ("Brave New World", 1932, "Aldous Huxley", 288, 4.44, 10.34);
```

Ukázka kódu 11 - (vlastní zpracování)

```
example.createCollection("books");

db.books.insertMany([
  {
  title: "The Metamorphosis",
```

```
    publication_year: 1915,
    author: "Franz Kafka",
    pages: 44,
    rating: 4.5,
    price: 3.99
    },
    {
    title: "Crime and Punishment",
    publication_year: 1886,
    author: "Fyodor Dostoevsky",
    pages: 430,
    rating: 4.6,
    price: 6.99
    },
    {
    title: "Brave New World",
    publication_year: 1932,
    author: "Aldous Huxley",
    pages: 288,
    rating: 4.44,
    price: 10.34
    }
  ]
)
```

Ukázka kódu 12 - (vlastní zpracování)

```
SELECT * FROM books;

SELECT * FROM books WHERE author = "Franz Kafka";

SELECT * FROM inventory WHERE price < 10 AND rating > 4.5;

SELECT title, author
FROM books
```

```
WHERE publication_year > 1900 OR pages > 100;
```

Ukázka kódu 13 - (vlastní zpracování)

```
db.books.find();


db.books.find({author : "Franz Kafka" });


db.books.find({ price: { $lt: 10}, rating: { $gt: 4.5 } });


db.books.find(
   { $or: [ { publication_year: { $gt: 1900 } }, { pages: { $gt: 100 } } ] },
   { title: 1, author: 1 }
);
```

Ukázka kódu 14 - (vlastní zpracování)

```
UPDATE books SET price = 5.99 WHERE title = "The Metamorphosis";
```

Ukázka kódu 15 - (vlastní zpracování)

```
db.books.updateMany({ title: "The Metamorphosis"}, { $set : { price : 5.99} })
```

Ukázka kódu 16 - (vlastní zpracování)

```
DELETE FROM books
WHERE author IN (
   SELECT TOP 1 author
   FROM books
   WHERE author = "Aldous Huxley"
);
```

Ukázka kódu 17 - (vlastní zpracování)

```
db.books.findOneAndDelete({}, { sort: { rating: 1 } })
```

Ukázka kódu 18 - (vlastní zpracování)

3.2.14 Transakce

Vlastnosti **ACID** jsou souborem čtyř charakteristik, které jsou klíčové pro spolehlivost transakcí v DBMS. Těmito vlastnostmi jsou atomicita (Atomicity), konzistence (Consistency), izolace (Isolation) a trvanlivost (Durability) (8).

- Princip atomicity znamená, že se s transakcemi pracuje jako s jedním celkem, nedělitelnou jednotkou práce. Buď se transakce provede od začátku až do úspěšného dokončení, nebo se data vrátí do původní podoby;
- Princip konzistence zajišťuje, že každá transakce přivede databázi z jednoho konzistentního stavu do druhého. To znamená, že po provedení transakce budou data nadále respektovat omezení stanovená systémem, jako například jedinečnost soukromého klíče;
- Princip izolace je nutný, pokud probíhá více transakcí současně. Tato vlastnost zajišťuje, že se transakce navzájem neruší, přičemž každá „porce“ dat je izolována od ostatních až do ukončení transakce;
- Princip trvanlivosti zajišťuje, že jakmile je transakce provedena, jsou její změny trvalé a přežijí jakékoli následné selhání systému (8).

Donedávna MongoDB plně neodpovídala principům ACID (31), což byla významná nevýhoda při srovnání s konkurenčními RDBMS na trhu.

„Podpora transakcí ACID s více dokumenty byla poprvé uvedena ve verzi MongoDB 4.0 v roce 2018 a v roce 2019 byla rozšířena verzí MongoDB 4.2, která umožňuje distribuované transakce ve sdílených clusterech.“ (31)

Všechny operace typu „write“ se zaměřují pouze na jednu kolekci a jsou atomické na úrovni každého jednotlivého dokumentu (32). MongoDB však nezaručuje obecnou atomicitu dotazů zaměřených na vícenásobné změny (obsahující metody insertMany(), updateMany(), deleteMany()). K zajištění bezpečnostních garancí pro tyto operace se používají repliky, distribuované dotazy a „Transactions API“ (31).

3.2.15 Příklady použití MongoDB

Podle oficiálních stránek MongoDB lze tuto databázi použít pro následující případy použití:

- Poskytování dat pro aplikace využívající umělou inteligenci (33);

- Vývoj mobilních aplikací (34);
- Správa internetu věcí (35);
- Vytváření a podpora dynamických katalogů (36);
- Správa dynamického a nestrukturovaného obsahu webových stránek (37);
- Logování (od anglického „logging“ – zaznamenávaní) dat (38);
- Agregace a analýza dat v reálném čase (39).

Mezi příklady společností, které integrovaly MongoDB do svých systémů, patří Bosch, Electrolux, Vodafone, Trade Ledger, KPMG (40).

3.3 Kompatibilní technologie

MongoDB je multifunkční databázový systém, ale k interakci a vytváření plně funkčních projektů na její bázi je nutná sada nástrojů a dodatečných technologií.

3.3.1 Linux Ubuntu

MongoDB je kompatibilní se všemi třemi hlavními desktopovými operačními systémy: MacOS, Windows a Linux (41).

Lídrem mezi severními operačními systémy je Linux. Ze všech webů, jejichž operační systém je znám, běží 84,6 % na unixových systémech, a toto číslo zahrnuje i všechny varianty Linuxu – 41,4 % z tohoto počtu (42).

Oficiálně MongoDB podporuje 5 rodin Linuxu (43):

- Amazon Linux;
- Debian;
- RHEL / CentOS;
- SLES;
- Ubuntu.

Nejpopulárnější z výše uvedených platforem je Ubuntu. Ze všech linuxových serverů na něm běží 25 % (44).

Ubuntu je řada distribucí Linuxu s otevřeným zdrojovým kódem. Byla založena na Debianu a je vyvíjena společností Canonical Ltd. Díky své modulární struktuře je flexibilní a vysoce přizpůsobitelná. Ubuntu využívá pro správu balíčků nástroj APT (Advanced Package Tool), který usnadňuje instalaci, aktualizaci a odebírání softwarových balíčků prostřednictvím příkazového řádku (45).

3.3.2 Python 3

Kromě možnosti interakce prostřednictvím shellu mongosh podporuje MongoDB řadu oficiálních knihoven a rozšíření pro různé programovací jazyky a frameworky. V současné době jich je 13 (46):

- C;
- C++;
- C#;
- Go;
- Java;
- Kotlin;
- Node.js;
- PHP;
- Python;
- Ruby;
- Rust;
- Scala;
- Swift.

Python 3 je třetí verze vysokoúrovňového programovacího jazyka s otevřeným zdrojovým kódem napsaném v jazyce C. On má dynamickou typování (typing), jednoduchou syntaxi a je snadno čitelný pro lidi. Díky modulární struktuře a podpoře balíčků a knihoven je univerzální a také snižuje cenu vývoje a podpory psaného kódu (47) (48).

PyMongo je oficiální ovladač MongoDB pro Python 3. Tento ovladač je určen pro synchronní pythoní kód. Aktuální verze 4.6 je kompatibilní s verzí Python 3.11 a instaluje

se jako knihovna pomocí standardního správce balíčků „pip“. PyMongo obsahuje sadu tříd a metod pro připojení k databázovému serveru, provádění příkazů a dotazů. Tato knihovna také automaticky překládá standardní datovou strukturu Pythonu "dictionary" (slovník) do JSON-podobných řetězců a také transformuje dokumenty BSON a vrací je ve formě slovníku nebo seznamu slovníků (49)

4 Vlastní práce

Tato část práce se zaměřuje na praktické použití MongoDB.

4.1 Analýza

Cílem této podkapitoly je vybrat a popsat případovou studii, která demonstruje výhody datového modelu MongoDB.

4.1.1 Výběr případu použití

Podle seznamu případů použití MongoDB popsaného v kapitole 3.2.15 teoretické části této práce lze tuto databázi použít k organizaci úložiště logů.

„Logování dat znamená systematické zaznamenávání událostí, pozorování nebo měření. Zařízení, které se podílí na záznamu dat, se nazývá loggerem dat.“ (50)

Argumenty ve prospěch použití tohoto systému právě pro logování jsou vysoká rychlost zápisu dat, různorodé kolekce a samotný datový model orientovaný na dokumenty. Absence schématu umožňuje měnit typ dat a počet polí v závislosti na potřebách (38).

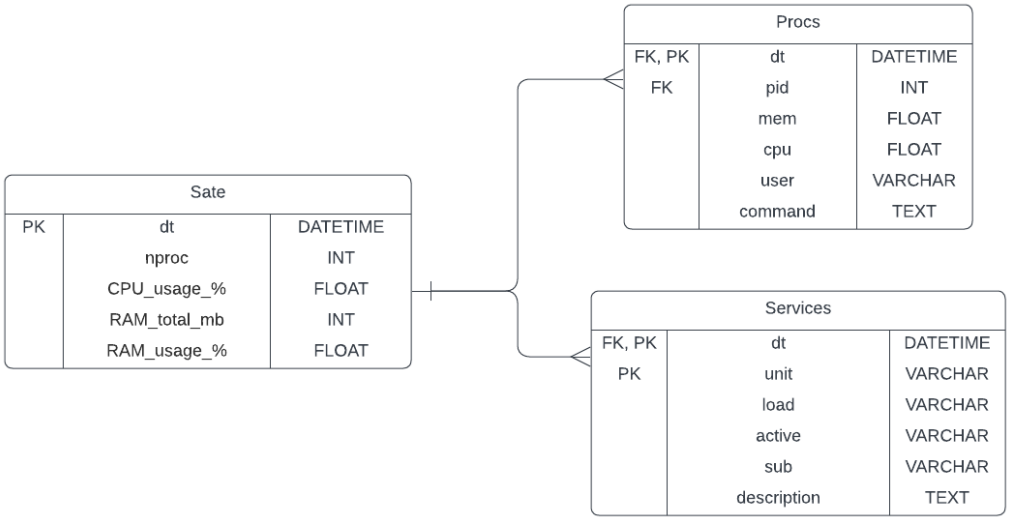
Tento případ užití byl vybrán pro implementaci v následujících částech práce, protože je v porovnání s ostatními případy užití relativně jednoduchý, ale zároveň je schopen explicitně demonstrovat výhody datového modelu MongoDB.

4.1.2 Popis problému

Velké společnosti a korporace, jako je např. Commerzbank, spravují vlastní servery pro provoz své infrastruktury. Každý server, stejně jako programy na něm nainstalované, generuje textová data o svém aktuálním stavu. Výhodou těchto dat je, že umožňují přesnější správu systému a také identifikaci příčin poruch. Vzhledem k velkému množství a polostrukturované povaze dat jsou však některá data ukládána do textových dokumentů a archivována pod názvem, který odráží název serveru a čas záznamu. To pomáhá šetřit místo na paměťovém médiu, ale v případě potřeby analýzy musí správce systému ručně vyhledat příslušný archiv (nebo skupinu archivů), rozbalit jej a až poté začít vyhledávat data v dokumentu.

4.1.3 Složitosti relačního přístupu

Problém popsaný v předchozí podkapitole je možné řešit pomocí relačních databází. V tomto případě by prvním krokem bylo vytvoření relačního schématu.



Obrázek 6 – ER diagram ukládání logů (vlastní zpracování)

Ve výše uvedeném diagramu (viz obrázek 6) jsou data generovaná serverem rozdělena do 3 entit. Zatímco takové ukazatele, jako například zatížení procesoru, jsou popsány číselnou hodnotou, počet aktivních procesů a služeb se může lišit, proto jsou umístěny v samostatných tabulkách a propojeny s tabulkou "State“ (stav) vztahem „1-to-many“ na základě atributu času. I přes funkčnost tohoto přístupu obsahuje několik nevýhod. V určitém časovém okamžiku může být v systému několik desítek až několik tisíc aktivních procesů a běžících služeb. Pokud správce potřebuje získat kompletní přehled o tom, jak systém vypadal v určitém okamžiku, bude muset tabulky spojit pomocí operátoru JOIN. Pro urychlení operace spojování můžeme indexovat atribut "dt", ale s rostoucím množstvím uložených dat se bude proces zápisu nových dat zpomalovat. Toto znázornění ukazuje schéma ukládání logů vztahujících se k jednomu serveru. Chceme-li ukládat logy více strojů, můžeme pro každý z nich vytvořit samostatnou databázi nebo do každé tabulky přidat další atribut, který servery jednoznačně identifikuje. Druhá možnost dále zkomplikuje operaci spojování tabulek. Pokud bude nutné v budoucnu sbírat více dat, bude nutné schéma ukládání zcela přepracovat, aby se počítalo s novými atributy, tabulkami a propojeními.

Na základě těchto faktorů lze dospět k závěru, že řešení využívající relační databáze není optimální, protože může způsobit problémy při dlouhodobém používání.

4.1.4 Formulace podmínek pro potenciální řešení

Hlavní úkol při řešení problému logování spočívá ve vytvoření flexibilního systému, který lze snadno konfigurovat. Navrhovaný systém by měl pravidelně ukládat informace o stavu serveru a jeho aplikací ve formátu optimálním pro skladování a vyhledávání. Je také nutné vzít v úvahu skutečnost, že objem a struktura dat se mohou v průběhu času měnit.

Další podmínkou je používání softwaru s otevřeným zdrojovým kódem nebo od důvěryhodných vydavatelů v co největší míře. Tato podmínka je nezbytná pro dodržení digitálních bezpečnostních opatření při práci s kritickou infrastrukturou.

4.2 Výběr nástrojů

V této kapitole budou popsány technologie a nástroje pro realizaci projektu z hlediska klíčových charakteristik, které ovlivnily výběr.

4.2.1 Hosting a operační systém

K simulaci práce sady serverů bude použit **VirtualBox** od společnosti Oracle. VirtualBox je bezplatný nástroj, který umožňuje vytvářet plnohodnotné virtuální počítače. To znamená, že každý virtuální stroj má vlastní simulaci hardwaru a operační systém na něm nainstalovaný bude izolován od hostitele a ostatních virtuálních strojů.

Oba typy, a server ukládající logy, a monitorované servery budou používat jako operační systém **Linux Lubuntu**. Lubuntu je jednou z variant Ubuntu (popsán v kapitole 3.3.1), kde písmeno "L" je zkratka pro „lightweight” (lehký). Je založen na stejné platformě, ale je omezen ve funkčnosti, obsahuje pouze nejnutnější balíčky a je schopen běžet i na systémech s jedním 64bitovým jádrem a 512 megabajty paměti RAM (51).

4.2.2 Backend

Podle zadání bude jako technologie ukládání dat použita MongoDB. **MongoDB Community Edition** je bezplatná verze databázového serveru, která podporuje až 500 souběžných připojení.

Pro přístup k databáze a jako hlavní backendový programovací jazyk bude zvolen **Python 3**. Pro tuto volbu hovoří dva argumenty. První je, že v současné době jsou aktuální verze Ubuntu vybaveny interpretem CPython a podporují verzi 3.11.2 bez nutnosti instalace dalších balíčků (52). Druhým argumentem je, že Python má přehlednou syntaxi. Díky tomu kódu porozumí i administrátoři systému bez znalosti tohoto konkrétního jazyka. Jako adaptér mezi Pythonem a MongoDB bude použita oficiální knihovna **PyMongo 4.6**.

Podle studie ekosystému Pythonu z roku 2022 společnosti JetBrains je nejoblíbenějším frameworkem pro vývoj aplikací v jazyce Python Flask. Používá ho 40 % webových aplikací v jazyce Python (53). Ale, kromě standardní knihovny vyžaduje Flask další balíčky, například Werkzeug a Jinja2 (54).

Flask má odlehčenou alternativu -- **Bottle**. Framework Bottle má podobný systém směrování, šablony a je také kompatibilní s WSGI, ale nemá žádné závislosti nad rámec standardní knihovny. Díky tomu je vhodnější pro aplikace malého rozsahu (55).

„WSGI je rozhraní Web Server Gateway Interface. Jedná se o specifikaci, která popisuje, jak webový server komunikuje s webovými aplikacemi a jak lze webové aplikace řetězit za účelem zpracování jednoho požadavku. WSGI je standard jazyka Python podrobně popsaný v dokumentu PEP 3333.“ (56)

„Bottle je rychlý, jednoduchý a odlehčený webový mikroframework WSGI pro Python. Je distribuován jako modul v jednom souboru a nemá žádné jiné závislosti než standardní knihovnu Pythonu.“ (57)

Kromě knihoven Bottle a PyMongo bude použita ještě jedna knihovna, která vyžaduje další instalaci. Pro vizualizaci dat a renderování obrázků bude použita knihovna **Matplotlib**. Matplotlib je knihovna s otevřeným zdrojovým kódem, která se objevila již v roce 2003. Ona je považována za spolehlivý nástroj a používá se mimo jiné pro vědecké projekty (58).

4.2.3 Frontend

Protože vizualizace dat je výpočetně náročná úloha, bude se provádět na straně backendu. Úkolem frontendu bude pouze zobrazovat statické stránky s odpovídajícími, předem vygenerovanými daty. Pro tento účel je vhodný formát **TPL**. TPL dynamicky generuje stránky HTML na základě předem definovaných šablon a skriptů, a díky tomu lze na stránky aplikovat styly **CSS**.

4.2.4 Web-server

Apache HTTP Server je projekt webového serveru s otevřeným zdrojovým kódem, který vznikl v roce 1995. Novější verze Apache 2 je kompatibilní se všemi nejpoužívanějšími operačními systémy včetně Linuxu. Díky své modulární architektuře je vysoce konfigurovatelný (59). Apache umožňuje hostovat aplikace Python pomocí balíčku **wsgi_mod** (60).

4.3 Návrh řešení

4.3.1 Organizace úložiště

Dokumentově orientovaný model MongoDB umožní ukládat všechna data v jednom "balíčku", přičemž každý proces a služba budou uloženy jako samostatný objekt v poli rodičovského objektu. Tímto způsobem se vyhneme nutnosti rozdělovat data mezi kolekcemi. V rámci jedné kolekce se struktura dokumentů bude lišit pouze počtem objektů v polích. Data každého serveru budeme moct ukládat do samostatné kolekce, a to bez překročení hranic jedné databáze.

„Pokud mají všechny dokumenty v kolekci podobnou, ale ne totožnou strukturu, nazýváme ji polymorfní vzor. Jak již bylo řečeno, polymorfní vzor je užitečný, když chceme přistupovat k informacím (dotazovat se na ně) z jedné kolekce. Seskupení dokumentů dohromady na základě dotazů, které chceme provádět (namísto rozdělení objektu do tabulek nebo kolekcí), pomáhá zlepšit výkon.“ (61)

Níže je uveden příklad, jak může dokument vypadat při použití polymorfního návrhového vzoru (viz obrázek 7).

```
{
  "_id": {...},
  "timestamp": {...},
  "nproc": 1,
  "CPU_usage_%": 33.1,
  "RAM_total_mb": 998,
  "RAM_usage_%": 74.5,
  "procs": [...],
  "services": [...]
}
```

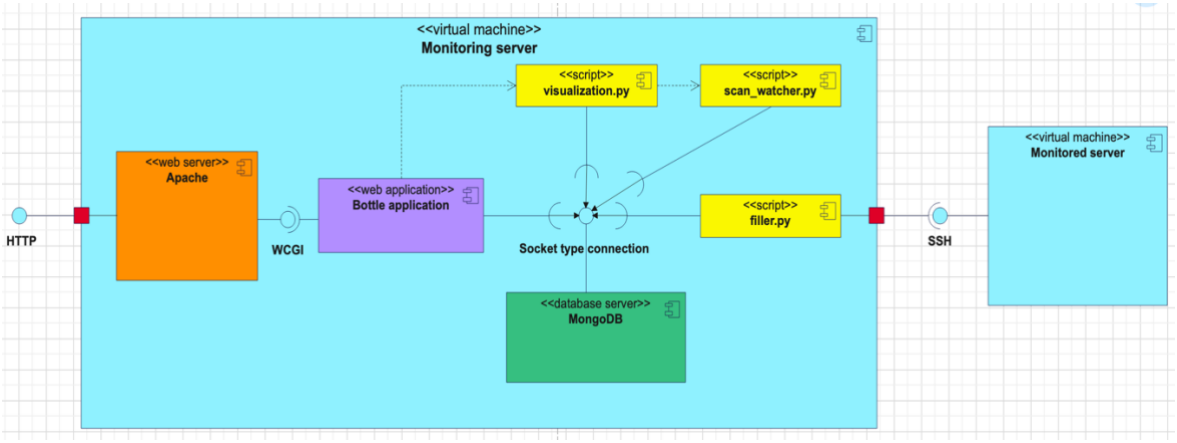

Obrázek 7 - Struktura dokumentu obsahujícího informace o serveru (vlastní zpracování)

Důležitou součástí problému je, že "snímky" stavu systému budou identifikovány podle času záznamu. Toto pole bude také základem pro vyhledávání. Optimalizovat úlohu vyhledávání pomůže speciální typ kolekcí MongoDB – **time series kolekce** (kolekce časových řad).

„V porovnání s běžnými kolekcemi ukládání dat časových řad do time series kolekcí zlepšuje efektivitu dotazů a snižuje využití disku pro časová data a sekundární indexy.“

(18)

4.3.2 Návrh struktury systému



Obrázek 8 - UML diagram komponent (vlastní zpracování)

Výše uvedený obrázek ukazuje princip interakce mezi jednotlivými komponentami systému logování.

Komunikace mezi hlavním serverem a sledovaným serverem bude realizována pomocí připojení SSH prostřednictvím vnitřní virtuální sítě. Komponentou provádějící spojení bude

skript pro sběr dat – "filler script". Stejný skript bude provádět parsing textových dat, převádět je do pythoního slovníku a ukládat je do databáze.

K databázi budou připojeny i další dvě komponenty, ale pouze pro čtení. Skript pro vizualizaci dat bude číst poslední uložená data s určitou periodicitou a generovat grafy ve formátu .PNG. Rozhodnutí oddělit vizualizaci od hlavní aplikace je vysvětleno tím, že pokud se data ze serveru sbírají jednou za minutu, nashromáždí se během 12 hodin práce 720 dokumentů. Vykreslení grafu obsahujícího takové množství značek může trvat až 5 sekund, což negativně ovlivní rychlost aplikace. Pro zvýšení rychlosti aplikace bude skript pro vizualizaci umístěn do samostatné komponenty, která pracuje periodicky, nikoli na základě požadavku.

Aplikace Bottle bude také mít přístup k databázi, aby mohla zobrazit seznam kontrolovaných serverů a vyhledat snapshoty na základě zadaných časových údajů. Pokud jde o zobrazování grafů se zaznamenanými metrikami, závisí tato funkce na práci vizualizační komponenty. Bottle vykreslí stránku s posledním statickým obrázkem vygenerovaným skriptem.

Kromě databázového serveru bude na centrálním virtuálním počítači nainstalován webový server Apache a balíček mod_wsgi, aby byla zajištěna kompatibilita s aplikací Bottle. Tato kombinace technologií umožní nejen přístup k aplikaci prostřednictvím protokolu HTTP, ale také spuštění instancí Bottle() ve více vláknech.

4.4 Implementace

4.4.1 Příprava sítě virtuálních počítačů

Velikost virtuální struktury a výkonnost serverů závisí na množství zdrojů, které jsou jim v hostitelském stroji přiděleny. Vzhledem k tomu, že prostředky pro vytvoření a testování projektu jsou omezeny hardwarem autorova osobního počítače, bylo přiděleno o něco více paměti a výpočetních jader, než je minimální požadovaný objem. Po stažení obrazu lubuntu-23.10 z oficiální stránky byly vytvořeny 2 virtuální počítače.

Centrální server “s0”:

- 4096 MB RAM
- 4 logická jádra
- 50 GB diskového prostoru

Sledovaný server “s1”:

- 1024 MB RAM
- 1 logické jádro
- 10 GB diskového prostoru

Dále byl server s1 naklonován dvakrát. Nové stroje se stejnými parametry byly pojmenovány s2 a s3.

V dalším kroku, pomocí grafického rozhraní VirtualBoxu byla vytvořena virtuální síť:

- Typ – Host Network
- Dolní hranice IP adresy – 192.168.56.1 (adresa hostitelského počítače)
- Horní hranice adresy IP – 192.168.56.99
- Maska – 255.255.255.0

Dále byl ke existujícímu adaptéru NAT přidán nový virtuální adaptér pro každý server, který bude pracovat s lokální sítí.

Poté byl postupně zapnut každý virtuální počítač, vytvořen technický uživatel xmori001 a nakonfigurováno nastavení sítě. Příkaz "**sudo nano /etc/hostname**" otevřel soubor a stroje byly pojmenovány odpovídajícím způsobem (s0-s3).

V souladu s předchozím krokem byl soubor odpovědný za párování jmen a síťových adres otevřen příkazem "**sudo nano /etc/hosts**" a upraven následujícím způsobem (viz ukázka kódu 19):

```
# Standard host addresses
127.0.0.1  localhost
::1        localhost ip6-localhost ip6-loopback
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
# This host address
127.0.1.1  s0
```

Ukázka kódu 19 – soubor /etc/hosts (62)

Poslední řádek se liší podle názvu stroje.

S vědomím, že kódový název druhého síťového adaptéru ve všech systémech Ubuntu je "**enp0s8**", byl pomocí příkazu "**sudo nano /etc/netplan/01-network-manager-all.yaml**" otevřen konfigurační soubor sítě a upraven jak je uvedeno níže (viz ukázka kódu 20).

```
# Let NetworkManager manage all devices on this system
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    enp0s8:
      dhcp4: no
      addresses: [192.168.56.10/24]
```

Ukázka kódu 20 - netplan konfigurace (63)

V posledním řádku byly nastaveny adresy 192.168.56.10-13 tak, aby odpovídaly serverům s0-s3. Aby se nová konfigurace uplatnila, byl proveden příkaz "**sudo netplan apply**" a síťová služba byla restartována příkazem " **sudo systemctl restart NetworkManager**".

Dalším krokem bylo nakonfigurování připojení SSH z centrálního serveru na zkoumané servery. Pomocí správce balíčků byl nainstalován nástroj implementující tento protokol: "**sudo apt-get install openssh-server**". Na každém serveru byl příkazem "**ssh-keygen -t rsa -b 4096 -C "xmori001[at (@)]studenti.czu.cz"**" vytvořen pár klíčů sestávající z veřejného a soukromého klíče. Při navazování šifrovaného spojení mezi dvojicemi serverů

se klíče vyměňují pomocí algoritmu RSA. Pro zjednodušení vývoje programů pomocí připojení SSH byl vytvořen bezheslový přístup z centrálního serveru na studované servery pomocí příkazu "**ssh-copy-id xmori001@192.168.56.11**" (adresa se liší v závislosti na serveru) a jednorázového zadání hesla technickému uživateli na příslušném virtuálním stroji.

4.4.2 Příprava databáze

Jakmile byla vytvořena a nakonfigurována síť virtuálních počítačů, byly na S0 nainstalovány všechny potřebné programy a balíčky pro implementaci systému logování.
Nejprve byl nainstalován databázový server MongoDB.

Vzhledem k tomu, že oficiální balíčky pro instalaci MongoDB Community Edition jsou obsaženy v soukromém úložišti, byla k jejich instalaci pomocí APT nutná následující (viz ukázka kódu 21) sekvence příkazů:

```
curl -fsSL https://www.mongodb.org/static/pgp/server-7.0.asc | \
  sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg \
 --dearmor

echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg ] https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-7.0.list

sudo apt-get update

sudo apt-get install -y mongodb-org
```

Ukázka kódu 21- aktualizace odkazů na repozitáře a instalace MongoDB pomocí správce balíčků apt (64)

Ihned po instalaci, před prvním spuštěním, byl pomocí příkazu "**sudo nano /etc/mongod.conf**" otevřen soubor YAML konfigurace mongoDB a upraven tak, aby sekce parametrů replikace obsahovala následující kód (viz ukázka kódu 22):

```
#replication:
replication:
  replSetName: rs0
```

Ukázka kódu 22 - mongod.conf replika set nastaveni (vlastní zpracování)

Převedením místního databázového serveru do formátu sady replik s jedním uzlem byla povolena funkce sledování proudu transakcí. Tato funkce bude užitečná při implementaci skriptů doplňujících aplikaci.

Konfigurace MongodB byla dokončena po prvním spuštění. Za tímto účelem byl primární proces mongod spuštěn příkazem "**sudo systemctl start mongod**". Poté byl spuštěn standardní shell **mongosh** a v něm příkaz "**rs.initiate()**".

Dalším krokem bylo vytvoření databáze a kolekcí pro ukládání logů. Programy používající oficiální ovladače MongoDB jsou schopny implicitně vytvářet databáze a standardní kolekce, v našem případě však pro ukládání logů byly vybrány kolekce typu "časové řady". Pro zjednodušení skriptu, který plní úložiště protokolů, bylo rozhodnuto, že sbírky budou vytvořeny předběžně a ručně (viz ukázka kódu 23).

```
use logs
db.createCollection("s1", { timeseries: { timeField: "timestamp" } });
db.createCollection("s2", { timeseries: { timeField: "timestamp" } });
db.createCollection("s3", { timeseries: { timeField: "timestamp" } })
```

Ukázka kódu 23 - tvorba kolekcí (vlastní zpracování)

Aby systém ukládání logů věděl, se kterými servery má komunikovat a se kterými ne, byla vytvořena samostatná databáze "info" (viz ukázka kódu 24). Kolekce "monitored_servers" z této databáze obsahuje dokumenty, které zahrnují všechny potřebné údaje pro identifikaci a připojení k vybraným serverům. Později bude do "info" přidána další kolekce "scans", která bude uchovávat informace o úspěšnosti sběru dat.

```
use info
db.monitored_servers.insertMany(
  [
    {
      name: "s1",
      ip: "192.168.56.11",
      tech_user: "xmori001"
    },
    {
      name: "s2",
      ip: "192.168.56.12",
      tech_user: "xmori001"
    },
    {
      name: "s3",
      ip: "192.168.56.13",
      tech_user: "xmori001"
    }
  ]
)
```

Ukázka kódu 24 - vytváření kolekce "monitored_servers" (vlastní zpracování)

4.4.3 Instalace webového serveru

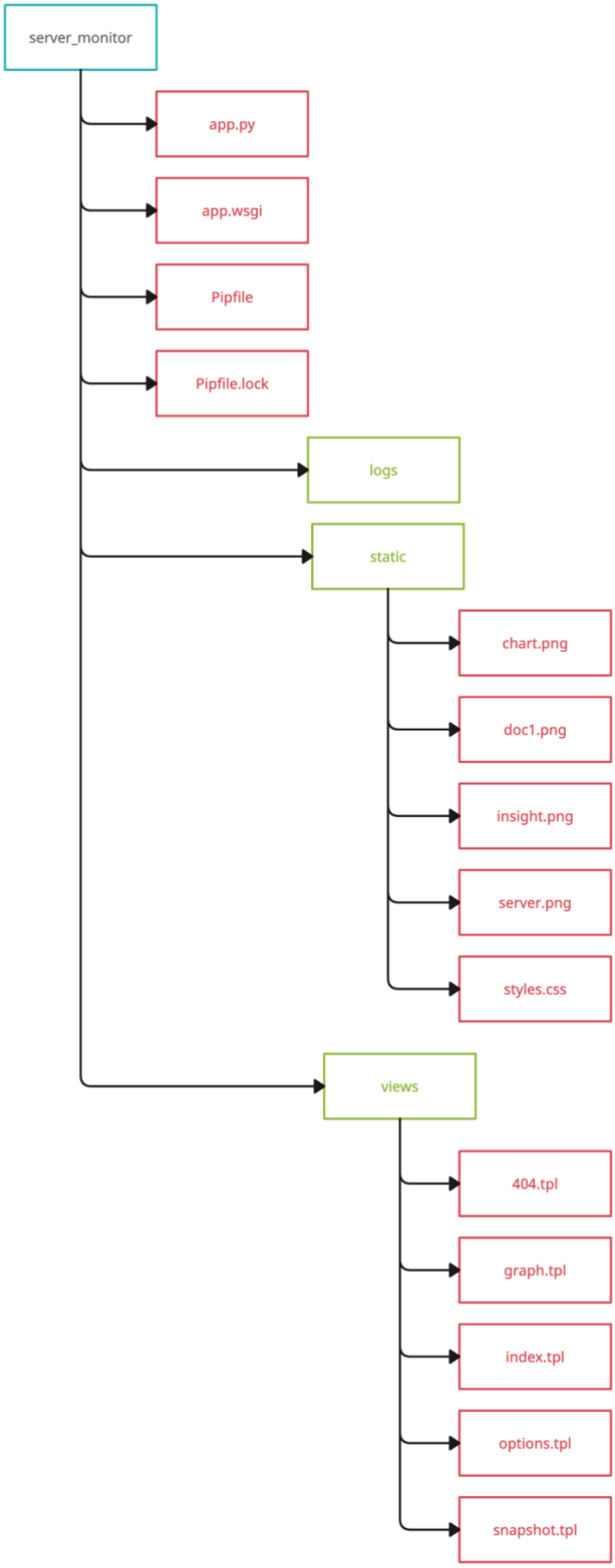
Aby se prototyp systému sbírajícího a zobrazujícího logy přiblížil skutečnému modelu, byl nainstalován webový server Apache a modul mod_wsgi umožňující interakci s aplikacemi Python. Tento krok byl proveden pomocí apt "**sudo apt install apache2 libapache2-mod-wsgi-py3**" bez jakýchkoli dalších akcí. Je důležité, že při instalaci webového serveru Apache byl vytvořen adresář "**/var/www**", ve kterém se doporučuje vytvořit strukturu webové aplikace. V operačním systému byla také vytvořena skupina **www-data**.

Konfigurace a spuštění webového serveru se provede po vytvoření struktury aplikace a vložení kódu a dalších materiálů.

4.4.4 Vytvoření struktury aplikace a virtuálního prostředí

Po instalaci webového serveru byla vytvořena struktura aplikace, jejíž cesta se nachází ve složce "/var/www/server_monitor". Soubor "app.py" bude obsahovat kód se základní logikou aplikace. Soubor "app.wsgi" bude skript zajišťující interakci mezi aplikací a webovým serverem. "logs" je prázdná složka, ve které Apache vytvoří soubory "access.log" a "error.log" pro zaznamenávání připojení a chyb serveru. Adresář "views" bude obsahovat 5 souborů s příponou ".tpl". Tyto soubory budou šablonami pro generování stránek HTML. Sousední adresář "views" bude obsahovat ikony pro výzdobu stránek aplikace a také grafy generované na základě získaných dat.

Po vytvoření adresáře s aplikací je třeba vytvořit virtuální prostředí Pythonu. K tomuto účelu byl použit příkaz "**sudo apt install pipenv**", který spustí instalaci balíčku pro příslušnou jazykovou verzi. Zatímco v adresáři "server_monitor" příkaz "**sudo pipenv install**" vytvořil virtuální prostředí přidružené k tomuto projektu. Dále byl použit příkaz "**pipenv shell**" k aktivaci virtuálního prostředí, které umožňuje přístup ke správci balíčků. Pomocí příkazů "**pipenv install pymongo**" a "**pipenv install bottle**" byly staženy a nainstalovány knihovny třetích stran potřebné pro aplikaci. Stejnými akcemi byly vytvořeny soubory "Pipfile" a "Pipfile.lock", které uchovávají informace o knihovnách použitých v projektu a jejich verzích (soubory samotného virtuálního prostředí byly vytvořeny v adresáři "/home/xmori001/.local/share/virtualenvs/server_monitor-fBoou3Ne").
Před prvním spuštěním aplikace a skriptů pro sběr a vizualizaci dat by tedy struktura projektu měla odpovídat níže uvedenému schématu (viz obrázek 9).



Obrázek 9 - struktura adresářů aplikací (vlastní zpracování)

4.4.5 Kód aplikace

Hlavní logika aplikace je v kódu souboru "app.py" (viz příloha 1). Na začátku kódu jsou importovány všechny potřebné knihovny, třídy a metody. V konstantní proměnné je uloženo aktuální umístění souboru, který bude výchozím bodem pro strukturu aplikace. Podle specifikace WSGI bude aplikace spuštěna na základě instance třídy Bottle. Na začátku se také vytvoří spojení s lokálním databázovým serverem. Dále se vytvoří funkce pro aktualizaci sledovaných serverů. Dále je tato funkce často volána pro kontrolu neočekávaných manipulací v poli adresy URL.

Dále následuje 6 funkcí zabalených do dekorátorů "@app.route". Vytvářejí logiku přechodů mezi čtyřmi typy stránek aplikace. Hlavní stránka s adresou "/" zobrazuje seznam dostupných logů. Výběrem jednoho ze serverů se uživatel dostane na stránku možností "/options/<jméno>". Zde má uživatel na výběr ze dvou možností. První volba uživatele přenese na stránku "/graph/<jméno>", kde si lze vyžádat graf na základě vybraných metrik a časového intervalu. Druhá volba přenese uživatele na stránku "/snapshot/<jméno>". Zde bude mít uživatel možnost vyhledat soubor logu popisující server v určitém časovém okamžiku. Protože poslední 2 stránky jsou určeny ke čtení dat zadaných uživatelem, byly pro ně napsány 2 další funkce implementující metodu "POST".

Poslední 2 funkce obalené dekorátory "@app.route('/static/<filename:path>')" a "@app.error(404)" zajišťují vracení statických souborů ze složky "static" a ošetření chyb přístupu.

4.4.6 WSGI skript

Soubor "app.wsgi" obsahuje kód (viz obrázek 10), který umožňuje aplikaci komunikovat s modulem "mod_wsgi". I když má soubor příponu .wsgi, jedná se o kód interpretovaný v jazyce Python. Jeho smyslem je importovat vytvořenou instanci aplikace a aktivovat virtuální prostředí.

```
import sys
sys.path.insert(0, '/var/www/server_monitor')

activate_this = '/home/xmori001/.local/share/virtualenvs/server_monitor-
fBoou3Ne/bin/activate_this.py'
with open(activate_this) as file_:
    exec(file_.read(), dict(__file__=activate_this))

from app import app as application
```

Obrázek 10 - WSGI script (vlastní zpracování)

4.4.7 Šablony a stylování

K vytvoření standardní struktury pro většinu webových stránek se ve "styles.css" (viz příloha 2) používá modul rozvržení "grid". Na hlavních stránkách tak byly vyčleněny 3 hlavní sekce: záhlaví, střed a zápatí. Na domovské stránce, stejně jako na stránce s možnostmi, byly volby navrženy ve formě pohyblivých karet. K realizaci tohoto efektu byl použit modul rozvržení "flexbox". Další vlastnosti souboru stylů se týkají úpravy velikosti obrázků, velikosti písma a barev prvků.

Šablona "index.tpl" (viz příloha 3) obdrží jako argument slovník serverů vygenerovaný funkcí "get_servers_dict()" v backendu. Klíči jsou názvy serverů a hodnotami jejich IP adresy nebo textové oznámení, že server již není sledován. Smyčka “for” vnořená do šablony prochází slovník a vytváří kartu pro každou dvojici klíč-hodnota.

Šablona "options.tpl" (viz příloha 4) neobsahuje dynamicky generované prvky. Do této stránky je integrována 1 proměnná – název serveru vybraného na předchozí stránce. Tato proměnná slouží k vytvoření odkazů na následující stránky.

Šablona "graph.tpl" (viz příloha 5) obsahuje tag "form", který implementuje metodu "POST". Informace o metrikách a časovém období vybrané pomocí prvků "radio button" se tak budou odesílat do backendu pomocí protokolu HTTP. Konec této šablony obsahuje kontrolu logické podmínky. Pokud uživatel provede volbu, pak se stránka překreslí a pod

tlačítkem obnovení se zobrazí obrázek s příslušným grafem o velikosti 640 × 480 pixelů. V opačném případě je toto místo obsazeno neviditelným blokem stejné velikosti.

Šablona "snapshot.tpl" (viz příloha 6) obsahuje také tag "form", ale zobrazuje se ve formě polí, do kterých musí uživatel zadat číselné hodnoty. Tyto údaje jsou odeslány do backendu a v případě úspěšné validace je stránka znovu vykreslena. Šablona obdrží jako argument dokument z databáze rozdělený na 3 části a transformovaný do textového tvaru. Na stránce se zobrazí datum vybrané uživatelem a 3 části -- obecné informace, aktivní procesy a aktivní systémové služby.

Šablona "404.tpl" (viz příloha 7) je nejminimalističtější. Obsahuje pouze logo převzaté z hlavičky a oznámení, že požadovaný zdroj nebyl nalezen.

4.4.8 Konfigurace Apache

Po vytvoření struktury aplikace Bottle a jejím naplnění soubory se pokračovalo v konfiguraci webového serveru. Pro změnu vlastníka všech souborů a podadresářů byl proveden příkaz "**chown -R www-data:www-data /var/www**" a poté byl do této skupiny přidán aktuální technický uživatel: "**sudo usermod -a -G www-data xmori001**".

Poté byl v adresáři "/etc/apache2/sites-available" vytvořen soubor "server_monitor.conf" s následujícím obsahem (viz ukázka kódu 25).

```
<VirtualHost *:80>
    Servername 192.168.56.10

    WSGIDaemonProcess server_monitor user=www-data group=www-data threads=20
    WSGIScriptAlias / /var/www/server_monitor/app.wsgi

    <Directory /var/www/server_monitor>
        WSGIProcessGroup server_monitor
        WSGIApplicationGroup %{GLOBAL}
        Options ExecCGI Includes
        AllowOverride None
        Require all granted
    </Directory>

    <Directory /var/www/server_monitor/static/>
        Order allow,deny
        Allow from all
    </Directory>

    <Directory /var/www/server_monitor/views/>
        Order allow,deny
        Allow from all
    </Directory>

    ErrorLog /var/www/server_monitor/logs/error.log
    CustomLog /var/www/server_monitor/logs/access.log combined

</VirtualHost>
```

Ukázka kódu 25 - Apache server_monitor.conf (vlastní zpracování)

Ihned po vytvoření konfiguračního souboru byl spuštěn příkaz "**chmod 755 server_monitor.conf**", který přidal možnost spouštět tento soubor pro všechny uživatele. Aby se nová konfigurace uplatnila, byl spuštěn příkaz "**sudo a2ensite server_monitor.conf**".

4.4.9 Vytvoření virtuálního prostředí pro skripty

Kromě kódu napsaného ve frameworku Bottle obsahuje backend tohoto projektu další 2 skripty. Účelem prvního skriptu je pravidelně skenovat servery a plnit databázi. Účelem druhého skriptu je načítat nejnovější záznamy z databáze a vykreslovat grafy ve formátu

.png. Oba skripty vyžadují knihovnu PyMongo a pro druhý skript navíc Matplotlib. Pro tento účel bylo rozhodnuto vytvořit další lokální virtuální prostředí.

Mimo strukturu aplikace byla v adresáři "/home/xmori001" vytvořena složka "db_scripts", do které byly později umístěny skripty. Zatímco v této složce bylo vytvořeno virtuální prostředí příkazem "**python3 -m venv local_env**" a aktivováno příkazem "**source local_env/bin/activate**". V novém prostředí byly pomocí správce balíčků pip nainstalovány 2 požadované knihovny: "**pip install pymongo matplotlib**".

4.4.10 Naplnění databáze

Databáze se naplní spuštěním skriptu "filler.py" (viz příloha 8). Pro pochopení principu fungování tohoto skriptu je nutné ho číst od řádku "if __name__ == '__main__'". Po jeho spuštění se vytvoří spojení s lokálním databázovým serverem a z databáze "info" se vybere kolekce "monitored_servers". Pro každý dokument z této kolekce je v samostatném procesu zavolána funkce proc_logic. To znamená, že v našem konkrétním případě bude práce se servery s1, s2 a s3 probíhat ve třech samostatných nezávislých procesech. Ihned po vytvoření procesu se ve funkci vytvoří instance třídy „DataCollector“. Smysl této třídy spočívá v tom, že se vytvoří podproces, připojí se k cílovému serveru prostřednictvím SSH a provede řadu příkazů v jazyce bash.

- příkaz "**ps --no-headers aux | grep -v -E \"ps\"**“ vypíše běžící procesy.
- příkaz "**systemctl --state=active --quiet**" vypíše aktivní systémové služby.
- příkaz "**nproc**" vrátí počet výpočetních jader
- příkaz "**vmstat 1 2 | tail -1**" vypíše údaje pro výpočet zatížení procesoru
- příkaz „**free --mega | awk \"NR==2\"**“ vypíše řádek s údaji o systémové paměti.

Všechny tyto příkazy se provedou jeden po druhém a oddělí se příkazem "**echo -e \"---\n\"**". To umožňuje snadněji oddělit a zpracovat textová data, která se dostala na standardní výstup podprocesu. Statické metody pak provádějí parsování, převod na číselné hodnoty, výpočty a tvorbu slovníku. Všechna tato data jsou uložena v lokálních atributech a poté použita k vytvoření kompletního snímku serveru. Dále každý proces vytvoří nezávislé připojení k databázi a provede operaci vložení.

Nadřazený proces počká na dokončení všech procesů a poté zkontroluje jejich návratové kódy. Pokud během provádění nedošlo k žádné chybě, vloží se záznam o úspěšném dokončení do kolekce "scans" databáze "info".

4.4.11 Vizualizace dat

Vykreslování grafů ve formátu .png se provádí ve skriptu "visualization.py" (viz příloha 9). Tento skript stejně jako předchozí načte z databáze servery, které jsou monitorovány a pro každý z nich vytvoří samostatný proces.

Každý podřízený proces vytvoří nezávislé připojení k MongoDB a vybere kolekci s názvem odpovídajícím názvu serveru. Dále se vyberou dokumenty zadané za posledních 12 hodin. Po iteraci vráceného seznamu dokumentů se vytvoří nové seznamy pro uložení extrahovaných hodnot. V našem konkrétním případě použití jsou tyto hodnoty čas, využití procesoru a využití paměti RAM.

Na začátku tohoto kódu skriptu je importována knihovna "matplotlib" a pomocí této knihovny je definována vlastní funkce "render_graph". Tato funkce je volána 6krát v každém procesu. Takto se vytvoří a uloží grafy využití CPU a RAM za 1 hodinu, 5 hodin a 12 hodin. Připravené obrázky se uloží do adresáře "/var/www/server_monitor/static/". „Visualization.py“ a „app.py“ tedy neinteragují přímo, ale prostřednictvím souborového systému.

4.4.12 Automatizace vykreslování

Aby skript, který vizualizuje data, poskytoval aplikaci aktuální obrázky grafů, měl by se spouštět pravidelně, stejně jako skript, který sbírá data. Pro zjednodušení úlohy synchronizace byl vytvořen další skript.

Kód skriptu "scan_watcher.py" (viz příloha 10) je připojen ke kolekci "info" z databáze "scans" a začne sledovat tok transakcí. Pomocí správce kontextu "with" a metody watch() skript sleduje transakce typu "insert", které signalizují, že sbírka informací je dokončena. Pokud je podmínka splněna, skript spustí "visualisation.py" prostřednictvím interpretu lokálního virtuálního prostředí.

Nepřetržitý provoz skriptu scan_watcher.py se provádí pomocí systémové služby vytvořené příkazem "sudo nano /etc/systemd/system/scan_watcher.service" a níže uvedeného kódu (viz ukázka kódu 26).

```
[Unit]
Description=non-stop running script to monitor changes done to "info" DB, "scan" collection

[Service]
User=root
WorkingDirectory=/home/xmori001/db_scripts/
ExecStart=/bin/bash -c 'cd /home/xmori001/db_scripts/ && source local_env/bin/activate &&
python scan_watcher.py'
Restart=always
RestartSec=5

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

Ukázka kódu 26 - scan_watcher.service (vlastní zpracování)

4.5 Spuštění a testování

4.5.1 Spuštění komponent

Aby systém složený z mnoha komponent správně fungoval, je nutné se přesvědčit, že každá komponenta byla spuštěna a funguje.

Po konfiguraci databázového serveru (kapitola 4.4.2) musí tento server zůstat spuštěný. To lze zkontrolovat příkazem "systemctl status mongod", případně restartovat příkazem "sudo systemctl restart mongod".

Dalším krokem bylo naplánování skriptu pro naplnění databáze. To bylo provedeno pomocí standardního linuxového nástroje Cron. Pro naplánování sběru dat s minutovou frekvencí byl použit příkaz "**crontab -e**", který otevřel soubor crontab k úpravě. V posledním řádku pak byla naplánována úloha (tzv. "cron job") pomocí daného kodu:

„*** * * * * /home/xmori001/db_scripts/filler.py** “

Pět hvězdiček se interpretuje jako "každou minutu", "každou hodinu", "každý den v měsíci", "každý měsíc" a "každý den v týdnu". Protože první řádek skriptu obsahuje shebang s cestou k interpretu virtuálního prostředí, stačí v crontabu zadat pouze cestu k souboru. Poté byl příkazem "chmod 755 filler.py" skript uveden do spustitelného stavu.

Dále byla pro aktualizaci obrázků s grafy spuštěna služba vytvořená v kapitole 4.4.12. Za tímto účelem byly provedeny 2 příkazy – "**sudo systemctl daemon-reload**" a "**sudo systemctl start scan_watcher.service**".

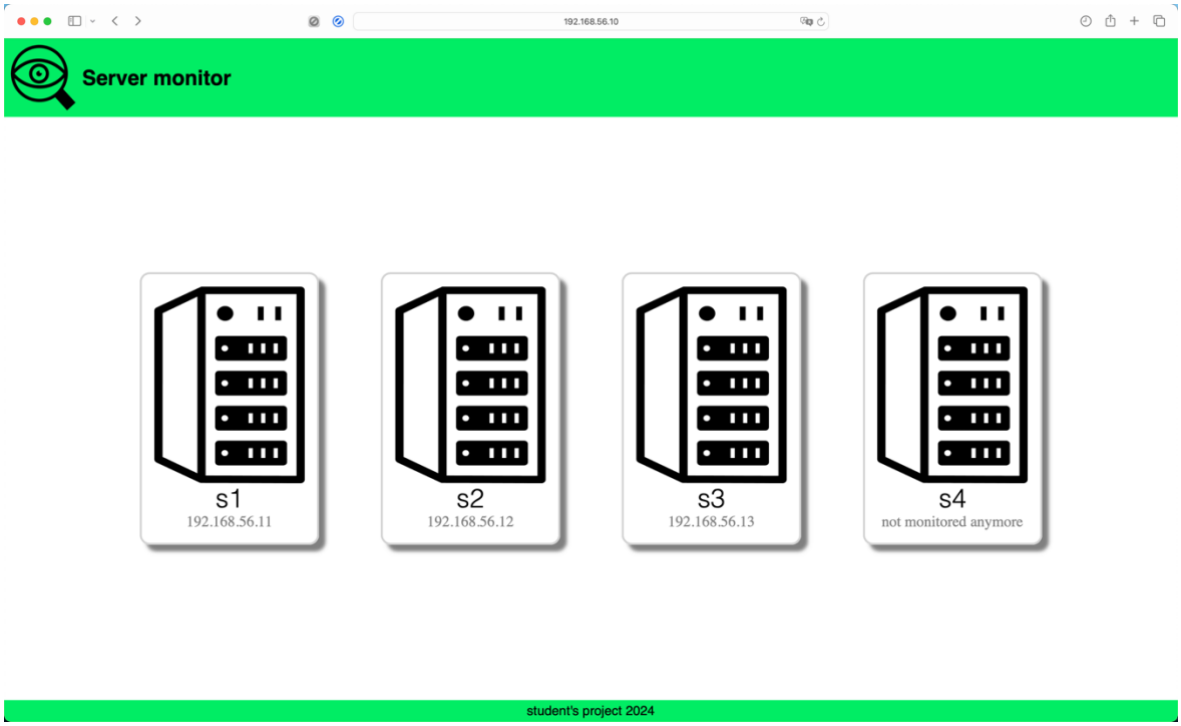
Posledním krokem před spuštěním projektu bylo restartování služby apache2, protože konfigurační soubor "server_monitor.conf" byl aktivován v kapitole 4.4.8. Za tímto účelem byl spuštěn příkaz "**systemctl reload apache2**".

4.5.2 Testování

Před testováním funkčnosti byl celý systém ponechán v aktivním stavu něco přes 12 hodin, aby se v databázi nahromadilo určité množství dat. Pro simulaci zátěže byla na virtuálním počítači s2 spuštěna řada programů integrovaných do distribuce Lubuntu. Hodinu před zahájením testování byly všechny programy ukončeny a poté byl spuštěn standardní prohlížeč firefox.

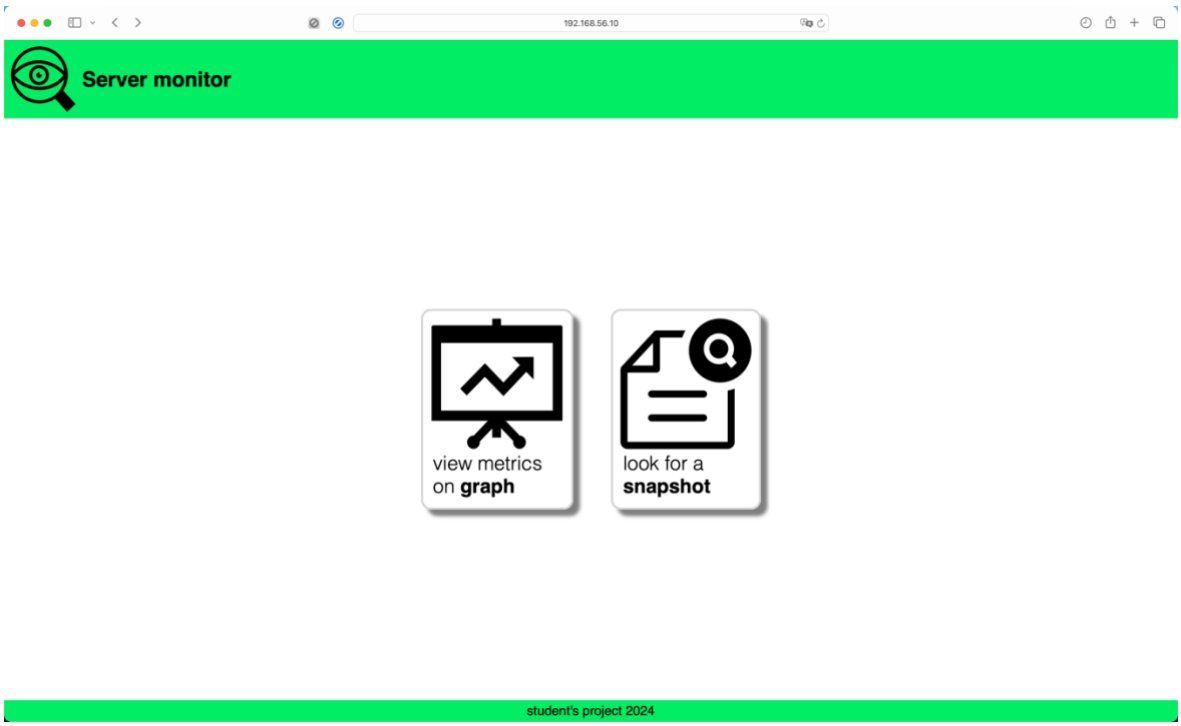
Pro demonstraci hlavního menu aplikace byla do databáze "logs" přidána prázdná kolekce s názvem "s4", aniž by byl server s4 vytvořen nebo zadán do kolekce "monitored servers".

Aplikace byla otevřena přechodem na adresu 192.168.56.10 pomocí prohlížeče na hostitelském počítači. Na hlavní stránce byly zobrazeny karty se servery s1, s2, s3 a také karta s fiktivní kolekcí s4 s poznámkou, že tento "server není monitorován" (viz obrázek 11).



Obrázek 11- domovská stránka (vlastní zpracování)

Protože jako testovaný stroj byl vybrán virtuální počítač s2, byl kliknutím na tuto kartu proveden přechod na stránku "Options" (viz obrázek 12).

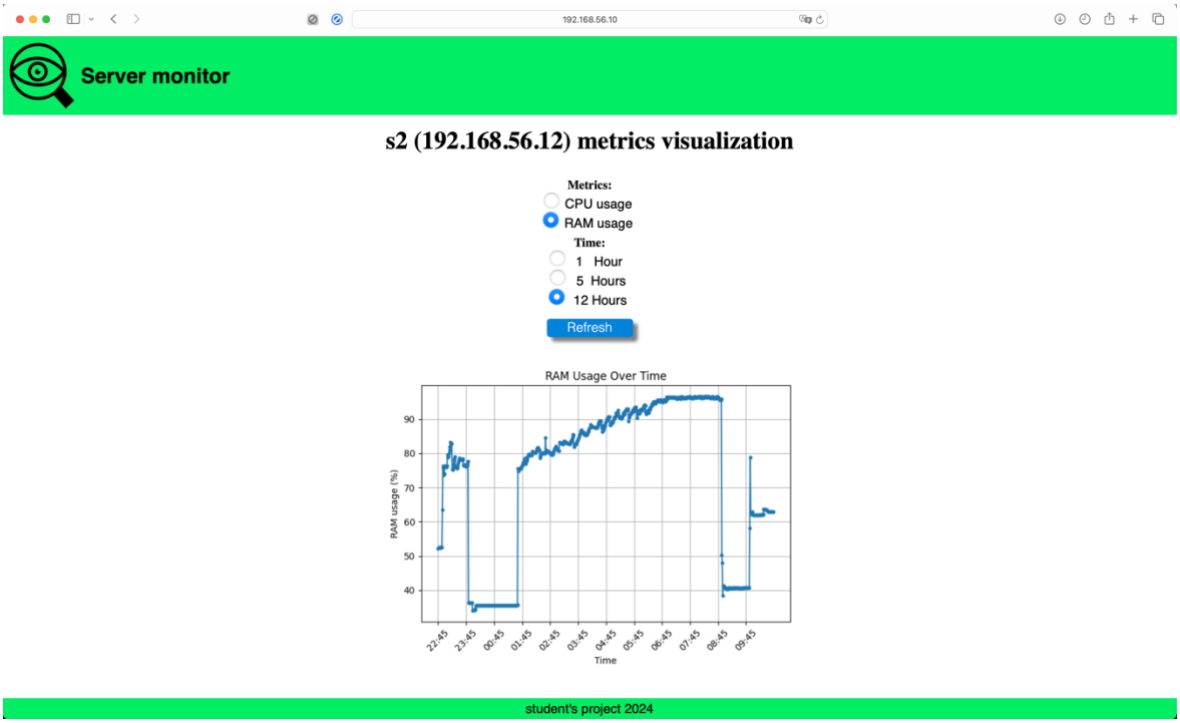


Obrázek 12 - stránka "options" (vlastní zpracování)

Dále byla adresa stránky nahrazena adresou "192.168.56.10/options/s999". To způsobilo chybu 404 a aplikace vrátila odpovídající stránku (viz obrázek 13).

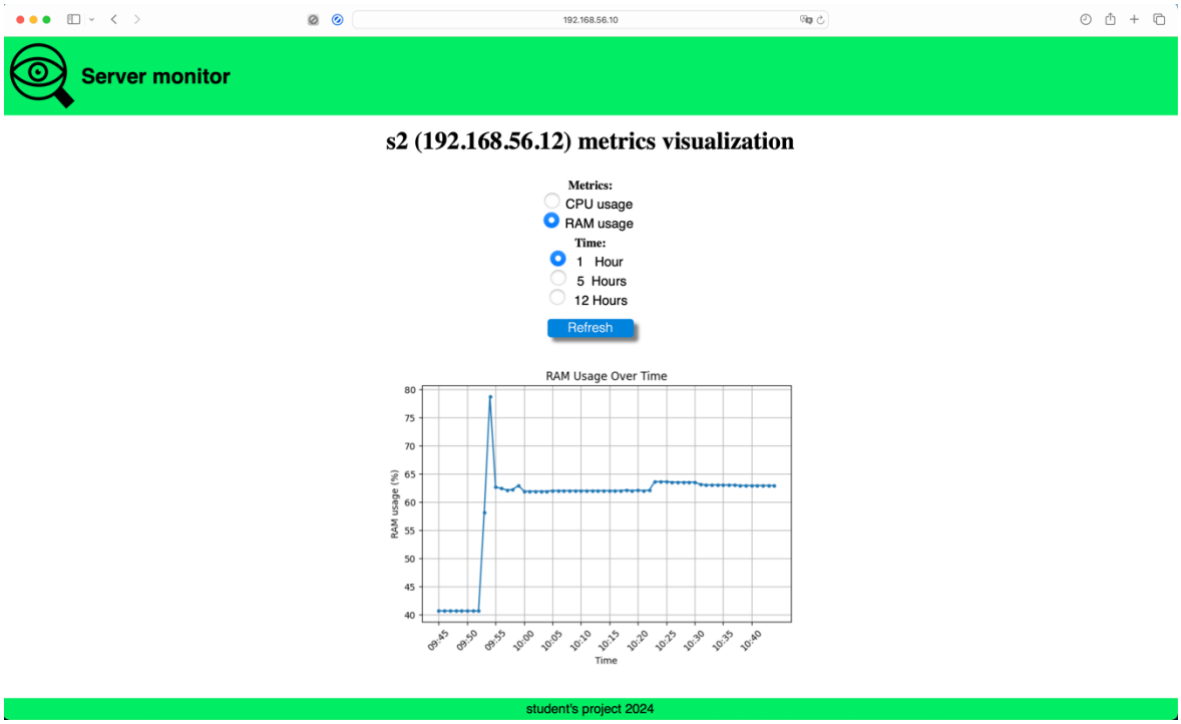
Obrázek 13 - stránka "404" (vlastní zpracování)

Po návratu na stránku "Options" byla vybrána karta s grafem. Dále byla na příslušné stránce v části metriky vybrána paměť RAM a v části časových intervalů bylo vybráno 12 hodin. Stisknutím tlačítka obnovit byla stránka obnovena a byl do ní zahrnut příslušný graf (viz obrázek 14).



Obrázek 14 - Využití paměti RAM za posledních 12 hodin (vlastní zpracování)

Graf jasně ukazuje okamžik vypnutí všech programů a následné spuštění prohlížeče Firefox. Pro podrobnější zkoumání byl zvolen časový interval 1 hodiny. Na aktualizovaném grafu (viz obrázek 15) je jasně vidět skok v 9:54.

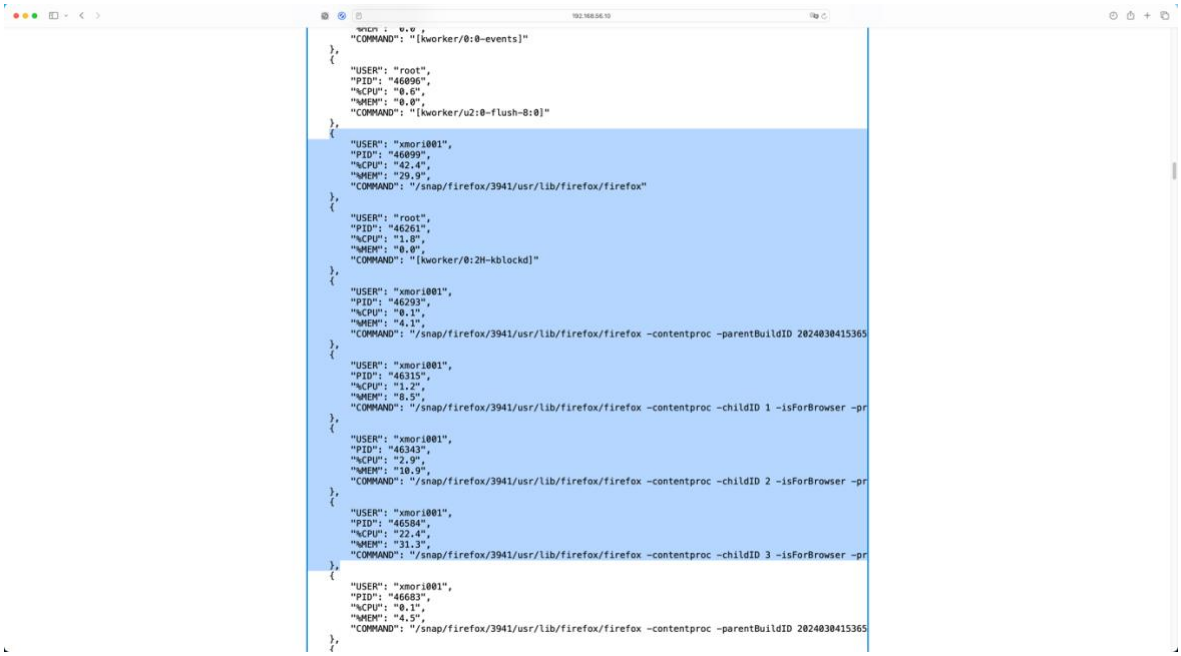


Obrázek 15 - Využití paměti RAM za poslední hodinu (vlastní zpracování)

Pro podrobnější studium tohoto časového okamžiku byl proveden přechod na stránku vyhledávání snímků. Dále byly do vstupních polí zadány číselné hodnoty odpovídající datu a času na grafu. Níže je uveden snímek obrazovky vygenerované stránky (viz obrázek 16).

Obrázek 16 – stránka vyhledávání snapshotů (vlastní zpracování)

Při prohlížení sekce aktivních procesů byla vyhledána skupina procesů firefox (viz obrázek 17).



Obrázek 17 - skupina procesů firefox (vlastní zpracování)

5 Výsledky a diskuse

5.1 Implementace

Během implementace projektu byl databázový server jednou z nejsnáze konfigurovatelných komponent. Tato skutečnost potvrzuje tvrzení, že MongoDB je snadno naučitelná technologie. Přehledně napsaná dokumentace a diskuse v uživatelských skupinách umožnily rychle najít odpovědi na všechny otázky. Instalace, převod na sadu replik s jedním uzlem a zprovoznění databázového serveru trvalo méně než hodinu. Pro srovnání, instalace, konfigurace a spuštění webového serveru Apache s demoverzí aplikace trvalo 2 dny.

Zvládnutí specifického dotazovacího jazyka pro práci s daty vyžaduje čas, ale obecně zkušenosti s používáním relačních databází pomohou rychle pochopit princip MongoDB a její funkce.

5.2 Testování

Při spuštění a testování finální verze projektu fungovala MongoDB podle původního plánu. Nalezení snapshotu a dokonce i načtení 720 posledních dokumentů pro vizualizaci bylo skoro okamžité. Problémem bylo renderování obrázků, které však nesouvisí s činností databáze.

Během testování předběžné verze projektu došlo k incidentu způsobenému nedostatkem přiděleného místa na disku virtuálního počítače S0. MongoDB neupozornila na nedostatek místa na disku, zatímco se ukládaly nové logy. To mělo za následek 99% obsazení místa a zamrznutí celého virtuálního počítače. I po uvolnění místa odstraněním souborů se shell mongosh nechtěl spustit. Situace se vyřešila návratem k dříve vytvořenému obrazu OS.

Pro kompletnější testování MongoDB je potřeba více času a zdrojů, což však přesahuje rámec bakalářské práce.

5.3 Zhodnocení

Díky flexibilitě dokumentově orientovaného datového modelu bylo možné vytvořit systém logování, který by mohl být alternativou k relačnímu řešení. Tento systém lze snadno přizpůsobit pro sběr jakéhokoliv formátu dat. Je důležité zdůraznit, že celý projekt byl

realizován na základě široce používaných open source technologií (s výjimkou VitualBoxu), které snadno projdou testem shody s bezpečnostními normami IT ve velkých korporacích.

5.4 Možnosti zlepšení

Navrhovaný projekt by mohl být vylepšen následujícími způsoby:

- přidáním data vypršení platnosti (tzv "time to live") do sbírky protokolů, po kterém by se dokumenty automaticky vymazaly;
- vytvořením samostatných uživatelů v MongoDB pro každou komponentu s omezenými právy pro čtení a zápis;
- přepsáním souboru "app.py" do formátu REST API a zvolením frontendové technologie schopné vykreslovat grafy v okamžiku žádosti;
- přidáním více konstrukcí „try-except“, aby se zvýšila stabilita.

6 Závěr

V teoretické části byl proveden přehled historie vývoje datových modelů od síťového modelu až po polostrukturovanou éru. Dále byl popsán dokumentově orientovaný datový model MongoDB s důrazem na porovnání s relačním modelem. Byla také porovnána syntaxe javascriptového shellu mongosh s jazykem SQL. Porovnání ukázalo, že kolekce dokumentů mohou být flexibilnější alternativou relačních tabulek. Na závěr teoretické části byly zmíněny příklady použití MongoDB a popsány technologie kompatibilní s touto databází.

V praktické části byl vybrán a podrobněji popsán způsob využití MongoDB jako úložiště logů. Byly analyzovány problémy této případové studie, ukázány nevýhody možného řešení založeného na relační databázi a navrženo řešení založené na ukládání dat v dokumentech. Po výběru sady technologií byla popsána postupná realizace projektu sběru dat o stavu serverů, ukládání logů a zobrazování informací ve webové aplikaci. Testování konečné verze projektu prokázalo funkčnost systému a výhodnost použití MongoDB pro ukládání polostrukturovaných dat. Metodika tedy byla dodržena a cíle stanovené v zadání práce byly splněny.

7 Citovaná literatura

1. Joseph M. Hellerstein, Michael Stonebrker. *Readings in Database Systems.* Massachusetts : MIT, 2005. ISBN-13: 978-0262693233.

2. *A relational model of data for large shared data banks.* Codd, E. F. 6, New York : Communications of the ACM, 1970, Sv. 13. doi: 10.1145/362384.362685.

3. Hall, Mark. Oracle Corporation. *Encyclopedia Britannica.* [Online] Encyclopædia Britannica Inc., Feb 2024. [Citace: 25. 2 2024.] https://www.britannica.com/topic/Oracle-Corporation.

4. Otter, Alastair. Oracle, IBM fight on all fronts. *IT Web.* [Online] ITWeb Limited, 22. 10 2001. [Citace: 25. 2 2024.] https://www.itweb.co.za/article/oracle-ibm-fight-on-all-fronts/kLgB17eJm8YM59N4.

5. *The semantic data model: a modelling mechanism for data base applications.* Michael Hammer, Dennis McLeod. Austin Texas : Association for Computing Machinery, 1978. Proceedings of the 1978 ACM SIGMOD international conference on management of data. stránky 26-36.

6. *Semistructured data.* Buneman, Peter. Tucson Arizona USA : Association for Computing Machinery, 1997. PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. stránky 117-121. 978-0-89791-910-4.

7. Don Box, Aaron Skonnard, John Lam. *Essential XML: Beyond Markup.* místo neznámé : Addison-Wesley Professional, 2000. ISBN-13: 978-0201709148.

8. Shakuntala Gupta Edward, Navin Sabharwal. *Practical MongoDB: Architecting, Developing, and Administering MongoDB.* New Delhi : Apress, 2015. ISBN-13: 978-1-4842-0647-8.

9. *Survey on NoSQL database.* Jing Han, Haihong E, Guan Le and Jian Du. Port Elizabeth : IEEE, 2011. ISBN:978-1-4577-0208-2.

10. solid IT gmbh. DB-Engines Ranking. *db-engines.* [Online] solid IT, 2 2024. [Citace: 24. 2 2024.] https://db-engines.com/en/ranking.

11. MongoDB, Inc. What Is MongoDB? *mongodb.com.* [Online] MongoDB, Inc., 2024. [Citace: 25. 2 2024.] https://www.mongodb.com/what-is-mongodb.

12. Schaefer, Lauren. What is NoSQL? *mongodb.com.* [Online] MongoDB, Inc., 2 2024. [Citace: 24. 2 2024.] https://www.mongodb.com/nosql-

explained#:~:text=NoSQL%20databases%20(aka%20%22not%20only,wide%2Dcolumn%2C%20and%20graph..

13. *Evaluating NoSQL Document Oriented Data Model.* Ranc, H. Hashem and D. Vienna : IEEE, 2016. 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW). stránky 51-56. DOI: 10.1109/W-FiCloud.2016.26.

14. MongoDB, Inc. SQL to MongoDB Mapping Chart. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 2024. 2 24.] https://www.mongodb.com/docs/manual/reference/sql-comparison/.

15. —. Databases and Collections. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 25. 2 2024.] https://www.mongodb.com/docs/manual/core/databases-and-collections/.

16. Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garret, Tim Hawkins. *MongoDB in Action.* New York : Manning, 2016. ISBN: 9781617291609.

17. MongoDB, Inc. Capped Collections. [Online] 2023. [Citace: 10. 3 2024.] https://www.mongodb.com/docs/manual/core/capped-collections/.

18. —. Time Series. [Online] 2023. [Citace: 14. 3 2024.] https://www.mongodb.com/docs/manual/core/timeseries-collections/.

19. —. Collation. [Online] 2023. [Citace: 10. 3 2024.] https://www.mongodb.com/docs/manual/reference/collation/.

20. —. Clustered Collections. [Online] 2023. [Citace: 10. 3 2024.] https://www.mongodb.com/docs/manual/core/clustered-collections/.

21. —. MongoDB Limits and Thresholds. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 25. 2 2024.] https://www.mongodb.com/docs/manual/reference/limits/.

22. Matsievskii, Nikolai. JSON and XML. What is better? *habr.com.* [Online] Habr, airee.ru, 23. 8 2003. [Citace: 25. 2 2024.] https://habr.com/ru/articles/31225/.

23. MongoDB, Inc. JSON and BSON. *mongodb.com.* [Online] MongoDB, Inc., 2024. [Citace: 25. 2 2024.] https://www.mongodb.com/json-and-bson.

24. —. BSON types. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 25. 2 2024.] https://www.mongodb.com/docs/manual/reference/bson-types/.

25. —. Specify JSON Schema Validation. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 25. 2 2024.] https://www.mongodb.com/docs/manual/core/schema-validation/specify-json-schema/.

26. Naik, Shefali. *Concepts of Database Management System.* místo neznámé : Pearson, 2013. ISBN-13: 978-9332526280.

27. MongoDB, Inc. Embedded Data Versus References. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 26. 2 25.] https://www.mongodb.com/docs/manual/data-modeling/concepts/embedding-vs-references/.

28. —. FAQ: MongoDB Fundamentals. [Online] 2023. [Citace: 15. 3 2024.] https://www.mongodb.com/docs/manual/faq/fundamentals/#:~:text=for%20the%20collections.-,Does%20MongoDB%20support%20SQL%3F,language%2C%20see%20MongoDB%20CRUD%20Operations..

29. —. Query Documents. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 26. 2 2024.] https://www.mongodb.com/docs/manual/tutorial/query-documents/.

30. —. Welcome to MongoDB Shell. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 26. 2 2024.] https://www.mongodb.com/docs/mongodb-shell/#mongodb-binary-bin.mongosh.

31. —. Multi-Document ACID Transactions on MongoDB. *mongodb.com.* [Online] MongoDB, Inc., 2019. [Citace: 23. 2 2024.] https://www.mongodb.com/collateral/mongodb-multi-document-acid-transactions.

32. —. Atomicity and Transactions. *mongodb.com.* [Online] MongoDB, Inc., 2023. [Citace: 23. 2 2024.] https://www.mongodb.com/docs/manual/core/write-operations-atomicity/.

33. —. MongoDB for Artificial Intelligence. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/use-cases/artificial-intelligence.

34. —. A mobile backend for any stack. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/solutions/use-cases/mobile.

35. —. MongoDB for Internet of Things (IoT). [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/solutions/use-cases/internet-of-things.

36. —. Catalog. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/use-cases/catalog.

37. —. Content Management. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/use-cases/content-management.

38. —. MongoDB is Fantastic for Logging. [Online] 2017. [Citace: 14. 3 2024.] https://www.mongodb.com/blog/post/mongodb-is-fantastic-for-logging .

39. —. Single View. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/use-cases/single-view .

40. —. Customer success stories. [Online] 2024. [Citace: 14. 10 2024.] https://www.mongodb.com/who-uses-mongodb.

41. —. MongoDB Compatibility & Integration. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/compatibility .

42. Q-Success. Comparison of the usage statistics of Linux vs. Windows for websites. [Online] 2024. [Citace: 14. 3 2024.] https://w3techs.com/technologies/comparison/os-linux,os-windows .

43. MongoDB, Inc. MongoDB Compatibility & Integration. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/compatibility.

44. Q-Success. Usage statistics of Linux for websites . [Online] 2024. [Citace: 14. 3 2024.] https://w3techs.com/technologies/details/os-linux .

45. Canonical Ltd. The story of Ubuntu. [Online] 2024. [Citace: 14. 3 2024.] https://ubuntu.com/about .

46. MongoDB, Inc. MongoDB Supported Languages. [Online] 2024. [Citace: 14. 3 2024.] https://www.mongodb.com/languages.

47. Python Software Foundation. About. [Online] 2024. [Citace: 14. 3 2024.] https://www.python.org/about/.

48. —. What is Python? Executive Summary. [Online] 2024. [Citace: 14. 3 2024.] https://www.python.org/doc/essays/blurb/.

49. MongoDB, Inc. PyMongo. [Online] 2023. [Citace: 14. 3 2024.] https://www.mongodb.com/docs/drivers/pymongo/.

50. Rite, Encardio. Guide to Data Loggers: What is a Data Logger, Types & How it Works? [Online] 2019. [Citace: 11. 3 2024.] https://encardio.medium.com/guide-to-data-loggers-what-is-a-data-logger-types-how-it-works-5843109bb1a6.

51. Simon Quigley, Lyn Perrine, Jacob Kim, Daniel Lim. Lubuntu Manual. [Online] 2023. [Citace: 11. 3 2024.] https://manual.lubuntu.me/stable/.

52. Ubuntu wiki. Python. [Online] 2017. [Citace: 11. 3 2024.] https://wiki.ubuntu.com/Python.

53. JetBrains s.r.o. Python. [Online] 2022. [Citace: 12. 3 2024.] https://www.jetbrains.com/lp/devecosystem-2022/python/.

54. Pallets. User’s Guide. [Online] 2010. [Citace: 12. 3 2024.] https://flask.palletsprojects.com/en/3.0.x/.

55. StackShare, Inc. Bottle vs Flask. [Online] 2024. [Citace: 12. 3 2024.] https://stackshare.io/stackups/bottle-vs-flask.
56. WSGI. What is WSGI? [Online] [Citace: 13. 3 2024.] https://wsgi.readthedocs.io/en/latest/what.html.
57. Hellkamp, Marcel. Bottle: Python Web Framework. [Online] 2022. [Citace: 13. 3 2024.] https://bottlepy.org/docs/dev/.
58. Matplotlib development team. Matplotlib 3.8.3 documentation. [Online] 2024. [Citace: 13. 3 2024.] https://matplotlib.org/stable/index.html.
59. The Apache Software Foundation. Main page. [Online] 2023. [Citace: 13. 3 2024.] https://httpd.apache.org/.
60. Dumpleton, Graham. mod_wsgi. [Online] 2023. [Citace: 13. 3 2024.] https://modwsgi.readthedocs.io/en/master/.
61. Ken W. Alger, Daniel Coupal. Building with Patterns: The Polymorphic Pattern. [Online] 2022. [Citace: 13. 3 2024.] https://www.mongodb.com/developer/products/mongodb/polymorphic-pattern/.
62. RaspberryTips. The Complete Beginner's Guide to The Ubuntu Hosts File. [Online] 2024. [Citace: 14. 3 2024.] https://raspberrytips.com/ubuntu-hosts-file/.
63. Canonical Ltd. NetworkManager and netplan. [Online] 2024. [Citace: 14. 3 2024.] https://ubuntu.com/core/docs/networkmanager/networkmanager-and-netplan.
64. MongoDB, Inc. Install MongoDB Community Edition on Ubuntu. [Online] 2023. [Citace: 14. 3 2024.] https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-ubuntu/.
65. Data Modeling. *mongodb.com*. [Online] MongoDB, Inc., 2023. [Citace: 25. 2 2024.] https://www.mongodb.com/docs/manual/data-modeling/.

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam ukázek kódu

Přílohy

Příloha 1 – app.py

```
from bottle import Bottle, run, template, static_file, request
from datetime import datetime, timedelta
```

```
import pymongo
import json
import os

APP_DIR = os.path.dirname(os.path.abspath(__file__))

app = Bottle()
conn = pymongo.MongoClient("mongodb://localhost:27017")



def get_servers_dict(conn):
    db = conn["logs"]
    available_logs = [name for name in db.list_collection_names() if not name.startswith("system.")]

    db = conn["info"]
    collection = db["monitored_servers"]

    servers_dict = {}
    for sname in sorted(available_logs):
        monitored_srv = collection.find_one({ "name": sname })
        if monitored_srv:
            servers_dict[sname] = monitored_srv["ip"]
        else:
            servers_dict[sname] = "not monitored anymore"

    return servers_dict



@app.route('/')
def index():
    servers = get_servers_dict(conn)
    return template(f'{APP_DIR}/views/index.tpl',
                    servers=servers)
```

```
@app.route('/options/<sname>')
def options(sname):
    servers = get_servers_dict(conn)
    if sname in servers.keys():
        return template(f'{APP_DIR}/views/options.tpl',
                        sname=sname,
                        servers=servers)
    else:
        return template(f"{APP_DIR}/views/404.tpl")

@app.route('/graph/<sname>')
def graph(sname):
    servers = get_servers_dict(conn)
    if sname in servers.keys():
        return template(f"{APP_DIR}/views/graph.tpl",
                        sname=sname,
                        servers=servers,
                        metric=None,
                        period=None)
    else:
        return template(f"{APP_DIR}/views/404.tpl")


@app.route('/graph/<sname>', method='POST')
def graph_submit(sname):
    servers = get_servers_dict(conn)
    if sname in servers.keys():
        metric = request.forms.get('metric')
        period = request.forms.get('period')
        return template(f"{APP_DIR}/views/graph.tpl",
                        sname=sname,
                        servers=servers,
                        metric=metric,
                        period=period)
```

```
    else:
        return template(f"{APP_DIR}/views/404.tpl")


@app.route('/snapshot/<sname>')
def snapshot(sname):
    servers = get_servers_dict(conn)
    if sname in servers.keys():
        return template(f"{APP_DIR}/views/snapshot.tpl",
                        sname=sname,
                        servers=servers,
                        valid=None)
    else:
        return template("404.tpl")


@app.route('/snapshot/<sname>', method='POST')
def snapshot_submit(sname):
    minute = request.forms.get('minute')
    hour = request.forms.get('hour')
    day = request.forms.get('day')
    month = request.forms.get('month')
    year = request.forms.get('year')
    try:
        validity = True
        converted_dt = datetime(year=int(year),
                                month=int(month),
                                day=int(day),
                                hour=int(hour),
                                minute=int(minute))

        current_time = datetime.now()
        if converted_dt > current_time:
            validity = False
```

```
except Exception:
    converted_dt = datetime.now()
    validity = False

servers = get_servers_dict(conn)
if sname in servers.keys():

    if validity:
        db = conn["logs"]
        collection = db[sname]
        buffer = timedelta(seconds=59)
        state = collection.find_one(
            {
                "$and": [
                    {"timestamp" : {"$gt": converted_dt}},
                    {"timestamp" : {"$lt": converted_dt + buffer}}
                ]
            }
        )
        procs = json.dumps(state["procs"], indent=4)
        services = json.dumps(state["services"], indent=4)
        del state["_id"], state["procs"], state["services"]
        state["timestamp"] = str(state["timestamp"])
        return template(f"{APP_DIR}/views/snapshot.tpl",
                        sname=sname,
                        servers=servers,
                        minute=minute, hour=hour, day=day, month=month, year=year,
                        dt = converted_dt,
                        valid=validity,
                        totals=json.dumps(state, indent=4),
                        procs=procs,
                        services=services)
else:
```

```
    return template(f"{APP_DIR}/views/404.tpl")


@app.route('/static/<filename:path>')
def serve_static(filename):
    return static_file(filename, root=f"{APP_DIR}/static")


@app.error(404)
def not_found(error):
    return template(f"{APP_DIR}/views/404.tpl")
```

Příloha 2 – styles.css

```
* {
  box-sizing: border-box;
  margin: 0;
  text-decoration: none;
}

body {
  background-color: white;
}

.layout {
  display: grid;
  grid-template-columns: 1fr;
  grid-template-rows: auto 1fr auto;
  gap: 0px 0px;
  min-height: 100vh;
}

.layout__header{
  background-color: #00ED64;
}

.layout__center{
  display: flex;
  justify-content: center;
  align-items: center;
}

.layout__footer {
  background-color: #00ED64;
  text-align: center;
  font-size: 20px;
```

```
  font-family: sans-serif;
  padding: 5px;
}

.header-container {
  display: flex;
  justify-content: flex-start;
  align-items: center;
  font-family: sans-serif;
}

.main-logo {
  margin: 10px;
  height: 100px;
}

.cards-container {
  display: flex;
  justify-content: center;
  align-items: center;
}

.card {
  margin: 5%;
  padding: 2%;
  border-style: solid;
  border-color: lightgrey;
  border-radius: 15px;
  box-shadow: 10px 10px 5px grey;
}

.card-content {
  display: flex;
  flex-direction: column;
  align-items: center;
}

.sname {
  color: black;
  font-size: 40px;
  font-family: sans-serif;
  font-weight: 200;
}

.address {
  color: grey;
  font-size: 23px;
}
```

```
.option {
  color: black;
  font-size: 30px;
  font-family: sans-serif;
  font-weight: 200;
  margin: 2%;
}

.central-container {
  display: flex;
  flex-direction: column;
  justify-content: space-around;
  align-items: center;
}

.graph {
  width: 640px;
  height: 480px;
  border: 1px white solid;
  margin-bottom: 1%;
  border-radius: 15px;
}

.description {
  font-size:  25px;
}


.selection-heading {
  text-align: center;
}

.refresh-button {
  width: 100%;
  margin-top: 10px;
  border: none;
  background-color: #0083DC;
  color: white;
  font-size: 20px;
  font-weight: 200;
  font-family: sans-serif;
  border-radius: 5px;
  box-shadow: 7px 7px 5px grey;
}

.rb {
  display: flex;
```

```
  justify-content: space-around;
  margin: 5%;
}

.rb label {
  margin-left: 15px;
}

.rb label pre{
  font-size: 20px;
  font-family: sans-serif;
}

input[type=radio] {
  transform: scale(2);
}

.central-container-snapshot {
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
  align-items: center;
}

.central-container-snapshot > * {
  margin: 1%;
}

.snapshot-search {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}

.snapshot-search-container {
  width: 100%;
}

.snapshot-search{
  width: 100%;
}

.dt-fields-container{
  width: 100%;
  display: flex;
  justify-content: center;
}
```

```
.dt-fields-container{
  display: flex;
}

.dt-fields-container > label {
  font-family: sans-serif;
  font-size: 20px;
  margin-left: 10px;
  margin-right: 10px;
}

.search-button {
  border: none;
  background-color: #0083DC;
  color: white;
  font-size: 20px;
  font-weight: 200;
  font-family: sans-serif;
  border-radius: 5px;
  box-shadow: 7px 7px 5px grey;
  width: 10%;
  margin: 1%;
  padding: 0.5%;
}

.doc-box {
  width: 90vh;
  border: 2px #0083DC solid;
}

.doc-box > pre {
  overflow: auto;
}
```

Příloha 3 – index.tpl

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/static/styles.css">
  <title>Server monitor</title>
</head>

<body class="layout">
  <header class="layout__header">
    <div class="header-container">
```

```
    <img src="/static/insight.png" class="main-logo">
    <h1>Server monitor</h1>
   </div>
  </header>

  <div class="layout__center">
   <div class="cards-container">
    % for sname in servers:
     <a href="/options/{{ sname }}" class="card">
      <div class="card-content">
       <img src="/static/server.png" alt="Server" width="230" height="300">
       <span class="sname">{{ sname }}</span>
       <span class="address">{{ servers[sname] }}</span>
      </div>
     </a>
    % end
   </div>
  </div>

  <footer class="layout__footer">
   <span>student's project 2024</span>
  </footer>
</body>
```

Příloha 4 – options.tpl

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/static/styles.css">
  <title>test</title>
</head>
<body class="layout">
  <header class="layout__header">
    <div class="header-container">
      <img src="/static/insight.png" class="main-logo">
      <h1>Server monitor</h1>
    </div>
  </header>


    <div class="layout__center">
       <div class="cards-container">
          <a href="/graph/{{ sname }}" class="card">
             <div class="card-content">
                <img src="/static/chart.png" alt="Server" width="200" height="200">
                <span class="option">view metrics on <strong>graph</strong></span>
             </div>
```

```
        </a>

        <a href="/snapshot/{{ sname }}" class="card">
          <div class="card-content">
            <img src="/static/doc1.png" alt="doc" width="200" height="200">
            <span class="option">look for a <strong>snapshot</strong></span>
          </div>
        </a>
      </div>
    </div>


  <footer class="layout__footer">
    <span>student's project 2024</span>
  </footer>
</body>
```

Příloha 5 – graph.tpl

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/static/styles.css">
  <title>test</title>
</head>
<body class="layout">
  <header class="layout__header">
    <div class="header-container">
      <img src="/static/insight.png" class="main-logo">
      <h1>Server monitor</h1>
    </div>
  </header>

  <div class="central-container">

    <div class="description">
      <h2> {{ sname }} ({{ servers[sname] }}) metrics visualization</h2>
    </div>


    <form action="/graph/{{ sname }}" method="post" class="form">

      <div class="variants-container">

        <div class="metrics">
          <h3 class="selection-heading">Metrics:</h3>
```

```
        <div class="rb">
          <input type="radio" name="metric" value="cpu" {{ "checked" if metric == "cpu" else "" }} class="graph-input">
          <label for="cpu"><pre>CPU usage</pre></label>
        </div>

        <div class="rb">
          <input type="radio" name="metric" value="ram" {{ "checked" if metric == "ram" else "" }} class="graph-input">
          <label for="cpu"><pre>RAM usage</pre></label>
        </div>
       </div>

       <div class="period">
        <h3 class="selection-heading">Time:</h3>

        <div class="rb">
          <input type="radio" name="period" value="1" {{ "checked" if period == "1" else "" }} class="graph-input">
          <label for="1"><pre>1   Hour </pre></label>
        </div>

        <div class="rb">
          <input type="radio" name="period" value="5" {{ "checked" if period == "5" else "" }} class="graph-input">
          <label for="5"><pre>5  Hours</pre></label>
        </div>

        <div class="rb">
          <input type="radio" name="period" value="12" {{ "checked" if period == "12" else "" }} class="graph-input">
          <label for="12"><pre>12 Hours</pre></label>
        </div>
       </div>

     </div>

     <div>
       <input type="submit" value="Refresh" class="refresh-button">
     </div>

   </form>


   % if metric and period:
     <div class="graph">
       <img src="/static/{{ sname }}_{{ metric }}_{{ period }}.png" alt="graph" width="640" height="480">
```

```
    </div>
   % else:
    <div class="graph"></div>
   % end

  </div>


  <footer class="layout__footer">
   <span>student's project 2024</span>
  </footer>
</body>
```

Příloha 6 – snapshot.tpl

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/static/styles.css">
  <title>test</title>
</head>

<body class="layout">
  <header class="layout__header">
    <div class="header-container">
      <img src="/static/insight.png" class="main-logo">
      <h1>Server monitor</h1>
    </div>
  </header>

  <div class="central-container-snapshot">
    <h2 class="description"> {{ sname }} ({{ servers[sname] }}) snapshot search:</h1>

    <div class="snapshot-search-container">
      <form action="/snapshot/{{ sname }}" method="post" class="snapshot-search">
        <div class="dt-fields-container">
          <label for="minute">Minute:</label>
          <input type="number" id="minute" name="minute" min="0" max="59" required>
          <label for="hour">Hour:</label>
          <input type="number" id="hour" name="hour" min="0" max="23" required>
          <label for="day">Day:</label>
          <input type="number" id="day" name="day" min="1" max="31" required>
          <label for="month">Month:</label>
          <input type="number" id="month" name="month" min="1" max="12" required>
          <label for="year">Year:</label>
          <input type="number" id="year" name="year" min="2024" max="2099" required>
        </div>
```

```
      <input type="submit" value="Search" class="search-button">


    </form>
  </div>

  % if valid:
    <h2> Snapshot from {{ dt.strftime("%H:%M %d/%m/%Y") }}</h2>
  % elif valid is None:
    <h2> Select date and time using input fields </h2>
  % else:
    <h2>Invalid date! Try again</>
  % end


  % if valid:
    <div>
      <h3>system state</h3>
      <div class="doc-box">
        <pre>{{ totals }}</pre>
      </div>
    </div>

    <div>
      <h3>active processes</h3>
      <div class="doc-box">
        <pre>{{ procs }}</pre>
      </div>
    </div>

    <div>
      <h3>active system services</h3>
      <div class="doc-box">
        <pre>{{ services }}</pre>
      </div>
    </div>
  % end

</div>

<footer class="layout__footer">
  <span>student's project 2024</span>
</footer>
</body>
```

Příloha 7 – 404.tpl

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>404</title>
</head>


<img src="/static/insight.png" width='500' height='500'>
<h1 >requested resource doesn't exist</h1>
```

Příloha 8 – filler.py

```
#!/home/xmori001/db_scripts/local_env/bin/python3.11

import pymongo
import multiprocessing
import datetime
import subprocess
import shlex
# import sys

class DataCollector:
    def __init__(self, doc):
        self.sname = doc["name"]
        self.ip = doc["ip"]
        self.uname = doc["tech_user"]

        self.data_list = self.ssh_scraper(self.uname, self.ip)
        self.procs = DataCollector.procs_parser(self.data_list[0])
        self.services = DataCollector.services_parser(self.data_list[1])
        self.nproc = DataCollector.nporc_parser(self.data_list[2])
        self.cpu_usage = DataCollector.cpu_usage_parser(self.data_list[3])
        self.mem_total, self.mem_usage = DataCollector.ram_stats_parser(self.data_list[4])

    @staticmethod
    def ssh_scraper(user, address):
        try:
            cmd = (
                f"ssh {user}@{address} "
                "'ps --no-headers aux | grep -v -E \"ps\" && "
                "echo -e \"---\n\" && "
                "systemctl --state=active --quiet && "
                "echo -e \"---\n\" && "
                "nproc && "
                "echo -e \"---\n\" && "
                "vmstat 1 2 | tail -1 && "
                "echo -e \"---\n\" && "
```

```
            "free --mega | awk \"NR==2\" '"
        )

        tokenized_cmd = shlex.split(cmd)
        process = subprocess.Popen(tokenized_cmd,
                                   stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE,
                                   text=True)

        stdout, stderr = process.communicate()

        if process.returncode != 0:
            print("Error executing combined command:", stderr)
            return None
        return stdout.split("---\n")

    except Exception as e:
        print("Error:", str(e))
        return None


@staticmethod
def procs_parser(text):
    content = text.strip()
    procs_list = []

    for line in content.split('\n'):
        data = line.strip().split()
        if data[2] == data[3] == '0.0':
            continue

        proc = {}
        proc["USER"] = data[0]
        proc["PID"] = data[1]
        proc["%CPU"] = data[2]
        proc["%MEM"] = data[3]
        proc["COMMAND"] = ' '.join(data[10:])
        procs_list.append(proc)
    return procs_list


@staticmethod
def services_parser(text):
    content = text.strip()
    services_list = []

    for line in content.split('\n'):
        data = line.strip().split()
```

```
            srv = {}
            srv["UNIT"] = data[0]
            srv["LOAD"] = data[1]
            srv["ACTIVE"] = data[2]
            srv["SUB"] = data[3]
            srv["DESCRIPTION"] = ' '.join(data[4:])
            services_list.append(srv)
        return services_list

    @staticmethod
    def nporc_parser(text):
        content = text.strip()
        return int(content)

    @staticmethod
    def cpu_usage_parser(text):
        content = text.strip()
        idle_time = int(content.split()[-3])
        cpu_usage = 100 - idle_time
        return max(1, cpu_usage)

    @staticmethod
    def ram_stats_parser(text):
        content = text.strip()
        data = content.split()
        mem_total = int(data[1])
        mem_available = int(data[6])
        mem_usage = (mem_total - mem_available) * 100 / mem_total
        return mem_total, round(mem_usage, 2)


def proc_logic(doc):

    server_data = DataCollector(doc)
    timestamp = datetime.datetime.now()

    snapshot = {
        "timestamp": timestamp,
        "nproc": server_data.nproc,
        "CPU_usage_%": server_data.cpu_usage,
        "RAM_total_mb": server_data.mem_total,
        "RAM_usage_%": server_data.mem_usage,
        "procs": server_data.procs,
        "services": server_data.services
    }

    proc_conn = pymongo.MongoClient("mongodb://localhost:27017/")
    db = proc_conn["logs"]
```

```
    collection = db[server_data.sname]

    collection.insert_one(snapshot)
    proc_conn.close()


if __name__ == "__main__":
    init_conn = pymongo.MongoClient("mongodb://localhost:27017")
    db = init_conn["info"]
    collection = db["monitored_servers"]

    cursor = collection.find()

    processes = []

    for doc in cursor:
        process = multiprocessing.Process(target=proc_logic, args=(doc,))
        processes.append(process)
        process.start()

    for proc in processes:
        proc.join()

    message = []
    for i, proc in enumerate(processes):
        if proc.exitcode == 0:
            continue
        else:
            message.append(f"Process {i+1} failed.")

    if not len(message):
        message = "success"

    collection = db["scans"]
    collection.insert_one({"scan_time": datetime.datetime.now(), "message": message})
    cursor.close()
    init_conn.close()
```

Příloha 9 – visualization.py

```
import pymongo
import multiprocessing
import matplotlib.pyplot as plt
from datetime import datetime

def render_graph(server, x_values, y_values, type):
    plt.plot(x_values, y_values, marker='.')
```

```
    plt.title(f"{type.upper()} Usage Over Time")
    plt.xlabel("Time")
    plt.ylabel(f"{type.upper()} usage (%)")

    if len(x_values) == 60:
        plt.xticks(x_values)
        plt.xticks(range(0, len(x_values), 5))
    elif len(x_values) == 300:
        plt.xticks(range(0, len(x_values), 30))
    else:
        plt.xticks(range(0, len(x_values), 60))

    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.grid(True)


plt.savefig(f'/var/www/server_monitor/static/{server}_{type}_{int(len(x_values)//60)}.png
')
    plt.clf()


def proc_logic(doc):
    sname = doc['name']

    proc_conn = pymongo.MongoClient("mongodb://localhost:27017")
    db = proc_conn["logs"]
    collection = db[sname]

    logs_cursor = collection.find().sort({ "timestamp": 1 }).limit(720)

    timestamps = []
    ram_values = []
    cpu_values = []

    for record in logs_cursor:
        timestamps.append(datetime.strftime(record["timestamp"], "%H:%M"))
        ram_values.append(record["RAM_usage_%"])
        cpu_values.append(record["CPU_usage_%"])

    render_graph(sname,  timestamps[len(timestamps)-60:],  ram_values[len(timestamps)-
60:], "ram") # 720 - 60
    render_graph(sname,  timestamps[len(timestamps)-300:],  ram_values[len(timestamps)-
300:], "ram") # 720 - 300
    render_graph(sname, timestamps, ram_values, "ram")

    render_graph(sname,  timestamps[len(timestamps)-60:],  ram_values[len(timestamps)-
60:], "cpu")
```

```
    render_graph(sname, timestamps[len(timestamps)-300:], ram_values[len(timestamps)-
300:], "cpu")
    render_graph(sname, timestamps, ram_values, "cpu")


if __name__ == "__main__":
    init_conn = pymongo.MongoClient("mongodb://localhost:27017")
    db = init_conn["info"]
    collection = db["monitored_servers"]

    cursor = collection.find()

    processes = []

    for doc in cursor:
        process = multiprocessing.Process(target=proc_logic, args=(doc,))
        processes.append(process)
        process.start()

    cursor.close()
    init_conn.close()

    for proc in processes:
        proc.join()
```

Příloha 8 – filler.py

```
import pymongo
import os

ENV_PATH = "/home/xmori001/db_scripts/local_env/bin/python"
VISUALIZER_PATH = "/home/xmori001/db_scripts/visualization.py"

conn= pymongo.MongoClient("mongodb://localhost:27017")
target_db = conn["info"]
target_collection = target_db["scans"]

with target_collection.watch() as stream:
    for change in stream:
        if change['operationType'] == 'insert':
            os.system(f"{ENV_PATH} {VISUALIZER_PATH}")
```