Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitatívnych metód

Vizualizácia BTDB

Bakalárska práca

Autor: Matúš Rychvalský
Študijný odbor: Aplikovaná informatika

Vedúci práce: Ing. Barbora Tesařová, Ph.D.
Odborný konzultant: Jan Juza (Quadient)

Hradec Králové November 2019

Prehlásenie

Prehlasujem, že som bakalársku prácu spracoval samostatne a s použitím uvedenej literatúry.

vlastnoručný podpis

V Hradci Králové dne 30.4.2020

Poďakovanie:

Ďakujem mojej vedúcej bakalárskej práce Ing. Barbore Tesařovej, PhD. za metodické vedenie práce a konzultácie ohľadom jej obsahu. Taktiež ďakujem pánovi Janovi Juzovi za odborné rady a pomoc pri návrhu aplikácie.

Anotácia

Firma Quadient, ktorá sídli v Hradci Králové, potrebuje aplikáciu na vizualizáciu dát z ich vlastnej objektovej databázy BTDB. Cieľom tejto bakalárskej práce je vytvoriť webovú aplikáciu, ktorá by zvládla zobraziť celý obsah BTDB databázy tejto firmy. Na vývoj spomínanej webovej aplikácie bola použitá technológia ASP.NET Core. Význam práce je preto vytvorenie danej webovej aplikácie podľa zadaných kritérií od firmy Quadient. Výsledkom tejto záverečnej bakalárskej práce je plne funkčná webová aplikácia, ktorá je pripravená na nasadenie do produkcie na ich server. Môže byť ale použitá kýmkoľvek, kto používa BTDB databázu a potrebuje zobrazovať všetky dáta, čo v nej sú. Výsledná webová aplikácia zobrazuje všetky potrebné dáta v zadanom rozsahu.

Annotation

Title: Visualization of BTDB

Quadient company, which branch office is located in Hradec Králové, requires an application for data visualization from their own object database BTDB. The aim of this bachelor thesis is to make the web application which could show the whole content of BTDB database of this company. ASP.NET Core technology was used for the web application development. The main purpose of this bachelor thesis is to develop the specific web application based on criteria which were given by Quadient company. The result of this bachelor thesis is a fully functional web application which is ready to deploy on their production server. It can be used by anyone who uses BTDB database and needs to visualize all data in specific BTDB. The final web application shows all required data.

Obsah

Zoznam tabuliek

Zoznam grafov

Zoznam odkazov na kód

Zoznam obrázkov

Zoznam schém

1 Úvod

Každá firma v súčasnosti potrebuje pracovať s veľkými objemami dát ktoré sú uložené v databáze. Stále viac a viac firiem prechádza z relačných databáz, ktoré boli viac využívané v minulosti, na iný druh databáz, ktoré viac vyhovujú súčasným požiadavkám (napríklad rýchle načítanie dát, jednoduchšie vyhľadávanie alebo ľahká rozšíriteľnosť a udržateľnosť databázy). Prípadne sa niektoré firmy snažia o kombináciu viacerých druhov databáz.

Táto záverečná práca sa teoreticky bude venovať porovnaniu relačných, objektových a NoSQL databáz. Dôraz bude kladený na objektové databázy. Budú predstavené najnovšie trendy v oblasti práce s dátami. Taktiež sa bude zaoberať otázkou prečo sa objektové databázy oproti ostatným druhom databáz až tak neuchytili. Jej praktická časť bude zameraná na predstavenie BTDB objektovej databázy od firmy Quadient. Bude ukázané ako sa vytvára a používa v praxi. Výsledkom tejto záverečnej práce bude vytvorená aplikácia na zobrazenie celého obsahu BTDB databázy.

2 Cieľ práce

Cieľom tejto práce bolo vytvoriť grafickú aplikáciu, ktorá bude vedieť jednoducho vizualizovať dáta z ľubovoľnej BTDB databáze. Základom bude dll knižnica, ktorá bude vedieť jednoducho prechádzať a konvertovať dáta z ľubovoľnej BTDB databázy do dátovej štruktúry. Každá vizualizovaná položka musí obsahovať jej názov, typ a hodnotu. Bolo potrebné vymyslieť spôsob ako vizualizovať len potrebné dáta, napríklad len jeden koreňový objekt, jednu propertu, v prípade listu len nejakú podmnožinu.

Použité technológie:

- C#
- ASP.NET Core
- github
- BTDB

3 Obecná problematika práce s dátami

V dnešnej dobe je čoraz väčší dôraz kladený na prácu s dátami a nato potrebujeme pracovať s určitou databázou. Z toho vyplýva otázka čo je databáza? „Databáza je množina štruktúrovaných dát a slúži na uloženie informácií takým spôsobom, že počítačový program, alebo človek môže použiť špeciálny jazyk na získavanie týchto informácií. Vďaka presne určenej štruktúre umožňuje ľahké vyhľadávanie a triedenie aj pri veľkom množstve dát.“[1] V súčasnosti sú stále najpoužívanejšie takzvané relačné databázy ako napríklad MySQL alebo Oracle.

Dnes máme k dispozícii dva hlavné druhy databáz a to relačné, teda SQL databázy, a NoSQL databázy. Tie budem ďalej porovnávať s objektovými databázami. NoSQL databázy sú rôzne databázové techniky ktoré umožňujú ukladanie a manažment dát v rôznych formátoch nie len s použitím SQL.[2] Preto NoSQL databázy ďalej delíme na grafové, dokumentové, kľúč–hodnota (key-value) a riadkové.[3]

Do väčšej hĺbky budú rozoberané predovšetkým rozdiely medzi relačnými databázami a objektovo orientovanými databázami (Tabuľka č.1), hlavným zdrojom pritom bude [4]. Taktiež budú rozoberané dôvody prečo sa objektové databázy nerozšírili do takej miery ako NoSQL databázy. Objektovo orientovaná databáza je databáza, ktorá ukladá dáta vo forme objektov, na rozdiel od viac používaných relačných databáz, ktoré na ukladanie dát používajú tabuľky, ktoré obsahujú riadky a stĺpce. Objektovo orientované databázy používajú podobný princíp ako sa používa pri objektovom programovaní.[5] Pri tvorbe objektov je možné uložiť viac informácií. Pomocou atribútov a jednotlivých metód je možné uložiť podobu, ale aj správanie daného objektu. To sa napríklad hodí pri tvorbe komplexných systémov, ktoré potrebujú viac informácií. Napríklad pri zázname o pacientovi v nemocnici je potrebné využiť viac dátových štruktúr, ako napríklad röntgenovú snímku alebo možnosť naprogramovať metódu, ktorá pomôže lekárovi vytlačiť záverečnú správu o stave pacienta. Podobný výsledok sa dá docieliť aj pomocou relačnej databázy ale bolo by to zložitejšie a načítanie dát z databázy by trvalo dlhšie.

Klasická relačná databáza umožňuje definíciu dátových štruktúr, ukladacie a načítacie operácie a integritné obmedzenia. V podobnej databáze sú dáta a vzťahy medzi nimi organizované vo forme tabuliek. Tabuľka predstavuje kolekciu záznamov a každý záznam obsahuje určité polia, teda položky.

Vlastnosti jednotlivých tabuliek sú [4]:

- Hodnoty sú atomické, teda ďalej nedeliteľné
- Každý riadok je jedinečný
- Hodnoty stĺpcov sú vždy rovnakého typu
- Poradie stĺpcov je nepodstatné
- Poradie riadkov je nepodstatné
- Každý stĺpec má unikátne meno

3.1 Objektovo orientovaná databáza (OODB)

Objektovo orientovaná databáza (OODB) ukladá objekty, nie jednotlivé dáta, ako napríklad celé čísla, reťazce znakov, reálne čísla, znaky a iné. Na tvorbu objektov sa používajú objektovo orientované programovacie jazyky ako Java, C++ alebo C#. Objekt ako taký obsahuje dve hlavné časti: atribúty a metódy. Atribúty definujú vlastnosti daného objektu. Ako príklad je možné použiť objekt študenta, ktorý bude mať atribúty meno, id, bydlisko, ale aj objekt vo forme výsledkov štúdia. Z toho vyplýva, že atribúty môžu mať jednoduchú štruktúru, ako celé číslo alebo reťazec znakov, ale môžu mať aj komplexnejšiu hodnotu vo forme referencie na iný objekt. Metódy definujú ako sa daný objekt bude chovať a čo bude môcť vykonávať. Metódy sú pomenované taktiež ako funkcie alebo presnejšie procedúry. Pomocou metód je možné nastavovať, upravovať a zisťovať hodnoty jednotlivých atribútov daných objektov, takzvané gettery a settery.

OODB teda pomáhajú hlavne pri mapovaní jedna k jednej z objektov daného programu do objektov danej objektovej databázy. Odpadá nutnosť rozkladu dát jednotlivých objektov a ich úpravy aby bolo možné zapísať do tabuliek relačných databáz. Dve hlavné výhody tohto prístupu sú rýchlejšie spracovanie objektov a lepšie spravovanie komplexných vzťahov medzi objektami danej databázy. Preto sa OODB viac hodia na tvorbu aplikácií, ktoré predvídajú finančné trhy a ich

správanie, telekomunikačné aplikácie, World Wide Web štruktúry dokumentov, konštrukčné systémy, výrobné systémy a nemocničné záznamy.

3.2 Výhody OODB

Medzi výhody OODB patria[4]:

- Schopnosť využitia množstva rôznych dátových typov

  OODB môže ponúknuť možnosť uloženia všetkých dátových typov textu, čísel, obrázkov, videí a hlasu.

- Kombinácia objektovo orientovaného programovania s OODB

  Technológia OODB kombinuje objektovo orientované programovanie s databázovou technológiou, čo poskytuje zabudovaný systém pre vývoj aplikácií. Tým vzniká mnoho výhod, ktoré zahŕňajú definíciu operácií spolu s definíciu dát. V prvom rade sú používané definované operácie, ktoré sú nezávislé na databázovej aplikácií, ktorá sa práve používa. Potom je možné rozšíriť dátovú štruktúru pre podporu komplexných dát.

- Zlepšuje produktivitu

  Dedičnosť umožňuje programátorom vývoj riešenia komplexných problémov pomocou definovania nových objektov, za pomoci podmienok objektov definovaných v minulosti. Polymorfizmus a dynamické väzby umožňujú programátorom definovať operácie pre jeden objekt a zdieľať špecifickú funkciu s ostatnými objektami. Tieto objekty dokonca môžu danú operáciu vylepšiť, aby odpovedala jedinečnému správaniu daných objektov.

- Prístup k dátam

  OODB reprezentujú vzťahy explicitne podporou oboch prístupov, navigačného aj asociatívneho prístupu k dátam.

- Obohacujú schopnosti modelovania

  Objektovo orientovaný dátový model umožňuje bližšie modelovať „skutočný svet“. Objekty, ktoré obsahujú stav a chovanie, bližšie reprezentujú reálne objekty sveta. Objekt môže uchovávať všetky vzťahy,

ktoré má s inými objektmi vrátane vzťahu množiny M ku množine N a objekty môžu byť združené do komplexnejšieho objektu. Tradičné dátové modely toto tak ľahko nedokážu.

- Rozšíriteľnosť

  OODB umožňuje rozšírenie stávajúcich dátových typov o nové dátové typy. Schopnosť vyňať spoločné vlastnosti niekoľkých tried a zjednotiť ich v jednej super triede z ktorej môžu ostatné triedy dediť, môže veľmi výrazne znížiť redundanciu systému čo je jedna z najväčších výhod objektového prístupu. Opätovná využiteľnosť tried podporuje rýchlejší vývoj a jednoduchšiu údržbu databázy aj celej aplikácie.

- Odstránenie impedančných nezrovnalostí

  Nezávislý jazyk, medzi DML (Data Manipulation Laguage) a programovacím jazykom, prekonáva impedančnú nezrovnalosť. Toto eliminuje mnoho z efektívnosti, ktorá sa objavuje v mapovaní deklaratívneho jazyka ako SQL na príkazový jazyk ako C.

- Expresívnejší jazyk vyhľadávania

  Navigačný prístup je najpoužívanejším prístupom k dátam v OODB. Je to výrazný kontrast oproti asociatívnemu prístupu jazyka SQL, ktorý je založený na deklaratívnom vyhlásení, s výberom na základe jedného alebo viacerých predikátov. Navigačná explózia je vhodná na manipulovanie s časťami navigačnej explózie, rekurzívne dotazy a mnoho ďalších.

- Podporuje evolúciu schémy

  Úzky vzťah medzi dátami a aplikáciou s OODB umožňuje omnoho jednoduchšiu evolúciu schémy.

- Použiteľnosť pre pokročilé databázové systémy

  Je mnoho oblastí kde sa tradičná OODB nepoužívala, ako napríklad Computer Aided Design (CAD), Computer Aided Software Engineering (CASE), Office Information System (OIS) a multimediálne systémy. Vďaka

obohateným modelovacím schopnostiam OODB, už je vhodné používať tento prístup v daných aplikáciách.

- Žiadne primárne kľúče

  Užívateľ relačných databáz sa musí starať o jedinečný identifikačný znak a jeho hodnotu. Musí sa uistiť že žiadne hodnoty tohto znaku nie sú rovnaké, aby sa vyhol chybovým podmienkam. V OODB sa o jedinečný identifikátor objektu starať netreba, lebo sa o to stará systém skrz OID (id objektu), ktorý je pre užívateľa neviditeľný. Preto nevznikajú žiadne obmedzenia toho, čo môže byť v objekte uložené.

Tieto výhody a rozdiely sú zosumarizované v Tabuľke 1.

Tabuľka 1 Rozdiely medzi objektovou a relačnou databázou

Objektovo orientovaný model	**Relačný model**	**Rozdiely**
Objekt	Entita	Objekt špecifikuje správanie
Trieda objektov	Typy entít	Trieda objektov obsahuje podobné správanie objektov z danej triedy
Triedna hierarchia	Databázové schéma	Triedna hierarchia obsahuje dedenie, zatiaľ čo schéma používa externe kľúče
Inštancia triedy	Entita alebo záznam	Inštancia môže mať viac obmedzujúcich aktivít
Identita objektu (OID)	Primárny kľúč	V relačnom modeli primárny kľúč nie je identifikovaný systémom

3.3 NoSQL databázy

Najpoužívanejšie druhy NoSQL databáz sú[6]:

- Dokumentový databázový systém (Document database managnemt system, DDBMS) je navrhnutý na ukladanie celých dokumentov ako jednu entitu spolu s jej atribútmi. Dokumenty sú typicky ukladané vo forme JSON alebo XML formátov, ktoré je možné ľahko čítať, uložiť a parsovať pomocou API a zvolenej knižnice. DDBMS sa oproti relačným databázovým systémom rýchlo načítavajú, sú prístupné a parsovateľné. Užívateľské profily, systémy na správu obsahu (content managment systems) alebo katalógy sú časté prípady použitia DDBMS. K uloženiu dokumentu sa používa dvojica kľúč hodnota, kde kľúč predstavuje unikátny identifikátor dokumentu podobný primárnemu kľúču v SQL.[2; 7] Najpoužívanejším DDBMS a zároveň najpoužívanejšou NoSQL databázou je MongoDB.[6]
- Kľúč-hodnota, alebo aj Key-Value, databázy sú podobné DDBMS, kde ale každý záznam obsahuje kľúč a hodnotu. Hodnotu vyhľadávame pomocou referencie na ňu, teda pomocou kľúča, preto je vyhľadávanie jednoduché. Key-Value databázy sú výborné vtedy, keď potrebujeme ukladať veľké množstvo dát, ale nepotrebujeme zložité vyhľadávania nad danými dátami. Najčastejšie sa využívajú napríklad k uloženiu užívateľských preferencií.[8] Najpoužívanejšou Key-Value databázou je Redis.[6]

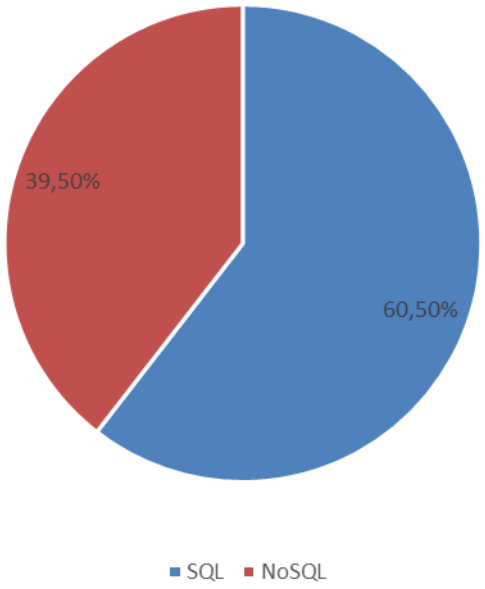
4 Aktuálne trendy práce s dátami

Pri vytvorení novej aplikácie alebo nového software je potrebné premýšľať o niekoľko rokov dopredu, uvedomiť si koľko a akých dát bude daná aplikácia spracovávať.

V dnešnom svete je kladený stále väčší dôraz na udržateľnosť, spracovanie v reálnom čase alebo na analytické spracovanie dát. Predstavenie Internet of Things (IOT) alebo umelej inteligencie prináša pre databázové systémy úplne nové výzvy. Odpoveďou na tieto nové výzvy je kombinácia cloudových služieb a automatizácie. Je teda kladený dôraz na to aby tieto systémy dokázali rásť a pritom si zachovali výkonnosť. Nastáva ale problém vo forme vysokých nákladov na vývoj a nedostatku odborníkov v danom obore. Najväčším problémom je však licencovanie spolu s podporou databáz.[9]

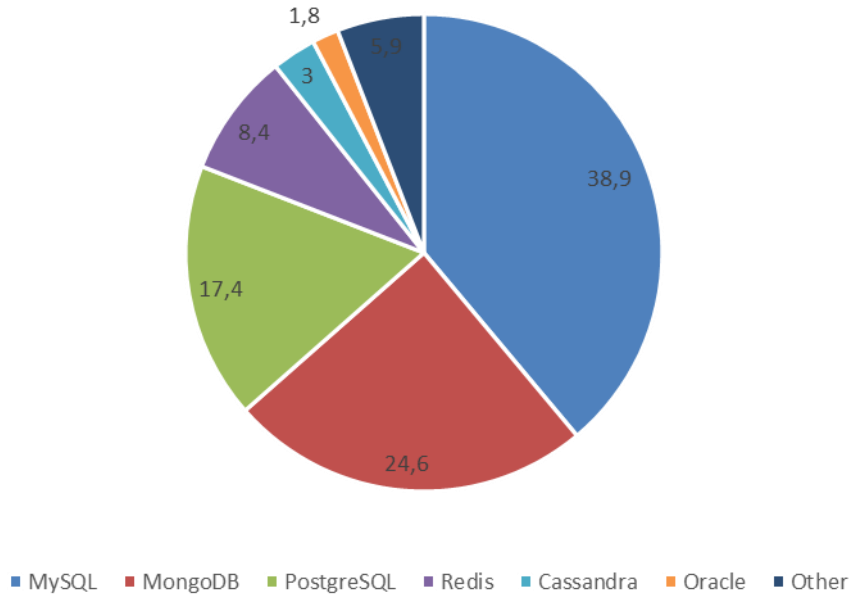
Aktuálne najpožívanejšími databázovými systémami sú stále relačné databázy, kde sa na čele stále drží databáza Oracle. Podľa DB-Engines ranking z prvých piatich databázových systémov sú prvé štyri relačné systémy. Na vrchu rebríčka sú umiestnené Oracle (1. miesto), MySQL, Microsoft SQL Server, PostgreSQL a MongoDB (5. miesto). Podľa tohto rebríčka najpoužívanejších databázových systémov sa na 116-tom mieste umiestnila prvá čisto objektová databáza a to Versant Object Database.[6]

Na druhej strane, podľa výsledkov prieskumu z DeveloperWeek-u 2019 sú tieto výsledky trochu iné. DeveloperWeek[10] je najväčšia konferencia v USA, ktorá sa koná v San Franciscu. Každý rok vo februári sa na týždeň stretne viac ako 8000 ľudí, ako sú jednotlivý vývojári, manažéri, architekti a vývojové tými z viac ako 70 krajín sveta, ktorí predstavujú nové trendy. Prieskum sa zaoberal otázkou porovnávania použitia relačných (SQL) a nerelačných (NoSQL) databáz. SQL malo dlhé roky obrovský náskok, ale ako sa zdá, NoSQL ho postupne dobieha a to hlavne vďaka populárnym databázam ako MongoDB, Casandra alebo Redis. Aj keď mnohé firmy sa rozhodli pre migráciu svojich starých databáz, ako napríklad Oracle, nie vždy si zvolia NoSQL, keďže SQL databázy používa stále 60,48%, čo znamená že NoSQL používa 39,52% (Graf 1).[11; 12]



Graf 1 Využitie SQL vs NoSQL
Zdroj: vlastné spracovanie na základe dát z prieskumu[12]

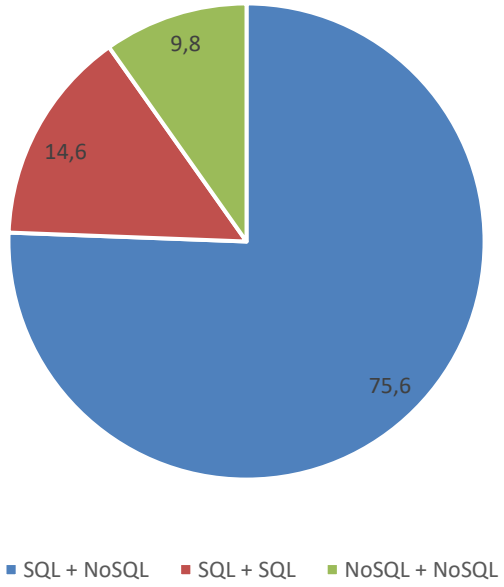
Prieskum sa ďalej zaoberal najčastejšie používanými jednotlivými databázami. A tu nastalo prekvapenie, pretože na prvom mieste sa neumiestnila databáza Oracle, ako by sa dalo predpokladať podľa rebríčka DB-Engines ranking. Jednoznačnú väčšinu získalo MySQL s 38,9%. Na druhom mieste sa umiestnilo MongoDB, ktoré veľa nezaostáva s 24,6%. Ďalšie poradie bolo naslednovné PostgreSQL s 17,4%, Redis s 8,4%, Casandra s 3%, Oracle s 1,8% a na ostatné databázy ako CouchDB, Berkeley DB, Microsoft SQL Server, Redshift ostalo spolu 5,9% (Graf 2). [12]



Graf 2 Najpopulárnejšie databázy
Zdroj: vlastné spracovanie na základe dát z prieskumu[12]

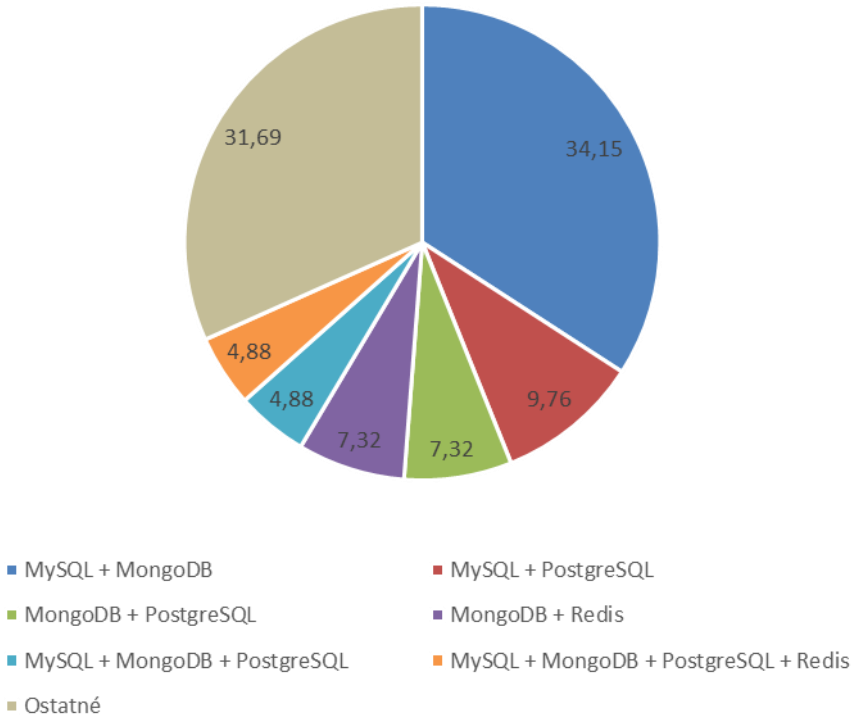
Z tohto prieskumu jednoznačne vyplýva, že MySQL, MongoDB a PostgreSQL sú na vzostupe a dá sa predpokladať, že ich použitie sa bude len zväčšovať. Prieskum ale mohol byť ovplyvnený aj malou účasťou užívateľov databázy Oracle.

V posledných rokoch sa taktiež zvyšuje trend prepojenia viacerých databázových systémov. Z tohto dôvodu bola v prieskume otázka ohľadom využitia Single Database vs Multi-Database. Už takmer polovica firiem používa vo svojich aplikáciách viac databázových systémov naraz. Konkrétne 44,3% z čoho vyplýva, že 55,7% firiem stále používa jedinú databázu. Zaujímavejšie sú ale kombinácie týchto databáz. Žiadnym prekvapením nie je dominancia kombinácie SQL + NoSQL, ktorá má 75,6%. Kombinácie SQL + SQL majú 14,6% a NoSQL + NoSQL má 9,8% (Graf 3).[12]



Graf 3 Využitie kombinácií SQL a NoSQL databáz
Zdroj: vlastné spracovanie na základe dát z prieskumu[12]

Čo sa týka konkrétnych používaných databáz, najpoužívanejšia kombinácia je MySQL a MongoDB. Mongo je síce považované za oponenta MySQL, ale ak je databázová štruktúra dobre navrhnutá, spolupracujú tie dve databázy veľmi dobre. Túto kombináciu volí až 34,15% opýtaných. Druhá najpopulárnejšia voľba je MySQL spojená s PostgreSQL. Táto kombinácia je prekvapením, pretože tieto SQL databázy sú priamymi konkurentmi na trhu, môžu byť ale použité spolu na ukladanie rôznych súborov dát. Táto kombinácia mala 9,76% z čoho vyplýva, že je to najpoužívanejšia kombinácia SQL a SQL databázy. Na tretej pozícii skončili kombinácie MongoDB s PostgreSQL a MongoDB s Redis, ktoré mali rovnaký výsledok 7,32% (Graf 4).[11; 12]



Graf 4 Najčastejšie kombinácie databáz
Zdroj: vlastné spracovanie na základe dát z prieskumu[12]

Z týchto prieskumov vychádza to, že objektové databázy sa používajú veľmi málo, hlavne v porovnaní s inými NoSQL databázovými systémami. Jedna z hlavných nevýhod OODB je jej príliš vysoká vývojová cena. Taktiež existuje mnoho firiem ktoré majú rozsiahle SQL databázy, ktorým sa neoplatí prejsť na objektovú databázu, keď majú dostupné lepšie varianty v NoSQL databázach.[13] Väčšina firiem nepotrebuje ukladať metódy, ktoré sa pri použití OODB ukladajú spoločne s dátami v atribútoch daného objektu. Preto je pre vývojára jednoduchšie uložiť si napríklad jeden dokument vo formáte JSON a potom pri načítaní dát do rekonštruovaného (deserialize) daného objektu. V dnešnej dobe sú už tiež dostupné prostriedky na mapovanie objektov do relačných databáz, ktoré odvedú väčšinu práce za vývojára. Najpopulárnejším je Hibernate[14].

5 Predstavenie BTDB

BTDB je produkčná open-sourcová objektová databáza vyvinutá vo firme GMC (v súčasnosti Quadient) predovšetkým pre ich potreby. BTDB je voľne dostupná napríklad na https://github.com/Bobris/BTDB. BTDB je celá napísaná v C# a preto je vhodná na prácu, kde sa používa primárne programovací jazyk C#. V tejto kapitole boli použité informácie predovšetkým zo zdroja [15].

5.1 Základné charakteristiky BTDB

BTDB využíva Key Value Store, ktorý je napísaný v C#, bez použitia natívneho kódu. Jedno úložisko je v podstate jeden priečinok. Má podporu ACID[1] spolu s MVCC[2] a to robí z BTDB plne funkčnú databázu. Naraz podporuje mnoho read-only transakcií, ale len jednu read-write transakciu, čo umožňuje prácu pre viac užívateľov naraz. V tejto databáze je možné importovať alebo exportovať streamy. Taktiež je možné prispôsobiť kompresiu podľa potreby. Relatívne rýchlo sa načítava aj keď je stále potrebné načítať do pamäte všetky kľúče. Preto je maximálna dĺžka kľúča a hodnoty obmedzená na 31 bitov. BTDB je postavená ako binárny strom.

5.2 Funkcie BTDB

Ukladá čisté .NET objekty a iba ich verejné vlastnosti s gettermi a settermi. Všetky ACID a MVCC vlastnosti sú ponechané. Ďalšími funkciami sú automatická aktualizácia modelu pri načítacej operácii s dynamicky generovaným optimálnym IL kódom, automatické verzovanie zmien v modeloch a enumerácia všetkých objektov. Každý objektový typ je možné uložiť pomocou „singletonu“, a to umožňuje jednoduché mapovanie ku koreňovému objektu. Objekty sú ukladané defaultne, inline s rodičovským objektom a je možné použiť IIndirect pre objekty s OID čo ale spôsobí spomalenie databázy.

---

[1] ACID – Atomicity, Consistency, Isolation, Durability. Definícia dostupná z https://database.guide/what-is-acid-in-databases/

[2] MVCC – MiltiVersion Concurrency Control. Definícia dostupná z: https://docs.keydb.dev/docs/pro-mvcc/

5.3 Dto Channel a ukladanie udalostí

V staršej verzii BTDB bola používaná RPC knižnica, ktorá ale bola nahradená Dto Channelom, ktorý teraz zaisťuje zasielanie a prijímanie správ Dto pomocou Tcp/Ip. To pomohlo zvýšeniu rýchlosti Tcp/Ip kanálov. BTDB umožňuje optimálnu serializáciu pre metadata. Ukladanie je formou transakcií a podporuje technológiu Azure Page Blobs[3].

[3] Azure Blob Page definícia a dokumentácia dostupná z https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-pageblob-overview

6 Ukážky použitia BTDB

Táto kapitola je zameraná na ukážku použitia jednotlivých funkcii BTDB. Vytvorí sa tabuľka alebo Relácia v poňatí BTDB a budú zobrazené ukážky práce s BTDB. Bude ukázané vytvorenie jednoduchšieho a rýchlejšieho záznamu v BTDB, ktorý je pomenovaný Singleton. Tento záznam sa vytvára alebo načíta len raz a následne už ostáva v pamäti. Toto všetko vyššie spomínané nie je možné bez založenia BTDB, takže bude zobrazený postup vytvorenia novej databáze a jej prvej transakcie. Táto transakcia je potom využitá pri ukážke jednotlivých funkcií BTDB.

6.1 Ukážka vytvorenia BTDB

Ak je potrebné do BTDB niečo uložiť musí byť najprv vytvorená. Na to sú potrebné hlavne dva objekty: IKeyValueDB a IObjectDB. Po vytvorení týchto dvoch objektov je možné databázu otvoriť. Potom už len stačí vytvoriť transakciu a je možné začať pracovať s obsahom BTDB (Kód 1). Ak je už vytvorená BTDB, ktorá je uložená v súboroch s vlastnou príponou .trl, je možné ju načítať, otvoriť a pracovať s ňou. Jediný rozdiel je v tom, že do KeyValueDB je potrebné vložiť iný objekt (Kód 2).[16]

```
IKeyValueDB kvDb = new InMemoryKeyValueDB();
IObjectDB db = new ObjectDB();
db.Open(kvDb, false);
IObjectDBTransaction tr = db.StartTransaction();

tr.Dispose();
db.Dispose();
```

Kód 1 Vytvorenie novej BTDB databázy
Zdroj: vlastné spracovanie

```
string directory = @"C:\";
var kvDB = new KeyValueDB(new OnDiskFileCollection(directory));
var db = new ObjectDB();
db.Open(kvDB, false);
IObjectDBTransaction tr = db.StartTransaction();

tr.Dispose();
db.Dispose();
```

Kód 2 Otvorenie existujúcej databázy
Zdroj: vlastné spracovanie

6.2 Ukážka použitia usporiadaného slovníka

Nižšie v Kóde 3 je znázornené jednoduché vytvorenie a uloženie usporiadaného slovníka (Ordered Dictionary) do databázy. Tiež sú tam zobrazené niektoré základné funkcie, ktoré je možné nad usporiadaným slovníkom urobiť, ako napríklad pridanie a odstránenie dát, prejdenie všetkých dát a ich následný výpis do konzoly alebo použite enumerátora na nájdenie potrebných dát.[16]

```
using (var tr = db.StartTransaction())
{
       var root = tr.Singleton<Root>();
       var dict = root.Id2User;

       dict.Add(1, new User { Name = "Matus1", Age = 24 });

       foreach (var user in dict)
       {
       Console.WriteLine("Id: " + user.Key + ", Name: " +
       user.Value.Name + ", Age: " + user.Value.Age);
       }

       dict.RemoveRange(2, true, 5, false);

       var enumerator = dict.GetAdvancedEnumerator());
       Console.WriteLine("Total count of users: " + enumerator.Count);
}
```

Kód 3 Práca so slovníkom
Zdroj: spracované na základe dokumentácie k BTDB [16]

6.3 Ukážka práce s reláciami

Na vytvorenie relácie, je potrebné v prvom rade vytvoriť štruktúru. Na ukážke Kód 4 je vytvorená trieda osoba a je tam deklarované čo bude použité ako pole primárnych kľúčov, v tomto prípade je to id a Name. Ako sekundárny kľúč je zvolený Age. Ďalším krokom je vytvorenie rozhrania tabuľky, v ktorom sú potrebné funkcie. V tejto podkapitole vychádzam zo zdroja [17].

```
public class Person
{
	[PrimaryKey(1)]
	public ulong Id { get; set; }
	[SecondaryKey("Name")]
	public string Name { get; set; }
	[SecondaryKey("Age")]
	 public int Age { get; set; }
}

public interface IPersonTable : IReadOnlyCollection<Person>
{
	void Insert(Person person);
	bool RemoveById(ulong id);
	Person FindById(ulong id);
	void Update(Person person);
	bool Upsert(Person person);
	bool Contains(ulong id);
	Person FindByAgeOrDefault(int age);
	IEnumerator<Person> FindByAge(int age);
	IEnumerator<Person> ListByAge(AdvancedEnumeratorParam<int> param);
	IOrderedDictionaryEnumerator<ulong, Person>
	ListById(AdvancedEnumeratorParam<ulong> param);
}
```

Kód 4 Ukážka jednoduchej štruktúry dát a vytvorenie tabuľky
Zdroj: spracované na základe dokumentácie k BTDB [17]

Potom ako je hotová dátová štruktúra, je potrebné ďalej vytvoriť creator, ktorý poslúži na vytváranie relácií. Pomocou neho je možné ukladať v tomto prípade osoby do tabuľky osôb (Kód 5).

```
Func<IObjectDBTransaction, IPersonTable> creator;
using (var tr = db.StartTransaction())
{
	creator = tr.InitRelation<IPersonTable>("Persons");
	var personTable = creator(tr);
	personTable.Insert(new Person { Id = 1, Name = "Matus", Age = 24 });
	tr.Commit();
}
```

Kód 5 Pridanie dát do tabuľky
Zdroj: spracované na základe dokumentácie k BTDB [17]

Creator sa dá znovu použiť, teda nie je potrebné ho vždy znova vytvárať. Ďalšie funkcie databázy je potrebné definovať v rozhraní, v tomto prípade to bolo Insert, RemoveById, FindById, Update a Upsert. BTDB ale podporuje aj mnoho iných metód:

- **Insert**

  Funkcia insert vloží záznam do tabuľky a vráti hodnotu true, ak bol záznam pridaný a vráti hodnotu false ak už záznam existuje. Pri použití void pridá záznam, ak by záznam už existoval vráti výnimku. Častejšie používaná je metóda void (Kód 6).

```
void Insert(Person person);
bool Insert(Person person);

personTable.Insert(new Person { Id = 1, Name = "Matus", Age = 24 });
```

Kód 6 Funkcia Insert
Zdroj: vlastné spracovanie

- **Update**

  Funkcia Update upraví záznam v tabuľke. Vyhodí výnimku ak záznam neexistuje (Kód 7).

```
void Update(Person person);

personTable.Update(new Person { Id = 1, Name = "Matus", Age = 25 });
```

Kód 7 Funkcia Update
Zdroj: vlastné spracovanie

- **Upsert**

  Funkcia Upsert vytvorí nový záznam, ak už ale záznam existoval tak ho upraví. Funkcia je používanejšia ako Insert (Kód 8).

```
bool Upsert(Person person);

personTable.Upsert(new Person { Id = 2, Name = "Nick", Age = 22 });
personTable.Upsert(new Person { Id = 2, Name = "Nick", Age = 56 });
```

Kód 8 Funkcia Upsert
Zdroj: vlastné spracovanie

- **Shallow update / upsert**

  Je podobná funkcia ako Update / Upsert, ale nesnaží sa porovnať a uvoľniť zanorený obsah. Je znateľne rýchlejší bez sekundárnych indexov hlavne preto, že nemusí čítať staré hodnoty.

- **Remove**

  Funkcia Remove vracia hodnotu true, ak bol záznam odstránený. Pri použití variantu void vyhodí výnimku ak záznam neexistuje. Primárne kľúče sú použité ako parametre. Vracia počet vymazaných záznamov s prefixom daných primárnych kľúčov a odstráni všetkých užívateľov s prefixom (Kód 9). Existuje aj ShallowRemove, ten ale neuvoľni obsah.

```
bool RemoveById(ulong id);
void RemoveById(ulong id);
int RemoveById(ulong id);
int RemoveByIdPartial(ulong id, ulong maxCount);personTable.RemoveById(1);

personTable.RemoveById(1);
personTable.RemoveById(1, 2, 3);
personTable.RemoveByIdPartial(1, 5);
```

Kód 9 Funkcia Remove
Zdroj: vlastné spracovanie

- **Contains**

  Funkcia Contains vracia true, ak bola položka s primárnym kľúčom nájdená. Môžeme použiť viac primárnych kľúčov ako parametre (Kód 10).

```
bool Contains(ulong id);
bool Contains(ulong id1, ulong id2, ulong id3);

personTable.Contains(1);
personTable.Contains(1, 2, 3);
```

Kód 10 Funkcia Contains
Zdroj: vlastné spracovanie

- **Find**

  Funkcia Find vráti nájdený záznam alebo vyhodí výnimku ak záznam neexistuje. Ako parameter očakáva primárny kľúč rovnako ako RemoveById. Pri použití default vráti nájdenú hodnotu alebo hodnotu čo je prednastavená, ak záznam neexistuje vráti null. Pri použití Enumerator-a môžeme hľadať viac záznamov naraz. Nájde všetky záznamy s prefixom primárneho kľúča. Vyhľadávame pomocou primárneho kľúča alebo sekundárnych kľúčov (Kód 11).

```
Person FindById(ulong id);
Person FindByIdOrDefault(ulong id);
IEnumerator<Person> FindById(ulong id);
Person FindByAgeOrDefault(int age);
IEnumerator<Person> FindByAge(int age);

personTable.FindById(1);
personTable.FindByIdOrDefault(1);
personTable.FindByAge(15);
personTable.FindByAgeOrDefault(24);
```

Kód 11Funkcia Find
Zdroj: vlastné spracovanie

- **List**

  Funkcia List umožňuje vzostupné alebo zostupné poradie a rozsah záznamov. Čiastočné polia sú brané do úvahy pomocou primárnych kľúčov. Kód 12 je príklad využitia listu kde môžeme listovať všetkými izbami alebo iba izbami danej spoločnosti.

```
public class Room
{
	[PrimaryKey(1)]
	public ulong CompanyId { get; set; }
	[PrimaryKey(2)]
	public ulong Id { get; set; }
	public string Name { get; set; }
}

public interface IRoomTable : IReadOnlyCollection<Room>
{
	void Upsert(Room room);
	IOrderedDictionaryEnumerator<ulong, Room>
	ListById(AdvancedEnumeratorParam<ulong> param);
	IOrderedDictionaryEnumerator<ulong, Person> ListById(ulong companyId,
	AdvancedEnumeratorParam<ulong> param);
}
```

Kód 12 Funkcia list
Zdroj: spracované na základe dokumentácie k BTDB [17]

- **Count**

  Funkcia Count rýchlo vracia počet výskytov položiek uložených v danej tabuľke (Kód 13). Okrem int môžeme použiť aj uint, long alebo ulong. Je možné použiť aj iný parameter ako primárny kľúč a to pomocou Enumerátora.

```
int CountById(ulong id);
int CountById(AdvancedEnumeratorParam<ulong> param);
```

Kód 13 Funkcia Count
Zdroj: vlastné spracovanie

- **Enumerate**

  Funkcia Enumerate spočítava všetky záznamy podľa primárneho kľúča. Momentálna hodnota nie je braná do úvahy ak je časťou poľa v rozhraní tabuľky, inak je braná každá ďalšia hodnota, ktorá je spočítavaná. Spočítané sú vždy všetky záznamy, ale je možné vynechať časti poľa (Kód 14).

```
IEnumerator<Person> GetEnumerator();
```

Kód 14 Funkcia Enumerate
Zdroj: vlastné spracovanie

7 Analýza a návrh aplikácie pre vizualizáciu BTDB

Cieľom práce je vytvorenie aplikácie na vizualizáciu dát z ľubovoľnej BTDB databázy. Táto aplikácia má byť schopná ukázať čo všetko daná databáza obsahuje vo forme jednoduchých tabuliek, kde je zobrazená štruktúra dát vo formáte meno, typ a hodnota daného objektu. Prístup k dátam bude prostredníctvom cesty k .trl súborom danej BTDB databázy. Aplikácia má bežať na serveri, ktorý bude mať prístup k daným súborom a tým pádom sa jedná o webovú aplikáciu.

Presné zadanie od firmy Quadient:
K priechodu databázou treba použiť ODBIterator, použitie sa nachádza v ObjectDBTest. Ako testovacie dáta vložte do BTDB tieto typy:

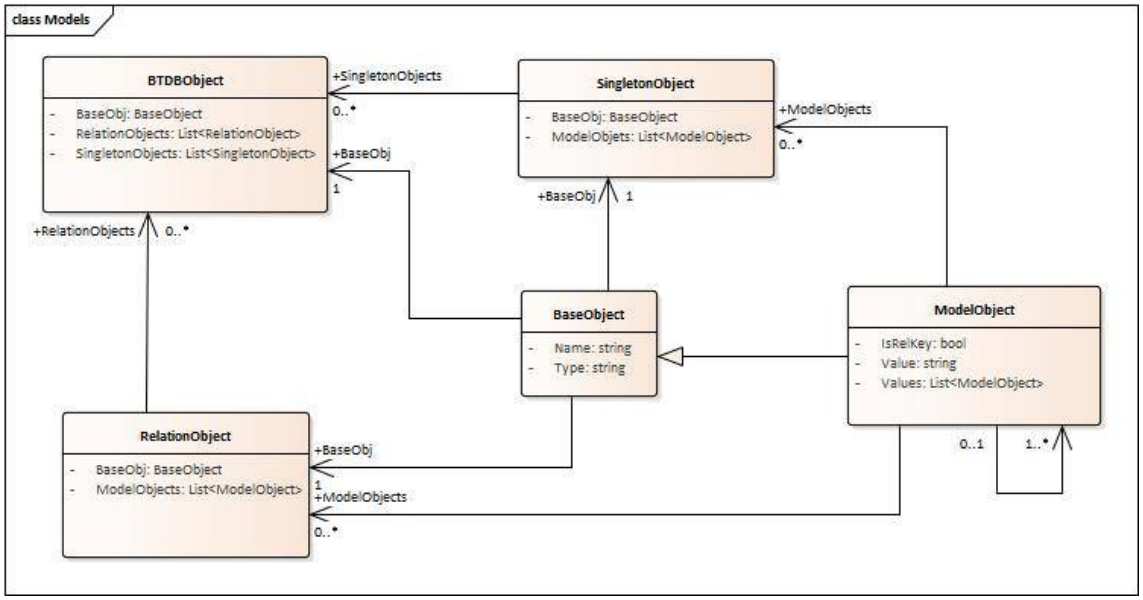
1. Objekt (vlastný objekt CustObj) s propertami typu:
   - string
   - ulong
   - int
   - enum
   - IList<>
   - IDictionary<>
   - Byte[]
2. IList<string>
3. IList<int>
4. IList<CustObj>
5. IDictionary<ulong, string>
6. IDictionary<ulong, CustObj>
7. IDictionary<DateTime, CustObj>
8. IDictionary<KeyObj, CustObj>

   KeyObj :
   - DateTime
   - ulong
   - Enum

Ako prvé bolo potrebné vytvoriť dané testovacie dáta, potom ako bola v minulej kapitole ukázaná práca s BTDB. Bolo potrebné vytvoriť CustObj a k nemu potrebnú tabuľku do ktorej by sme ukladali záznamy ICustObjTable. Všetky ostatné záznamy bolo možné dať do jednotlivých singletonov podľa zadania.

Ako druhé bolo potrebné prejsť OBDIterator, ako funguje a čo je jeho vstup a výstup. IOBDIterator je rozhranie ktoré poskytuje metódy potrebné na prechádzanie obsahu BTDB databázy. V ODBIteratorTest je vytvorený ToStringVisitor z ktorého je možné čerpať. Problém je ale v tom, že jeho návratový typ je string. V tomto konkrétnom prípade to nevyhovuje, pretože je potrebný návratový typ, ktorý bude koreňovým objektom.

Ako tretie bolo potrebné vymyslieť koreňový objekt ktorý by obsahoval niečo ako stromovú štruktúru objektov v danej databáze. Tento objekt bol vytvorený a nazvaný BTDBObject. Tento objekt obsahuje meno a typ, ale predovšetkým dva listy objektov typu SingletonObject a RelationObject. Oba tieto objekty sú si veľmi podobné v tom, že uchovávajú dáta o jednotlivých reláciách a singletonoch v danej databázy. Obsahujú meno, svoj typ a list objektov typu ModelObject. ModelObject je objekt ktorý v sebe obsahuje dáta, ktoré chceme zobrazovať, konkrétne meno, typ a hodnotu. Ak ju nemá, znamená to, že ide o vnorený objekt. Preto obsahuje ModelObject taktiež list modelových objektov. Pre zjednodušenie bol vytvorený BaseObject, ktorý obsahuje property Name a Type. Z tohto objektu dedí ModelObject a všetky ostatné objekty tj., SingletonObject, RelationObject a BTDBObject, ho majú ako property pomocou ktorej sa dá vyhľadávať.



Obrázok 1 UML Class Diagram
Zdroj: vlastné spracovanie

Teraz, keď bola vytvorená štruktúra objektov do ktorých je možné vkladať potrebné dáta, je potrebné vytvoriť vlastný visitor, v tomto prípade dva visitori, pretože veľkosť koreňového objektu by bola priveľká. Odhadovaná veľkosť databázy je približne 50GB, čo je pomerne veľká záťaž. BaseDataVisitor obsahuje BTDBObject a v ktorom nie sú vyplnené všetky metódy. V tomto visitore sú prevzaté len základné informácie. Konkrétne list SingletonObjectov a list RelationObjectov, ale iba s ich menami a typmi. Myšlienka je taká, že z týchto informácií bude vytvorená prvá tabuľka, ktorá bude obsahovať len mená, typy a odkaz na daný objekt. Tento objekt či už je to SingletonObject alebo RelationObject sa vyhľadáva práve pomocou mena a typu. Pri druhom Visitore už budú vyplnené všetky potrebné metódy a budú získané potrebné dáta z databázy. Visitor vracia list typu ModelObject. V tomto liste sú všetky singletony a relationy spolu s ich obsahom.

Obsah databázy už je k dispozícii ale zatiaľ nie je možné ho zobraziť. Na zobrazenie bol zvolený postup vytvorenia webovej aplikácie v ASP.NET Core, čo malo mnoho výhod. Jedna z hlavných výhod je použitie návrhového vzoru MVC[4].

[4] Návrhový vzor MVC (Model View Controller), definícia je dostupná na https://techterms.com/definition/mvc

Ďalšie výhody sú napríklad to, že model alebo skôr modely, už boli pripravené stačilo ich len použiť. Bolo by potrebné vytvoriť dva kontrolery konkrétne HomeController a DataController. HomeController bol jednoduchší, pretože je v ňom potrebné len to, aby sa v ňom zobrazovala a zapisovala cesta k súborom databázy. DataController bol zložitejší, pretože tu sa prenášajú všetky údaje z databázy a posielajú sa do jednotlivých pohľadov. Najprv bolo potrebné nájsť spôsob ako sa tieto dáta dostanú do kontroleru, kde sa s nimi dá ďalej pracovať. Nato bol vytvorený priečinok Services, kde sú jednotlivé services a ich rozhrania uložené. Podľa zadania bola vytvorená dll knižnica, ktorá obsahuje práve tieto services, modely a visitory popísané vyššie. Táto knižnica dostala názov BTDBPart. Boli potrebné presne tri services, jednu na dodanie základných dát, ktorá dostala názov BaseDataService. Rozhranie IBaseDataService obsahuje presne jednu metódu a to IterateDB s návratovým typom BTDBObject. Ďalšie dve services boli podobné až nato, že vracali buď SingletonObject alebo RelationObject, preto dostali názov SpecificSingletonDataService a SpecificRelationDataService. Keď boli tieto services vytvorené mohli byť použité v DataControllery.

Posledným bodom bolo už len vytvorenie jednotlivých pohľadov, ktoré by boli schopné zobraziť tabuľky s potrebnými dátami.

8 Implementácia návrhu aplikácie

8.1 Vytvorenie skúšobných dát

Bolo potrebné vytvoriť danú testovaciu BTDB databázu a na to bola vytvorená jednoduchá konzolová databáza. Z návrhu vyplynula potreba vytvoriť CustObj a k nemu potrebnú tabuľku do ktorej by bolo možné ukladať záznamy ICustObjTable. CustObj bol premenovaný na User a tabuľka na IUserTable aby to bolo jednoduchšie pre prezentačné účely (Kód 15).

```
public enum Gender
{
	Male,
	Female
}

public class User
{
	[PrimaryKey(1)]
	public ulong UserId { get; set; }
	[SecondaryKey("Name")]
	public string Name { get; set; }
	[SecondaryKey("Age")]
	public int Age { get; set; }
	public Gender Gender { get; set; }
	public IList<string> Addresses { get; set; }
	public IDictionary<string, int> NameToAge { get; set; }
	public byte[] ByteArray { get; set; }
}

public interface IUserTable : IReadOnlyCollection<User>
{
	void Insert(User customObj);
	bool RemoveById(ulong userId);
	User FindById(ulong userId);
	bool Contains(ulong userId);
}
```

Kód 15 Vytvorenie tabuľky
Zdroj: vlastné spracovanie

Následné vloženie záznamov do tabuľky bolo tiež bez problémov (Kód 16).

```
Func<IObjectDBTransaction, IUserTable> creator;
using (var tr = db.StartTransaction())
{
	creator = tr.InitRelation<IUserTable>("UserTable");
	var customObjTable = creator(tr);

	customObjTable.Insert(new User
	{
		UserId = 1,
		Name = "admin",
		Age = 100,
		Gender = Gender.Male,
		Addresses = new List<string> { "Brno" },
		ByteArray = Encoding.ASCII.GetBytes("admin")
	});
	tr.Commit();
}
```

Kód 16 Vytvorenie dát v tabuľke
Zdroj: vlastné spracovanie

Toto bola podľa zadania jediná relácia, ostatné dáta sú uložené pomocou singleton formy. Na to boli potrebné jednoduché triedy, ktoré by obsahovali dáta. V Kóde 17 je ukázané vytvorenie jednej z týchto tried.

```
public class KeyObj
{
	public DateTime DateTime { get; set; }
	public Gender Gender { get; set; }
	public ulong Id { get; set; }
}

public class KeyObjToUserClass
{
	public IDictionary<KeyObj, User> KeyObjToUser { get; set; }
}
```

Kód 17 Vytvorenie jednej triedy pre danú singleton entitu
Zdroj: vlastné spracovanie

Následne už stačilo len vytvoriť jednotlivé singletony z daných tried a vložiť ich do databázy. V Kóde18 je ukázané vloženie jednej z týchto entít.

```
// dictionary KeyObjToUserClass
var keyObjToUser = tr.Singleton<KeyObjToUserClass>();
var keyDict = keyObjToUser.KeyObjToUser;

keyDict.Add(new KeyObj
{
	DateTime = DateTime.Now,
	Gender = Gender.Male,
	Id = UInt64.MinValue
}, new User
{
	UserId = 1,
	Name = "Matus1",
	Age = 24,
	Gender = Gender.Male,
	Addresses = new List<string> {"HK", "KE"}
});
```

Kód 18 Vytvorenie jednej singleton entity
Zdroj: vlastné spracovanie

8.2 Vytvorenie ASP.NET Core aplikácie

Tento krok išiel veľmi rýchlo, vďaka šablónam od Microsoftu je to otázka pár kliknutí myšou. Teraz keď už je vytvorená defaultná aplikácia, je možné ju upraviť podľa aktuálnych potrieb. Prvým krokom je upravenie HomeControlleru. Bol potrebný objekt ktorý by držal vo svojej properte cestu k priečinku, v ktorom sa nachádzajú súbory databázy. Bola vytvorená trieda Settings, ktorá má jedinú propertu a to DirPath (Kód 19).

```
public class Settings
{
	[Required (ErrorMessage = "Path to directory is required.")]
	public string DirPath { get; set; }

	public Settings()
	{

	}

	public Settings(Settings other)
	{
		DirPath = other.DirPath;
	}
}
```

Kód 19 Vytvorenie triedy Settings
Zdroj: vlastné spracovanie

Taktiež bol potrebný mechanizmus, ktorý by do tejto triedy dokázal zapisovať a čítať z nej. To znie relatívne jednoducho až na to, že Startup, v ktorom je uložená konfigurácia celej aplikácie sa zavolá len raz, a to pri štarte aplikácie. To znamená že túto triedu je možné predať len raz. Z toho vyplýva, že ak sa prepíše cesta, tak sa znova nenačíta do ďalšieho spustenia aplikácie. Preto bol vytvorený nový WebHostBuilder. Keďže dané nastavenia je potrebné ukladať do súboru, bol zvolený formát json, pretože sa s ním jednoducho pracuje. Bola vytvorená statická trieda SettingsExtensions, ktorá má v sebe statickú metódu UseSettings() s dvoma parametrami: inštanciou na IWebHostBuilder a cestou k json súboru. Návratová hodnota je typu IWebHostBuilder, ktorej bola nastavená SettingsConfigurationSource property Path. Tiež bola pridaná jedna servica typu SettingsManager. SettingsManager potrebuje SettingsConfigurationManager a parameter ktorý sa bude sledovať, v tomto prípade to bude Settings. Ako posledný krok bolo pridané sledovanie hodnoty v Settings. Teraz keď je vytvorený WebHostBuilder stačí ho použiť v Program.cs (Kód 20).

```
public static class SettingsExtensions
{
	public static IWebHostBuilder UseSettings(this IWebHostBuilder @this,
                                                string path)
	{
	return @this.ConfigureServices((ctx, services) =>
		{
		var configuration = new ConfigurationBuilder()
			.SetBasePath(ctx.HostingEnvironment.ContentRootPath)
			.Add(new SettingsConfigurationSource {Path = path})
			.Build();

		var configurationProvider = (SettingsConfigurationProvider)
	configuration.Providers.Single();
		services.AddSingleton<ISettingsManager>(sp =>
		new SettingsManager(configurationProvider,
		sp.GetRequiredService<IOptionsMonitor<Settings>>()));

		services.Configure<Settings>(configuration);
		  });
	}
}
```

Kód 20 Vytvorenie SettingsExtension
Zdroj: vlastné spracovanie

V Kóde 20 bola použitá trieda SettingsConfigurationSource, ktorá dedí z JsonConfigurationSource a slúži na nastavenie spracovania json súboru. Metóda Build() bola prepísaná, aby vracala nový SettingsConfigurationProvider (Kód 21).

```
class SettingsConfigurationSource : JsonConfigurationSource
{
	public override IConfigurationProvider Build(IConfigurationBuilder builder)
	{
		Optional = true;
		ReloadOnChange = false;
		EnsureDefaults(builder);
		return new SettingsConfigurationProvider(this);
	}
}
```

Kód 21 Vytvorenie SettingsConfigurationSource
Zdroj: vlastné spracovanie

V Kóde 21 SettingsConfigurationProvider dedí z JsonConfigurationProvider. Je potrebné vytvoriť metódu Reload(), ktorá slúži na prečítanie dát zo súboru (Kód 22).

```
public class SettingsConfigurationProvider : JsonConfigurationProvider
{
public SettingsConfigurationProvider(JsonConfigurationSource source) : base(source)
{
}

	public void Reload()
	{
		Data = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);

		var file = Source.FileProvider?.GetFileInfo(Source.Path);
		if (file != null && file.Exists)
		using (var stream = file.CreateReadStream())
			Load(stream);

		OnReload();
	}
}
```

Kód 22 Vytvorenie SettingsConfigurationProvider
Zdroj: vlastné spracovanie

SettingsManager z Kódu 20 má metódu Update(), ktorá zabezpečuje vytvorenie alebo prepis json súboru (Kód 23). Táto metóda bude ďalej používaná v HomeControllery.

```
class SettingsManager : ISettingsManager
{
	readonly object syncLock = new object();
	readonly SettingsConfigurationProvider _configurationProvider;
	readonly IOptionsMonitor<Settings> _optionsMonitor;

	public SettingsManager(SettingsConfigurationProvider configurationProvider,
	IOptionsMonitor<Settings> optionsMonitor)
	{
		_configurationProvider = configurationProvider;
		_optionsMonitor = optionsMonitor;
	}

	public Settings Current => _optionsMonitor.CurrentValue;

	public void Update(Action<Settings> configure)
	{
		lock (syncLock)
		{
			var newOptions = new Settings(Current);
			configure(newOptions);

			var configSource = _configurationProvider.Source;
			var fileName = Path.Combine(((PhysicalFileProvider)
			configSource.FileProvider).Root, configSource.Path);
			var tempFileName = fileName + ".new";
				File.WriteAllText(tempFileName,
				JsonConvert.SerializeObject(newOptions));
			if (File.Exists(fileName))
				File.Delete(fileName);
			File.Move(tempFileName, fileName);

			_configurationProvider.Reload();
		}
	}
}
```

Kód 23 Vytvorenie SettingsManager
Zdroj: vlastné spracovanie

8.3 HomeController

Do Startup.cs bol pridaný scope na Settings, ktorý je možné používať a meniť v jednotlivých controlleroch. V tomto prípade sa upravuje Settings len v HomeControlleri, pretože tu je potrebné načítať a meniť cestu k priečinku. Metóda Index() odkazuje na pohľad kam predávame vo ViewBagu cestu k priečinku, ktorá je predaná vďaka sledovaniu Settings. Settings bol určený ako povinný parameter do konštruktoru. Taktiež je potrebná metóda Update() a tá je dostupná v SettingsManager. Domovská stránka je veľmi jednoduchá obsahuje iba formulár, v ktorom je možné vyplniť danú cestu a výpis momentálnej cesty. Vyplnený formulár posiela dáta do metódy AddPath(), kde sa skontroluje či je context v platnom stave (valid state). Ak áno, zavolá sa vyššie zmienená metóda Update(). Ak nie, je poslaný späť do pohľadu Indexu spolu so zle vyplnenými údajmi. Inak je výstupom presmerovanie na Index().

Takto bolo docielené, že počas chodu aplikácie je možné meniť jej nastavenia, konkrétne cestu k súborom databázy (Kód 24).

```
public HomeController(ISettingsManager settingsManager, Settings settings)
{
	_settingsManager = settingsManager;
	_settings = settings;
}

public IActionResult Index()
{
	ViewBag.Path = _settings.DirPath;
	return View();
}

[HttpPost]
public IActionResult AddPath(Settings settings)
{
	if (ModelState.IsValid)
	{
		_settingsManager.Update(s => s.DirPath = settings.DirPath);
	}else
	{
		return View("Index", settings);
	}
	return RedirectToAction("Index");
}
```

Kód 24 Popis HomeControlleru
Zdroj: vlastné spracovanie

8.4 DataController

DataController je zobrazovacia časť aplikácie, používajú sa tu tri services, ktoré dodávajú potrebné dáta z BTDB databázy. V prvom rade budú popísané načo slúžia dané services.

8.4.1 BaseDataService

BaseDataService slúži na dodanie BTDBObjectu, ktorý má v sebe listy SingletonObjectov a RelationObjectov. Z tohto BTDBObjectu je tým pádom možné vytvoriť prvú tabuľku. Táto tabuľka má v sebe tri stĺpce: Name, Type a Value. Name a Type získame z BaseObjectu daného Singleton/RelationObjectu. BaseDataService na vytvorenie tohto BTDBObjectu využíva metódu IterateDB(). Táto metóda si nájde json súbor kde je cesta k súborom BTDB databázy a tú cestu predá KeyValueDB, ktorý je potrebný na čítanie z BTDB. Následne je možné otvoriť danú BTDB databázu a použiť StartReadOnlyTransaction. Na prechádzanie bude použitý Iterátor. Transakcia je dostupná, ale je potrebný ešte visitor. V tomto prípade bol vytvorený BaseDataVisitor, ktorý má v sebe práve vyplnený BTDBObject (Kód 25).

```
public BtdbObject IterateDB()
{
	IKeyValueDB kvDb;
	BtdbObject btdbObject;

	var json = File.ReadAllText(@"App_Data\settings.json");
	var settings = JsonConvert.DeserializeObject<Settings>(json);

	using (var d = new OnDiskFileCollection(settings.DirPath))
	{
		kvDb = new KeyValueDB(d);
		TempDb.Open(kvDb, false);

		using (var tr = TempDb.StartReadOnlyTransaction())
		{
			var visitor = new BaseDataVisitor();
			var iterator = new ODBIterator(tr, visitor);
			iterator.Iterate();
			btdbObject = visitor.BtdbObject;
		}
	}

	return btdbObject;
}
```

Kód 25 Funkcia BaseDataService
Zdroj: vlastné spracovanie

8.4.2 SpecificSingletonDataService a SpecificRelationDataService

Pre získanie Value je potrebné použiť jednu z ďalších services. Tu už záleží na mene a type, pretože podľa toho sa vyberie daná servica. Ak je typu singleton, tak sa zavolá SpecificSingletonDataService, ak je typu relation, bude to SpecificRelationDataService. Obe tieto services sú si podobné v tom, že obe používajú rovnaký Visitor, líšia sa však v návratových typoch. SpecificSingletonDataService (Kód 26) vracia špecifický SingletonObject, ktorý bol zvolený. Preto táto servica potrebuje predať meno daného singletonu ako parameter podľa ktorého sa vyhľadá daný singleton.

```
public SingletonObject IterateDB(string name)
{
	IKeyValueDB kvDb;
	SingletonObject singletonObject = new SingletonObject();

	var json = File.ReadAllText(@"App_Data\settings.json");
	var settings = JsonConvert.DeserializeObject<Settings>(json);

	using (var d = new OnDiskFileCollection(settings.DirPath))
	{
		kvDb = new KeyValueDB(d);
		TempDb.Open(kvDb, false);

		using (var tr = TempDb.StartReadOnlyTransaction())
		{
			var visitor = new Visitor();
			var iterator = new ODBIterator(tr, visitor);
			iterator.Iterate();
			visitor.Builder.Objs.ForEach(o =>
			{
			    if (o.Name.Equals(name))
			    {
			        singletonObject.BaseObj.Name = o.Name;
			        singletonObject.BaseObj.Type = o.Type;
			        singletonObject.ModelObjects = o.Values;
			    }
			});
		}
	}

	return singletonObject;
}
```

Kód 26 Funkcia SpecificSingletonDataService
Zdroj: vlastné spracovanie

V Kóde 26 bol použitý Visitor, ktorý vytvára peknú stromovú štruktúru v ktorej sa dá celkom rýchlo vyhľadávať. Základom je predpoklad, že ModelObject sa bude vnárať do samého seba tak dlho, kým sa nenájde elementárna hodnota. Na to bol vytvorený BuilderModelObject, ktorý má jedinú propertu a tou je list ModelObjectov. Zaujímavejšie sú skôr jeho metódy. Na zapísanie základných typov, ako sú meno, typ a elementárna hodnota, existujú odpovedajúce metódy teda WriteName(), WriteType() a WriteLastValue(). Pri vytvorení daného objektu BuilderModelObject sa v konštruktore zavolá WriteValue(), kde sa do listu pridá prvý ModelObject, ktorý sa stane základom stromovej štruktúry. Metóda Down() pridáva na koniec listu nový ModelObject do ktorého sú zapisované elementárne hodnoty. Posledná metóda je Up(), ktorá slúži na presun poslednej položky listu do položky nad ňou, čím vytvára danú štruktúru. Tento BuilderModelObject je základom Visitoru, ktorý získa všetky singletony a relationy z BTDB.

Celý tento Visitor je zobrazený v Prílohe 1) na konci mojej bakalárskej práce. V skratke bude tento kód opísaný slovne. Vytvorením tohto Visitoru sa vytvorí aj BuilderModelObject do ktorého sa budú zapisovať dané dáta. Bude opísaný prechod dát jedným singletom a jednou relationou. Každý Visitor má sadu metód, ktoré je potrebné implementovať. V týchto metódach boli použité metódy BuilderModelObjectu na zápis potrebných dát. Prechod BTDB databázou vždy začína prejdením všetkých singletonov a potom nasledujú relationy.

Bol zvolený jeden z najkratších prechodov, ListOfStringClass. Prechod týmto konkrétnym singletonom je zobrazené na Schéme 1.

```
VisitSingleton()
-> StartObject()
        -> StartField()
                ->StartList()
                        -> StartItem()
                                -> ScalarAsObject()
                                -> ScalarAsText()
                        -> EndItem()
                        -> opakovanie kolobehu StartItem – EndItem pre počet položiek v liste
                -> EndList()
        -> EndField()
-> EndObject().
```

Schéma 1 Prechod singleton entitou
Zdroj: vlastné spracovanie

Toto bol jeden prechod najjednoduchším singletom. Vždy keď začínal nový objekt ako StartList alebo StartItem, tak bola použitá metóda Builder.Down(). Vždy keď bola zavolaná metóda EndItem alebo EndList bol použitý BuilderUp().

Visitor pokračuje ďalej v priechode na všetky relácie ktoré sú v danej databáze (Schéma 2).

StartRelation()

-> StartRelationKey()

-> StartField()

-> ScalarAsObject()

-> ScalarAsText()

-> EndField()

-> EndRelationKey()

-> StartRelationValue()

-> StartField()

-> ScalarAsObject()

-> ScalarAsText()

-> EndField()

-> toto sa opakuje pre všetky elementárne property daného objektu – StartField() -> EndField()

-> StartList() – opakuje sa ako v prípade singletonu ->EndList()

-> StartField()

-> StartDictionary()

-> StartDictKey()

-> ScalarAsObject()

-> ScalarAsText()

-> EndDictKey()

-> StartDictValue()

-> StartField()

-> ScalarAsObject()

-> ScalarAsText()

-> EndField()

->EndDictValue()

-> EndDictionary()

-> EndRelationValue()

-> EndRelation().

Schéma 2 Prechod relation entitou
Zdroj: vlastné spracovanie

Ak singleton obsahuje napríklad objekt, tak medzi StartField a EndField pribudne zaobalenie vo forme StartInlineObject() -> EndInlineObject(). Z tohoto je zrejmé, že prechod je pomerne priamočiary a dosť cyklický.

8.4.3 Metódy DataControlleru

Teraz keď už je možné sa dostať pomocou service k danému singletonu alebo relationu je potrebné ich zobraziť pomocou DataControlleru. Ten obsahuje tri akcie, ktoré odkazujú na jednotlivé pohľady. Základná tabuľka je zobrazená v metóde Index() DataControlleru. Po kliknutí na detail, napríklad ListOfStringClass, dôjde k presmerovaniu do ďalšej metódy SpecificData(), ktorá má tri parametre: číslo aktuálnej stránky, meno a typ. Táto akcia vracia hodnotu konkrétneho hľadaného objektu (Kód 27).

```
public IActionResult SpecificData(int? page, string name, string type)
{
	ViewBag.Name = name;
	ViewBag.Type = type;

	List = GetSpecificData(name, type);
	var pg = page ?? 1;
	var totalItems = List.Count;
	ViewBag.Pages = (int) Math.Ceiling((double) totalItems / ItemsOnPage);
	ViewBag.CurrentPage = pg;
	if (List.Count > 1)
	{
		var x = (pg - 1) * ItemsOnPage;
		if ((List.Count - x) < ItemsOnPage)
		{
			List = List.GetRange(x, List.Count - x);
		}
		else
		{
			List = List.GetRange(x, ItemsOnPage);
		}
	}

	return View(List);
}
```

Kód 27 Ukážka SpecificData akcie
Zdroj: vlastné spracovanie

V Kóde 27 je použitá metóda GetSpecificData(), ktorá vytvára json súbor s obsahom konkrétneho Singleton/RelationObjectu z dát, ktoré dostane od SpecificRelationDataService/SpecificSingletonDataService. Na rozhodnutie ktorú servicu zavolať bol použitý parameter typ (Kód 28).

```
private List<ModelObject> GetSpecificData(string name, string type)
{
	if (type.Equals("singleton"))
	{
		List = GetSingletonObject(name).ModelObjects;
		var json = JsonConvert.SerializeObject(List);
		System.IO.File.WriteAllText(_provider.Root + "JSON.json", json);
	}
	else if (type.Equals("relation"))
	{
		List = GetRelationObject(name).ModelObjects;
		var json = JsonConvert.SerializeObject(List);
		System.IO.File.WriteAllText(_provider.Root + "JSON.json", json);

		List.ForEach(o =>
		{
			if (o.IsRelKey == true)
			{
				o.Name += " (relKey)";
			}
		});
	}

	return List;
}
```

Kód 28 Ukážka metódy GetSpecificData
Zdroj: vlastné spracovanie

V tomto pohľade je tiež možné kliknúť na tlačidlo ShowJson, ktorý zobrazí json vytvorený metódou GetSpecificData, alebo DownloadJson pomocou ktorej je možné daný súbor stiahnuť. Teraz je zobrazené všetko do hĺbky jedného zanorenia daného objektu. Ak pokračuje zanorenie hlbšie, je tam znova odkaz na Detail ďalšieho objektu v poradí. Tento odkaz ho presmeruje do ďalšej akcie DetailData.

DetailData má dva parametre: číslo aktuálnej stránky a cestu k danému detailu. Táto cesta sa zanoruje tak dlho, kým sa nezobrazí elementárna hodnota (Kód 29).

```
public IActionResult DetailData(int? page, string path)
{
	ViewBag.Path = path;
	var model = FindModelObject(path);

	var pg = page ?? 1;
	var totalItems = model.Values.Count;
	ViewBag.Pages = (int) Math.Ceiling((double) totalItems / ItemsOnPage);
	ViewBag.CurrentPage = pg;

	if (model.Type != "InlineObject" && model.Type != "relValue")
	{
		var x = (pg - 1) * ItemsOnPage;
		if ((model.Values.Count - x) < ItemsOnPage)
		{
			model.Values = model.Values.GetRange(x, model.Values.Count - x);
		}
		else
		{
			model.Values = model.Values.GetRange(x, ItemsOnPage);
		}
	}

	return View(model);
}
```

Kód 29 Ukážka DetailData akcie
Zdroj: vlastné spracovanie

V Kóde 29 je použitá metóda FindModelObject, ktorá vo vytvorenom json súbore vyhľadá potrebné dáta nech sú zanorené akokoľvek hlboko v štruktúre koreňového objektu. Táto metóda ako parameter dostane cestu, ktorú následne rozdelí na list mien objektov. Prvú hodnotu v liste mien vždy priradí prvému výskytu tohto mena z json súboru. Všetky ostatné sú potom cyklicky prechádzané kým sa nenájde posledná hodnota z listu mien. Všetky mená sú jedinečné, preto je možné pomocou nich vyhľadávať. Tento pohľad teda odkazuje sám na seba, pretože sa vždy predlžuje cesta s tým, ako je potrebné ísť hlbšie do štruktúry daného koreňového objektu (Kód 30).

```
private ModelObject FindModelObject(string path)
{
	var objNames = path.Split("/").ToList();
	List = GetDataFromJson();
	var model = List.Find(x => x.Name.Contains(objNames[0]));
	if (objNames.Count > 1)
	{
		for (var i = 1; i < objNames.Count; i++)
		{
			model = model.Values.Find(x => x.Name.Contains(objNames[i]));
		}
	}

	return model;
}
```

Kód 30 Ukážka metódy FindModelObject
Zdroj: vlastné spracovanie

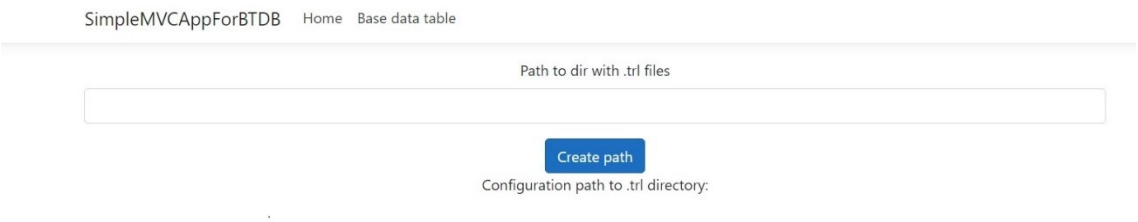
Takýmto spôsobom je možné zobraziť požadované informácie o daných objektoch ktoré sa nachádzajú v konkrétnej BTDB databáze. Zobrazovaný obsah databázy je možné jednoducho zmeniť pomocou HomeControllera, v ktorom je možné zmeniť cestu k iným súborom BTDB databázy.

9 Zhrnutie výsledkov

V tejto kapitole budú ukázané jednotlivé výstupy z webovej aplikácie.

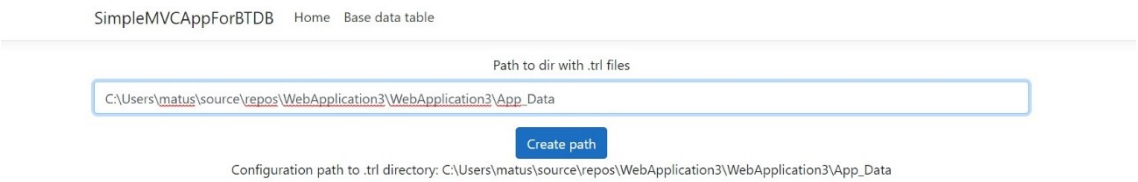
9.1 HomePage

Na HomePage (domovská stránka) je zobrazený formulár na vyplnenie cesty k súborom danej databázy a aktuálna cesta, ktorou sa k nim dostaneme. Ak pustíme aplikáciu prvý krát, keď ešte nebola zvolená žiadna cesta, tak ju nie je možné zobraziť (Obrázok 2).



Obrázok 2 HomePage pri prvom spustení aplikácie
Zdroj: vlastné spracovanie

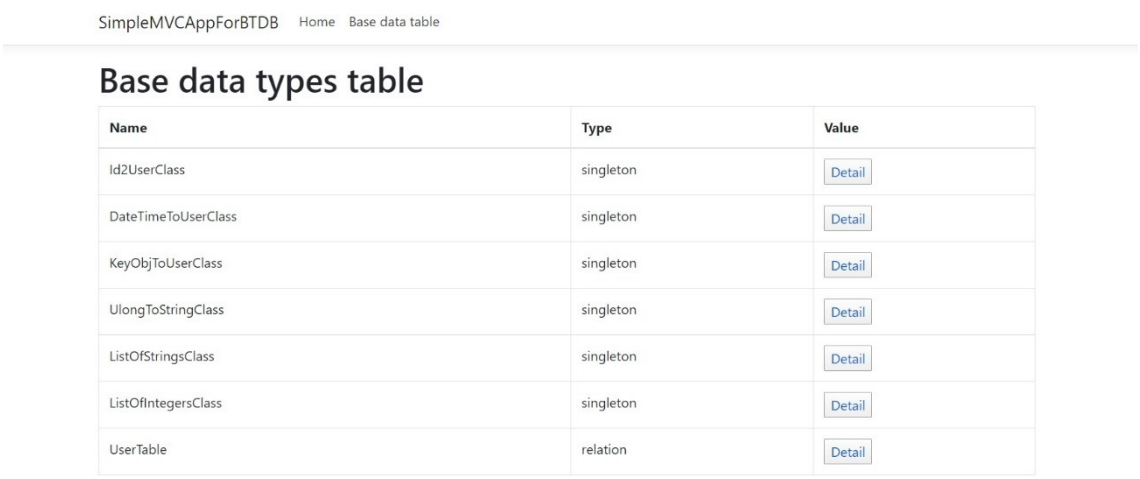
Ak ale vyplníme cestu k súborom, bude to zobrazené takto (Obrázok 3).



Obrázok 3 HomePage po vyplnení cesty k súborom databáze
Zdroj: vlastné spracovanie

9.2 Base Data Table

Po vyplnení správnej cesty je možné zobraziť prvú tabuľku kde sú všetky singleton a relation entity z danej BTDB databázy. Zobrazené je meno a typ danej entity. Kliknutím na Detail je možné zobraziť obsah zvolenej entity (Obrázok 4).



Obrázok 4 Ukážka prvej tabuľky(Base data table)
Zdroj: vlastné spracovanie

9.3 Zobrazenie zvolenej entity

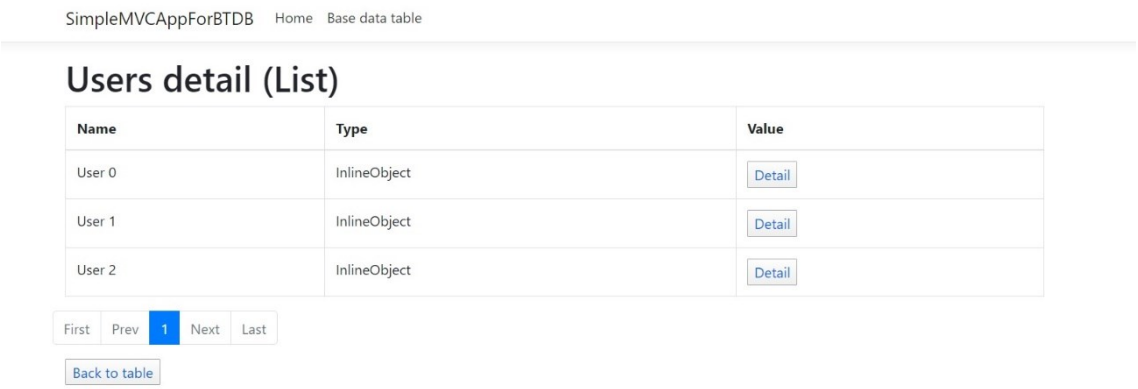
Po kliknutí na Detail zvolenej entity sa zobrazí tabuľka kde je možné vidieť obsah vo formáte Name, Type a Value. Ak Value nie je elementárna hodnota je tu presmerovanie na Detail daného objektu. Tento pohľad tiež obsahuje stránkovanie pre uľahčenie zobrazenia veľkého množstva výsledkov (Obrázok 5).

Obrázok 5 Ukážka obsahu zvolenej entity
Zdroj: vlastné spracovanie

Na tejto stránke je tiež možné zobraziť zvolenú entitu vo formáte json alebo si stiahnuť súbor v tomto formáte (Obrázok 6).

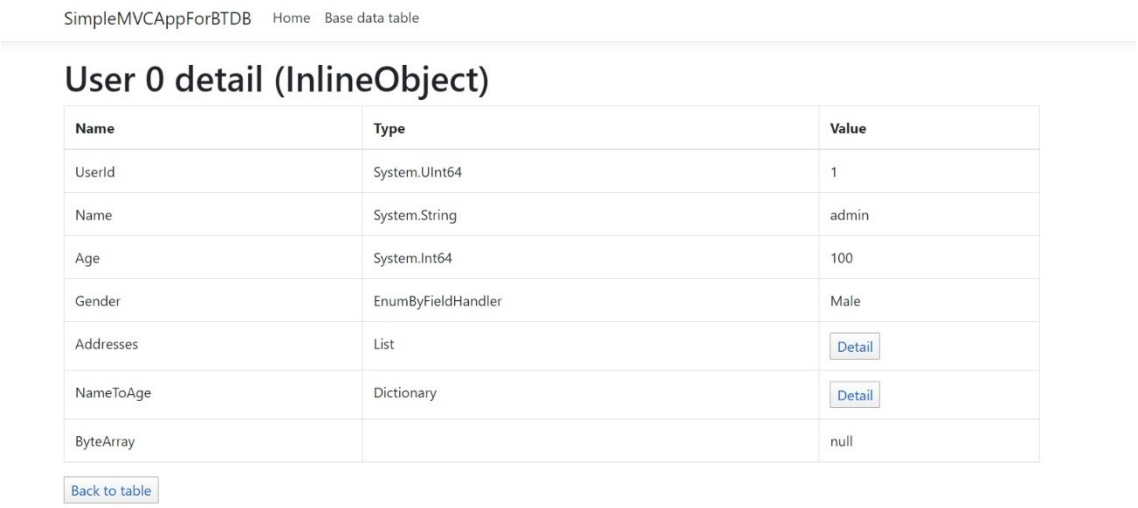
Obrázok 6 Ukážka obsahu zvolenej entity vo formáte json
Zdroj: vlastné spracovanie

Po kliknutí na Detail, je aplikácia presmerovaná na ďalšiu stránku v ktorej je zobrazený obsah zvoleného objektu danej entity, napríklad list Users. Opäť je formát tabuľky Name, Type a Value. Po kliknutí na Detail, už ale nie sme presmerovaný na inú stránku. Táto stránka slúži na zobrazenie zanorených objektov čo znamená, že jediné čo sa bude meniť, je obsah stránky (Obrázok 7).



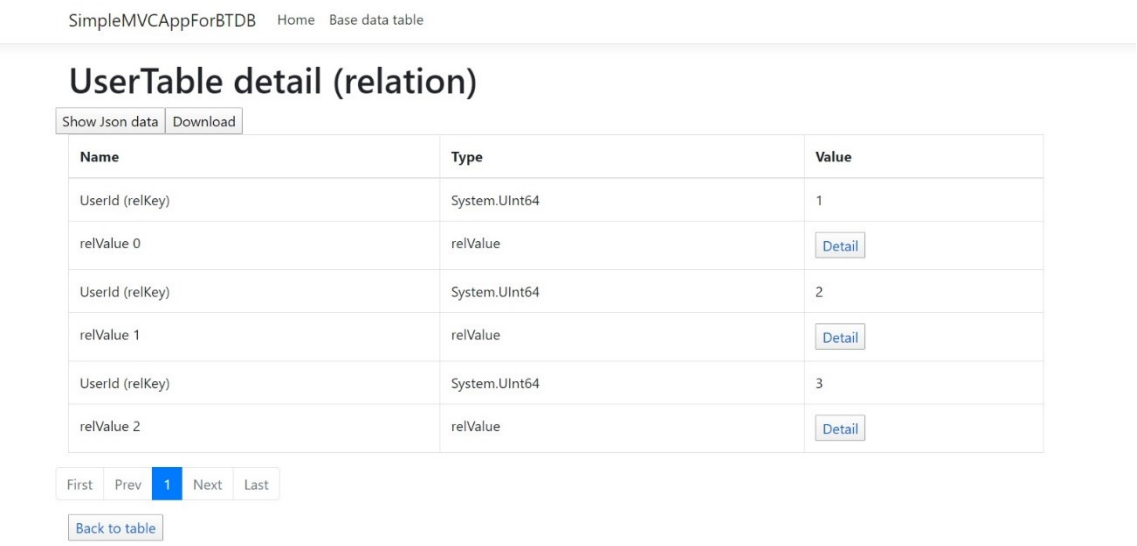
Obrázok 7 Ukážka obsahu zvoleného objektu danej entity
Zdroj: vlastné spracovanie

Po kliknutí na Detail sa zo zapísanej cesty vyberie správny obsah a ten sa prepošle stránke ktorá ho zobrazí. Napríklad klikneme na Detail User 0 (Obrázok 8).

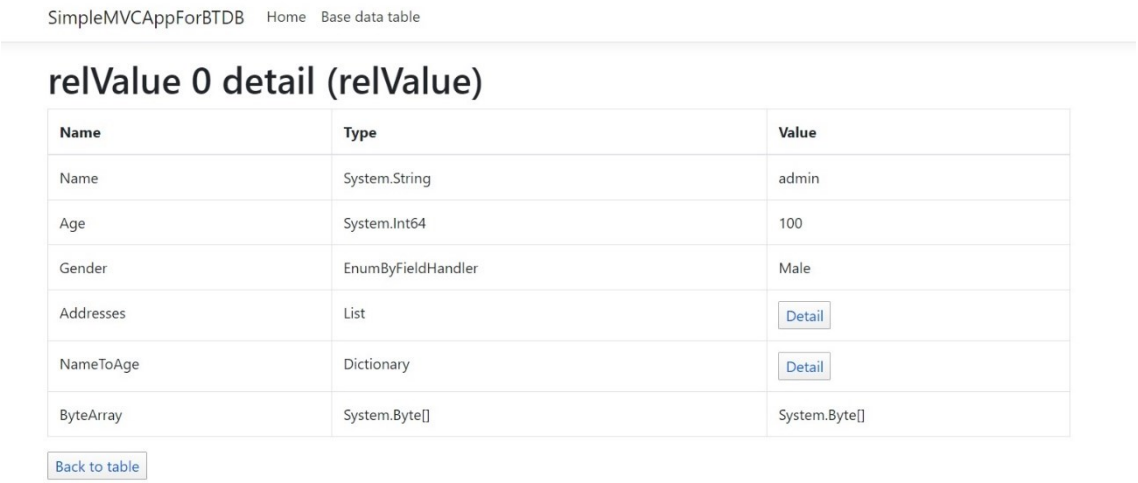


Obrázok 8 Ukážka zanoreného obsahu zvoleného objektu danej entity(inline object)
Zdroj: vlastné spracovanie

Takto je možné pokračovať až kým sa nezobrazia elementárne hodnoty ako na Obrázku 8. Ak zvolíme relation entitu je to veľmi podobné. Jej ukážka je na Obrázku 9 a 10.



Obrázok 9 Ukážka relation entity
Zdroj: vlastné spracovanie



Obrázok 10 Ukážka zanoreného obsahu zvoleného objektu danej entity (relation value)
Zdroj: vlastné spracovanie

V tejto práci som ukázal jeden z možných spôsobov ako vytvoriť potrebnú aplikáciu na zobrazovanie obsahu BTDB databázy. Netvrdím, že to nejde inak alebo inou formou. Moja aplikácia je ale funkčná a splnila svoj cieľ, ktorý bol zadaný. Umožňuje zobraziť všetko čo sa v danej databáze nachádza a zobrazuje to v požadovanom formáte. Taktiež je možné ju rozširovať o ďalšiu funkcionalitu, ak to bude v budúcnosti potrebné. Ak by napríklad Quadient chcel, môže použiť priloženú BTDBPart.dll knižnicu a vytvoriť si v ich infraštruktúre API a na zobrazenie si spraviť vlastné tabuľky pomocou Bobrilu[5].

[5] Bobril je webový framework podobný Reactu vytvorený Borisom Lechotom pre potreby firmy Quadient.

10 Záver

Táto záverečná práca sa teoreticky venovala porovnaniu relačných a objektových, NoSQL a objektových databáz. Povedala niečo o nových trendoch v oblasti používania databáz. Jej praktická časť bola zameraná na predstavenie BTDB objektovej databázy, prechádzanie jej obsahu a následné zobrazenie daného obsahu. Výsledkom je webová aplikácia, ktorá môže byť nasadená na server kde je uložená BTDB databáza pre jej zobrazenie.

11 Zoznam použitej literatúry

[1] WebSupport. *ČO JE TO DATABÁZA?* [online] Bratislava: WebSupport; [cit. 19.12.2019] Prevzaté z https://www.websupport.sk/faq/co-je-to-databaza

[2] Chand, M. *What Are NoSQL Databases* [online] Philadelphia: C# Corner; [cit. 24.1.2020] Prevzaté z https://www.c-sharpcorner.com/article/what-is-a-nosql-database/

[3] SearchSQLServer. *DEFINITION database (DB)* [online] Newton: SearchSQLServer c/o TechTarget; [cit. 19.12.2019] Prevzaté z https://searchsqlserver.techtarget.com/definition/database

[4] Rutija S. Ghongade, P.J.P. Comparison of Relational Database and Object OrientedDatabase. International Journal of Modern Trends in Engineering and Research (IJMTER).01(05):27-33.

[5] Imanuel. *TOP 9 OBJECT DATABESES* [online] Newton: PAT RESEARCH; [cit. 19.12.2019] Prevzaté z https://www.predictiveanalyticstoday.com/top-object-databases/

[6] DB-Engines. *DB-Engines Ranking* [online] Viedeň: DB-Engines; [cit. 19.12.2019] Prevzaté z https://db-engines.com/en/ranking

[7] Chand, M. *What Is A Document Database* [online] Philadelphia: C# Corner; [cit. 24.01.2020] Prevzaté z https://www.c-sharpcorner.com/article/what-is-a-document-database/

[8] MongoDB. *NoSQL Databases Explained* [online] Dublin: MongoDB, Inc.; [cit. 24.01.2020] Prevzaté z https://www.mongodb.com/nosql-explained

[9] McKendrick, J. *Cloud Opens the Path to Database Expansion* [online] New Providence: Database Trends and Applications; [cit. 19.12.2019] Prevzaté z http://www.dbta.com/Editorial/News-Flashes/Cloud-Opens-the-Path-to-Database-Expansion-128869.aspx

[10] DeveloperWeek. *About* [online] San Francisco: DeveloperWeek; [cit. 19.12.2019] Prevzaté z https://www.developerweek.com/about/

[11] Ramel, D. *Database Trends Report: SQL Beats NoSQL, MySQL Most Popular* [online] Chatsworth: APPLICATION DEVELOPMENT TRENDS; [cit. 19.12.2019] Prevzaté z https://adtmag.com/articles/2019/03/05/db-report.aspx

[12] ScaleGrid. *2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use* [online] Seattle: ScaleGrid; [cit. 19.12.2019] Prevzaté z https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/

[13] Panwar, A. *Types Of Database Management Systems* [online] Philadelphia: C# Corner; [cit. 24.01.2020] Prevzaté z https://www.c-sharpcorner.com/UploadFile/65fc13/types-of-database-management-systems/

[14] StackShare. *What are the best Object Relational Mapper (ORM) Tools?* : StackShare; [cit. 20.4.2020] Prevzaté z https://stackshare.io/orm?fbclid=IwAR1S7Hwqsh-SkDDEPPFsB7bRHrPu4Adn4Ybo7PzM61F-f-DuG_qwAdLtD9g

[15] Letocha, B. *BOBRIS. BTDB* [online] Hradec Králové: Github; [cit. 08.11.2019] Prevzaté z https://github.com/Bobris/BTDB

[16] Letocha, B. *BOBRIS. ODBDictionary* [online] Hradec Králové: Github; [cit. 08.11.2019] Prevzaté z https://github.com/Bobris/BTDB/blob/master/Doc/ODBDictionary.md
[17] Letocha, B. *BOBRIS. Relations* [online] Hradec Králové: Github; [cit. 08.11.2019] Prevzaté z https://github.com/Bobris/BTDB/blob/master/Doc/Relations.md

12 Prílohy

1) Trieda Visitor

```
public class Visitor : IODBVisitor
{
	public BuilderModelObjectv4 Builder =
	new BuilderModelObjectv4(new ModelObject {Name = "base", Type = "base"});

	private static string _tempName = "";
	private int a, b, c, d, e = 0;

	public bool VisitSingleton(uint tableId, string tableName, ulong oid)
	{
		Builder.Down(new ModelObject(tableName,"singleton"));
		return true;
	}

	public bool StartObject(ulong oid, uint tableId, string tableName, uint
version)
	{
		return true;
	}

	public bool StartField(string name)
	{
		_tempName = name;
		return true;
	}

	public bool NeedScalarAsObject()
	{
		return true;
	}

	public void ScalarAsObject(object content)
	{
		if (content != null)
		{
			switch (Builder.Objs[Builder.Objs.Count - 1].Type)
			{
			case null:
				Builder.WriteName(_tempName);
				Builder.WriteType(string.Format(CultureInfo.InvariantC
			ulture, "{0}", content.GetType()));
				break;
			case "InlineObject":
				Builder.Down(new ModelObject(_tempName));
				Builder.WriteType(string.Format(CultureInfo.InvariantC
			ulture, "{0}", content.GetType()));
				break;
			case "singleton":
				Builder.Down(new ModelObject(_tempName));
				Builder.WriteType(string.Format(CultureInfo.InvariantC
			ulture, "{0}", content.GetType()));
				break;
			case "relValue":
```

```
                        Builder.Down(new ModelObject(_tempName));
                        Builder.WriteType(string.Format(CultureInfo.InvariantC
                ulture, "{0}", content.GetType()));
                        break;
                default:
                {
                        Builder.WriteName(_tempName);
                        Builder.WriteType(string.Format(CultureInfo.InvariantC
                ulture, "{0}", content.GetType()));
                        break;
                }
        }
        }else
        {
                Builder.Down(new ModelObject(_tempName));
        }
}

public bool NeedScalarAsText()
{
        return true;
}

public void ScalarAsText(string content)
{
        Builder.WriteLastValue(content);
        Builder.Up();
}

public void OidReference(ulong oid)
{
}

public bool StartInlineObject(uint tableId, string tableName, uint version)
{
        Builder.WriteName(tableName + " " + e++);
        Builder.WriteType("InlineObject");
        return true;
}

public void EndInlineObject()
{
        Builder.Up();
}

public bool StartList()
{
        Builder.Down(new ModelObject(_tempName,"List"));
        return true;
}

public bool StartItem()
{
        _tempName = "ListItem " + a++;
        Builder.Down(new ModelObject(_tempName, "ListItem"));
     return true;
}
```

```
public void EndItem()
{
}

public void EndList()
{
	Builder.Up();
}

public bool StartDictionary()
{
	Builder.Down(new ModelObject(_tempName, "Dictionary"));
	return true;
}
public bool StartDictKey()
{
	_tempName = "DictKey " + b++;
	Builder.Down(new ModelObject(_tempName, "DictKey"));
	return true;
}

public void EndDictKey()
{
}

public bool StartDictValue()
{
	_tempName = "DictValue " + c++;
	Builder.Down(new ModelObject(_tempName, "DictValue"));
	return true;
}

public void EndDictValue()
{
}

public void EndDictionary()
{
	Builder.Up();
}

public void EndField()
{
}

public void EndObject()
{
	a = 0;
	b = 0;
	c = 0;
	e = 0;
}

public bool StartRelation(string relationName)
{
	Builder.Down(new ModelObject(relationName, "relation"));
	return true;
}
```

```
        public bool StartRelationKey()
        {
               Builder.Down(new ModelObject("relKey", "relKey", true));
               return true;
        }

        public void EndRelationKey()
        {
        }

        public bool StartRelationValue()
        {
               Builder.Down(new ModelObject("relValue " + d++, "relValue"));
               return true;
        }

        public void EndRelationValue()
        {
               Builder.Up();
        }

        public void EndRelation()
        {
               d = 0;
        }

        public void InlineBackRef(int iid)
        {
        }

        public void InlineRef(int iid)
        {
        }

        public void MarkCurrentKeyAsUsed(IKeyValueDBTransaction tr)
        {
        }
}
```

2) Odkaz na github repozitár, kde je k dispozícií celá aplikácia: https://github.com/rychvma1/BTDBWebApplication