Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Použití gRPC pro protokol ovladače databázového systému

Diplomová práce

Autor: Bc. Tomáš Pozler
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Hradec Králové Duben 2023

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 27.4.2023 Tomáš Pozler

Poděkování:

Tímto bych chtěl poděkovat vedoucímu diplomové práce prof. RNDr. PhDr. Antonínu Slabému, CSc. za odborné vedení práce, poskytnuté rady a přínosné konzultace. Zároveň bych chtěl poděkovat také rodině a přátelům za jejích pomoc a podporu a Ing. Janu Novotnému za umožnění práce na evitaDB projektu.

Anotace

Cílem této diplomové práce je navrhnout a vyvinout ovladač s využitím gRPC pro evitaDB, databázi zaměřenou oblast e-commerce. První část práce je věnována představení zmíněné databáze, principům jejího fungování a seznámení se s jí používanými koncepty. Následující část je zaměřena na porovnání třech nejznámějších a nejpoužívanějších technologií sloužících pro tvorbu API a jimi používaných serializačních formátů. Na základě zjištěných informací je provedena bližší specifikace pro použití gRPC v rámci evitaDB databáze, poté jsou definovány konkrétní požadavky na implementované ovladače. Poslední část práce je věnována samotnému vývoji, který je rozdělen na dvě části: návrh a realizace gRPC API a poté i samotného ovladače.

Klíčová slova: evitaDB, gRPC, Protocol Buffers, e-commerce, databáze

Annotation

Title: Database system driver protocol based on gRPC

The aim of this thesis is to design and develop a driver using gRPC for evitaDB, a database focused on e-commerce. The first part of the thesis is dedicated to the introduction of the mentioned database, principles of its operation, and to explanation of basic concepts. The next part is focused on the comparison of technologies most often used for the creation of APIs and the serialization formats used by them. Based on the gathered information, a more detailed specification for the use of gRPC within the evitaDB database is made, then the specific requirements for the implemented drivers are defined. The last part of the thesis describes the development, which consists of the design and implementation of the gRPC API and the driver itself.

Key words: evitaDB, gRPC, Protocol Buffers, e-commerce, database

Obsah

1. Úvod

V dnešní digitální a technologicky pokrokové době se vše modernizuje, zlepšuje a zrychluje. Tomuto trendu se nevyhnula ani sféra e-commerce, která se zaměřuje na internetové obchodování – zjednodušeně e-shopy. Postupem let mnohé obchody či společnosti začínají provozovat své služby i v online podobě, některé se na ně i kompletně přesouvají. S tím souvisí čím dál tím větší požadavky a náročnost ze strany klientů, kteří mají kvůli značné konkurenci na trhu méně a méně trpělivosti, když si chtějí prohlížet nabízené zboží v internetovém katalogu a na dostupné produkty aplikovat nesčetné filtrační podmínky. Tyto úkony mohou být na straně serveru výpočetně velmi náročné a získání uživatelem specifikovaných dat může s použitím klasické databáze zabrat velké množství času. Se zvyšujícími se odezvami mají potenciální zákazníci sklon ke ztrátě trpělivosti a radši se přesunou na jiný e-shop, jenž může nabídnout lepší uživatelský zážitek.

Na tento problém poukázala firma FG Forrest, která má již dlouholetou praxi s vývojem složitých webových aplikací. Problém bývá většinou na straně používaných databází, které nejsou schopny, obzvláště při nadměrném počtu prodávaných produktů, efektivně vyhledat požadované záznamy s aplikovanými filtračními podmínkami. Z jejich strany byla formulována hypotéza o tom, že databáze vyrobená přímo na míru danému problému bude vhodnější než obecně používané databáze. Následně se zrodil projekt evitaDB, jehož cílem bylo vytvořit specializovanou databázi zaměřující se čistě na oblast e-commerce s cílem dosáhnout co nejlepšího výkonu s maximálním využitím hardwarových prostředků v kombinaci s co nejširším spektrem nabízených funkcí.

Na základě zmíněné hypotézy začala ve spolupráci s Univerzitou Hradec Králové výzkumná část projektu, jež byla věnována návrhu a implementaci třech různých verzí evitaDB. Hlavní vyvíjenou verzí byla původní zamýšlená in-memory databáze, zbylé dvě verze byly realizovány na tradičně používaných SQL a NoSQL databázích. Pro tyto účely byly vytvořeny tři týmy, přičemž každý byl zaměřen na jeden typ databází. V průběhu následujících dvou let na nich byly realizovány všechny z požadovaných funkcí, na jejichž základě proběhlo ověření stanovené hypotézy. Výzkumná část projektu byla spolufinancována Evropskou unií a Ministerstvem průmyslu a obchodu.

V aktuálním chvíli je databáze evitaDB stále v alfa verzi a je nadále velmi aktivně vyvíjena a neustále jsou přidávány nové funkce a optimalizovány ty stávající. Jedním z cílových požadavků na databázi je umožnit její běh v izolovaném prostředí (například v Docker

kontejneru nebo v samostatné aplikaci), v němž by ji bez použití některého z již implementovaných API (REST a GraphQL) nešlo používat. Databáze je psaná v programovacím jazyku Java a jejího maximálního výkonnostního potenciálu dosahuje při jejím přímém používání jako knihovny. Tím přichází na řadu gRPC, třetí z požadované trojice API provozovaných nad evitaDB databází, jenž bude v teoretické části této práce představeno a porovnáno s ostatními zmíněnými API. Na základě zjištěných skutečností bude rozhodnuto o jeho specifickém účelu v rámci této databáze, a právě na implementaci gRPC API se zaměří praktická část této diplomové práce.

2. Cíl práce

Před samotnou realizací gRPC API bude vhodné se s evitaDB databází detailněji seznámit a obecně se lépe zorientovat v související problematice. To bude zahrnovat představení používaných konceptů a principů, vysvětlení základních pojmů a datových struktur a také nahlédnutí na její reálné používání jak z uživatelského, tak i z programátorského hlediska.

Pro dosažení co nejlepšího návrhu gRPC API budou blíže uvedena a rozebrána i ostatní evitaDB API – REST a GraphQL, jelikož jedním z požadavků na všechna API je jejich vzájemná podobnost pro usnadnění jejich údržby a přechodu z jednoho na druhé. U každého z nich budou popsány jejich obecné základní charakteristiky, používané komunikační prostředky a způsoby jejich vystavení i konzumování. Následovat bude výčet s nimi spojených výhod a nevýhod a vhodných případů jejich použití, načež proběhne i shrnutí a zhodnocení již implementovaných evitaDB API vzhledem k jejich výkonu a stylu používání.

Na základě zjištěných informací bude definována konkrétní podoba a jí odpovídající požadavky na navrhované gRPC API, které bude v rámci praktické části navrženo, vytvořeno a jeho použití otestováno z klientské perspektivy.

3. EvitaDB

Projekt evitaDB započal v roce 2020 s cílem vytvořit specializovanou databázi pro e-commerce. Výsledkem mělo být inovativní, rychlejší a účelové řešení – ve srovnání s již existujícími obecně použitelnými databázemi. Na počátku byla vydefinována hypotéza, že specializovaná databáze vytvořená na míru požadavkům e-commerce může dosahovat řádově rychlejší odezvy a propustnosti dat než obecné SQL či NoSQL databáze. [1] Pro její potvrzení byla realizována výzkumná část projektu, v níž byla vydefinovaná sada e-commerce funkcí poskytovaných evitaDB databází realizována na PostgreSQL a ElasticSearch databázích, které byly pro tento účel vyhodnoceny dle specifikovaných parametrů jako nejvhodnější [2]. V závěru této fáze proběhlo výkonnostní testování, které hypotézu potvrdilo – evitaDB byla vyhodnocena z testovaných databází jako nejrychlejší, konkrétně 100× rychlejší než PostgreSQL a 10× rychlejší než ElasticSearch. Výsledky byly velmi působivé, evitaDB dosáhla lineárního škálování při změnách hardwaru (CPU, RAM) a zároveň u ní bylo dosaženo nejefektivnějšího využití přidělených systémových zdrojů. V tabulce níže jsou uvedeny průměry naměřených hodnot představující provedené operace za sekundu z dílčích testů provedených v rámci výkonnostního testování na největší z testovaných datových sad, konkrétně sady z e-shopu senesi.cz. [3]

Tabulka 1 - Propustnost (požadavky/sek) databází na datové sadě z e-shopu senesi.cz. Zdroj: [3]

	evitaDB	PostgreSQL	ElasticSearch
Průměr	9694.48	67.59	1251.11
Sm. odchylka	1537.08	0.61	8.40

Tuto in-memory[1] databázi jako takovou lze zařadit do NoSQL kategorie. Celá její implementace je v programovacím jazyku Java, což u NoSQL databází, narozdíl od SQL, není nijak ojedinělé. Jak již bylo zmíněno, narozdíl od běžně používaných obecných databází je evitaDB zaměřená čistě pro účely e-commerce webových katalogů, a tudíž nemá sloužit jako hlavní úložiště dat. Má vystupovat v roli sekundárního úložiště sloužícího pro reálnou webovou aplikaci zaměřující se primárně na rychlost vyhledávání a rozmanité funkce. V provozu by měla tedy obsahovat reálná data z e-shopu, která je nutné naimportovat pomocí některého

[1] In-Memory – tímto pojmem jsou označovány databáze cílící na co nejmenší odezvy dotazů, přičemž většina jejich dat je uložena v paměti; s diskem pracují pouze minimálně (např. při inicializaci dat) [4]

z dostupných API. Obsah evitaDB musí vždy reflektovat aktuální data v primárním úložišti, zároveň z principu a účelu používaní této databáze jako vyhledávacího indexu by mělo být možné uložená data kdykoliv smazat a znovu si je naimportovat. Databáze má také vestavěnou chytrou mezipaměť pro maximalizaci výkonů a správy systémových prostředků a v provozním režimu katalogu (ALIVE stav) disponuje i vlastnostmi ACID[2], které zajišťují správné nakládaní s daty při provádění transakcí. [6]

Databázi lze provozovat v JVM v embedovaném režimu, kde má programátor přímý přístup k funkcím z modulu *evita_api*, pomocí něhož ji může nastartovat, ukládat do ní data a dotazovat je. Jelikož není třeba řešit procesy jako serializace nebo přenos dat po síti, je při tomto použití logicky dosaženo nejlepších výsledků z hlediska rychlosti. Alternativním přístupem k manipulaci s databází může být použití některého z dostupných API – ta by evitaDB ve finálním stavu měla mít hned ve třech provedeních: REST, GraphQL a gRPC (viz. 4. kapitola). První dvě zmíněná API šla do vývoje jako první, přičemž u nich byla snaha o udržování vzájemně podobných konceptů a v jistých směrech i o jejich přiblížení jak z implementačního, tak i z uživatelského hlediska, aby nebylo nijak zvlášť složité přecházet z jednoho na druhé. [6]

Cíl byl pro všechna API nastaven stejně: s použitím dané technologie vytvořit řešení, pomocí něhož by bylo možné ovládat databázi běžně užívanými technikami specifickými pro danou technologii, pokusit se neodchylovat od ostatních API a pokud možno co nejvíce sjednotit společnou logiku. Zároveň mezi cíle spadala i adaptabilita na provozovanou doménu, tedy aby se API dokázalo přizpůsobit a využít databázi, nad níž operuje, jako například omezením možných dotazů pouze na věci, které tam reálně jsou a dávají smysl. Doménou je zde myšlena možnost specifikovat si vlastní používanou strukturu a terminologii. U REST API se jednalo o vygenerování OpenAPI specifikace, u GraphQL pak o vygenerování schémat, nad nimiž je uživateli umožněno provádět dotazy, což například zamezilo zaslání nevalidního datového typu nebo nesmyslnému zkonstruování struktury dotazu. Společným rysem pro obě API bylo i přizpůsobit vystavené endpointy pro komunikaci s API tak, aby byly zaregistrovány specifické URL, například pro dynamicky získané katalogy z databáze. Tímto je zajištěno, že konkrétní dotaz bude prováděn nad odpovídajícím katalogem a schématem databáze. Poslední ze zmíněných API je gRPC, které by v optimálním případě mělo nabízet co nejpodobnější strukturu a funkce jako ostatní API a zároveň přinést konzumujícímu programátorovi co

[2] ACID – zkratka označující čtveřici vlastností, které u databází zajišťují spolehlivost při provádění transakcí, v překladu se jedná o vlastnosti: atomicita (nedělitelnost), konzistence, izolace a trvanlivost. [5]

nejlepší a nejpříjemnější zážitek. Charakteristika a analýza každého ze zmíněných API, vyzdvižení silných i slabých stránek a identifikování problémů, a to včetně případů, na které se API jako takové hodí jak z obecného hlediska, tak i při konkrétním používaní v kombinaci s evitaDB, budou rozvedeny ve 4. kapitole. Následně bude v realizační části vysvětlen návrh a popsán postup vývoje serverové části a demonstrováno používání tohoto API z technologie odlišné od té serverové. [7, 8]

Ve zbytku první kapitoly budou představeny stěžejní pojmy používané v evitaDB pro získání lepšího obrazu o jejím fungování a přehledu nezbytných funkcí, které musí být obsaženy ve výsledném řešení. Pro lepší pochopení a hlubší nastínění problematiky budou používány ukázky a vysvětlivky přímo z evitaDB implementace, a to buď z oficiální dokumentace nebo ze zdrojových kódů.

3.1. Katalog

Pojmem *Catalog* se v evitaDB označuje databáze pro jeden konkrétní e-shop. Samotná běžící instance evitaDB může obsahovat, a tedy i obsluhovat, mnoho katalogů, které jsou od sebe vzájemně izolovány. Při provádění operací na katalogu A nelze proto jakkoli odkazovat nebo používat katalog B. Díky tomu mohou různé katalogy obsahovat stejné pojmenované struktury bez rizika kolizí a lze tak provozovat více e-shopů na jedné instanci evitaDB. [9]

3.2. Kolekce entit

Pro oddělení dílčích typů uložených entit je používán pojem *Collection* nebo i *EntityCollection*. Jak již bylo zmíněno v předchozí podkapitole, tyto kolekce jsou součástí katalogů a může jich být v jednom katalogu libovolné množství. Tento název je také běžně používán k obdobnému účelu v jiných NoSQL databázích, jako například MongoDB. Nejblíže (z existujících používaných pojmů z relačních databází) by je šlo přirovnat k databázovým tabulkám, ještě specifičtěji by kolekce odpovídala projekci definované jako "*množina logicky souvisejících tabulek*". [9]

Kolekce, narozdíl od katalogů, od sebe nejsou nijak izolované; do nich náležící entity se mohou odkazovat i na entity z jiné kolekce. Typickým příkladem by mohla být entita typu *produkt* referující na (spadající pod) entitu typu *kategorie* nebo nějaké *značky*. V evitaDB je každá entita v kolekci jednoznačně identifikována unikátním číselným 32bitovým identifikátorem typu integer. Na úrovni katalogu je pro jednoznačnou lokalizaci potřeba použít

dvojici informací: názvu kolekce a číselného identifikátoru entity nebo jiného unikátního atributu entity. [9]

3.3. Podporované datové typy

EvitaDB podporuje celou řadu datových typů, přičemž většina z nich je přímo odvozená z datových typů Java platformy, v níž je databáze naprogramována. Podporovány jsou i různé implementace třídy *Range* umožňující práci s časovými nebo číselnými intervaly. Níže se nachází seznam základních datových typů, které mohou být použity pro atributy entit a podle nichž lze provádět vyhledávací a třídící operace. Ke každému typu je uveden příklad hodnoty, která jej reprezentuje [10]:

- String – “hodnota”,
- Byte – 120,
- Short – 20000,
- Integer – 65536,
- Long – 3000000000,
- Boolean – true,
- Character – “a”,
- BigDecimal – 3.1415,
- OffsetDateTime – 2023-01-01T00:00:00+01:00,
- LocalDateTime – 2023-01-01T00:00:00,
- LocalDate – 2023-01-01,
- LocalTime – 11:34:00,
- DateTimeRange – [2023-01-01T00:00:00+01:00,2024-01-01T00:00:00+01:00],
- BigDecimalNumberRange – [-1.123,9.36],
- LongNumberRange – [3,5],
- IntegerNumberRange – [0,10],
- ShortNubmerRange – [0,],
- ByteNumberRange – [0,1],
- Locale – `cs-CZ`,
- Currency - `CZK`.

Většina z datových typů je běžně používaná i v jiných databázových řešeních, nicméně některé z nich si zaslouží podrobnější komentář.

Jedna ze základních otázek by mohla směřovat k důvodu, proč je z vestavěných datových typů pro uchovávání desetinných čísel použitý právě typ BigDecimal. Na rozdíl od existujících alternativ ve formě typů float a double, které nejsou zdaleka tak přesné, slibuje BigDecimal absolutní přesnost. A jelikož jeden z hlavních účelů tohoto datového typu na úrovni databáze je ukládat ceny, u nichž je maximální přesnost nutností, přispěl poslední ze zmíněných faktorů k volbě právě tohoto datového typu. Tato přesnost může být nezbytná při práci s velmi hodnotnými, nebo naopak extrémně nízko hodnotnými měnami. Jako příklady lze uvést Bitcoin, u něhož nejvyšší jednotka přesnosti, tzv. " Satoshi", odpovídá osmi desetinným místům a téměř 2 000 Íránských Riálů odpovídá jedné české koruně. [11, 12]

Dalšími třídami, které by bylo vhodné trochu více představit, jsou Locale a Currency patřící mezi základní typy Java platformy. Locale, jak již název napovídá, reprezentuje konkrétní jazyk a místo, zemi či region identifikovaný nejčastěji pomocí zkratky jazyka (*cs*), nebo jeho kombinací se zkratkou země (*cs-CZ*), tato kombinace je v této podobě označována jako *Language tag* – jako používaný standard byl zvolen *IETF language tag* [13]. Seznam podporovaných hodnot lze dohledat online, například v Java dokumentaci [14]. Druhá ze zmíněných tříd, Currency, udržuje informace o měnách, přičemž jako parametr jsou akceptovány jejich zkratky ve formátu dle normy ISO 4217 – pro českou korunu je používána zkratka *CZK*. První dvě písmena představují kód země a třetí písmeno odpovídá prvnímu písmenu názvu používané měny. [15]

Datový typ Range byl navržen speciálně pro evitaDB. Je sám o sobě pouhým generickým rozhraním (interfacem) zaštiťujícím dvě jeho implementace v podobě třídy DateTimeRange a abstraktní třídy NumberRange. Poskytuje pouze pomocné metody a obsahuje definici metod pro navrácení hodnot *from* a *to* generického typu. Tyto hodnoty udávají začátek a konec intervalu, přičemž Range umožňuje specifikaci obou (inkluzivní oboustranný interval), nebo pouze počáteční či pouze koncovou hodnotu (zleva nebo zprava otevřený inkluzivní interval). Hodnoty uživatel v tomto smyslu nepředává napřímo přes konstruktor, ale využívá pro tyto účely statických metod nacházejících se na implementacích tohoto rozhraní: v případě DateTimeRange jde o metody *since* a *until*, u tříd dědících od třídy NumberRange se jedná o metody *from* a *to*. Všechny implementace zmíněného rozhraní disponují metodou *between*, jež slouží pro předání obou hranic intervalu. [10, 16]

Mohlo by se zdát, že evitaDB podporuje až zbytečně velké množství celočíselných datových typů, a to hned čtyři (byte, smallint, int a long). Vzhledem k tomu, že evitaDB je paměťová databáze, je v případech, kdy je zřejmé, že jejich hodnoty budou zaručeně menší

(například u typu *byte* podporuje hodnoty od -128 do 127), vhodné použít odpovídající datový typ pro minimalizaci paměťového vytížení a sekundárně i pro urychlení prováděných operací. Vhodné je také zmínit i skutečnost, že všechny výše zmíněné datové typy jsou podporovány i jako pole, což umožňuje mít v rámci jedné hodnoty (atributu) více konkrétních hodnot. I v případě pole hodnot je podporováno vyhledávání formou alespoň jedné shody.

Posledním datovým typem, který je potřebné zmínit, je ComplexDataObject. Tento typ není ve výše uvedeném výčtu datových typů, jelikož se od nich odlišuje tím, že jej není možné použít pro atributy (a tudíž k vyhledávání a třídění), ale pouze jako hodnotu Asociovaných dat (viz. 3.4.2), a také opět i ve formě pole. Do tohoto formátu lze převést objekty s bohatou strukturou, které je možné charakterizovat jako POJO (Plain Old Java Object) nebo nově tzv. record typy[3]. Zjednodušeně by se dalo říci, že ComplexDataObject odpovídá libovolné datové struktuře s hierarchickou organizací. S tímto typem uživatelé nepracují napřímo, používají pouze převaděč (konverter), který umožňuje transformovat instance jejich vlastních objektů do tohoto typu a z něj zpátky do původní podoby. Takový objekt může obsahovat, kromě uvedených typů výše v seznamu, jiné POJO třídy, generické kolekce nebo pole jakéhokoliv ze zmíněných datových typů. Tato struktura byla zvolena s přihlédnutím k možnosti pomocí reflexe konvertovat objekty z, resp. do jejich původních datových typů a zároveň kvůli umožnění snadné serializace do JSON formátu. [10]

3.4. Entita

Entita představuje reálný záznam z e-shopového systému uloženého v databázi. Kromě jejích unikátních identifikátorů (typ a primární klíč) odpovídá schématu předepsanému kolekcí, do níž typově spadá. Dále pak disponuje kolekcí typu Locale představující jazyky, pro které uchovává alespoň jednu lokalizovanou hodnotu atributu či asociovaného data, viz. 3.4.1 a 3.4.2. Následuje informace o nadřízeném záznamu entity v případě, že je její typ ve schématu označen jako hierarchický. Pomocí této informace lze entity filtrovat nebo si zjistit dodatečné detaily spojené s hierarchičností, jako například rekurzivní získání všech entit nacházejících se ve stromu nad ní (své rodiče) nebo získání statistik obsahujících data o počtech potomků daných entit ve stromové struktuře. [9]

[3] Record typy – speciální immutabilní typ třídy určený pouze k uchovávání a přenášení strukturovaných dat, cílí na snížení nutnosti opakovaného psaní stejných metod potřebných pro docílení chování POJO objektů, automaticky generuje konstruktor pro specifikované parametry a implicitně disponuje potřebnou implementací metod k porovnávání a testování unikátnosti objektů, jako hashCode a equals. [17]

Entita v evitaDB implementující rozhraní *EntityContract* disponuje i sadou s ní souvisejících dat, která jsou rozdělena do čtyř samostatných skupin: atributy, asociovaná data, reference a ceny. Tyto součásti budou více vysvětleny v podkapitolách níže. Je také podstatné podotknout, že entitu lze reprezentovat i rozhraním *EntityReferenceContract*, které pouze odkazuje na existující entitu pomocí typu a identifikátoru. Tato reprezentace entity neobsahuje žádná dodatečná data, slouží pouze jako informace o existenci entity v databázi – vzhledem k nulovému počtu navrácených dodatečných dat společně s takovou entitou jsou vykonané dotazy výrazně rychlejší (stačí jim pouze práce s indexy v paměti), ale mají menší informační hodnotu.

3.4.1. Atributy

Atributy představují nejrůznější vlastnosti entity, může se jednat například o jméno, barvu, expiraci, množství atd. Strukturu atributu je možné rozdělit na dvě části: identifikátor (klíč) a hodnotu. Klíč se může skládat buď pouze z názvu, nebo jeho kombinace s některou lokalizací. Podle klíče lze takto rozdělit atributy na lokalizované a globální. Dotazovací jazyk pak následně umožňuje nechat si vrátit entitu pouze s globálními, a například česky lokalizovanými, atributy. Ve webových katalozích je velké množství dat lokalizovaných pro použití v různých zemích a pohled na entitu v konkrétní lokalizaci zjednodušuje implementaci koncovému vývojáři. V rámci dotazu na evitaDB databázi lze specifikovat jazykové varianty, ve kterých mají být výsledky vráceny, což zjednodušuje práci vývojáři. Jako hodnotu mohou atributy mít cokoliv z podporovaných datových typů, uvedených v kapitole výše, s výjimkou typu ComplexDataObject. [9]

Dalším zajímavým faktem, jenž je možné zmínit u atributů, jsou jejich vlastnosti, které lze nastavit v jejich schématu. Konkrétně se jedná například o příznaky filtrovatelnosti (jestli je možné podle daného atributu filtrovat), třiditelnosti (zda je možné podle něj řadit výsledky) nebo unikátnosti (jestli jednoznačně identifikuje danou entitu). Na jejich základě jsou v paměti databáze udržovány indexy umožňující rychlé odezvy databáze. Vedlejším produktem je ovšem větší paměťová náročnost, a proto není vhodné vkládat do atributů velké množství dat. Také je ve schématu uvedeno, jestli je hodnota atributu nepovinná, může být nastavena i výchozí hodnota a datový typ, jenž bude databází pro atribut s tímto jménem vyžadován. [9]

3.4.2. Asociovaná data

Asociovaná data jsou velmi podobná atributům – také mají klíč skládající se z názvu a potenciálně i lokalizace a rovněž mají i hodnotu. Hlavním rozdílem je, že se pro ně netvoří žádné indexy v paměti, a proto mohou obsahovat větší množství dat. Rozdíl nastává také u podporovaných datových typů, jelikož do asociovaných dat lze ukládat i vlastní složitější POJO struktury, které nejsou databází implicitně podporovány (více viz. 3.3). Asociovaná data tak slouží čistě k uložení informací, jež mají být pouze zobrazeny bez možnosti podle nich filtrovat či řadit entity. Typickým příkladem asociovaných dat v praxi mohou být kupříkladu bohaté popisy produktů v HTML formátu nebo odkazy na seznam mediálních souborů ve fotogalerii. Čistě z principu a jejich účelu u nich nedává smysl provádět filtrační operace. [9]

3.4.3. Reference

Jak již název nasvědčuje, primárním účelem referencí je odkazovat na jinou entitu, která může být i odlišného typu, než byla entita, které ona reference náleží. Odkazovaná entita se nemusí ani nacházet v evitaDB, ale může se jednat o odkaz na externí entitu v jiném systému či databázi. Kvůli tomu je reference v evitaDB identifikována podobně jako entita, tedy primárním číselným identifikátorem a typem odkazované entity v kombinaci s názvem reference, který pak slouží hlavně k rozlišení více referencí na stejné kolekce entit. Reference může mít i vlastní atributy, což se dá ve světě relačních databází připodobnit k vazební tabulce, v níž jsou další datové sloupce. Reference mají jako ostatní zmíněné vnitřní kolekce schéma, jenž definuje, na jakou kolekci dané reference odkazují a jaké atributy jsou na této vazbě povolené. Pomocí referencí lze vyřešit celá škála funkcionalit vyskytujících se na většině e-shopů. [9]

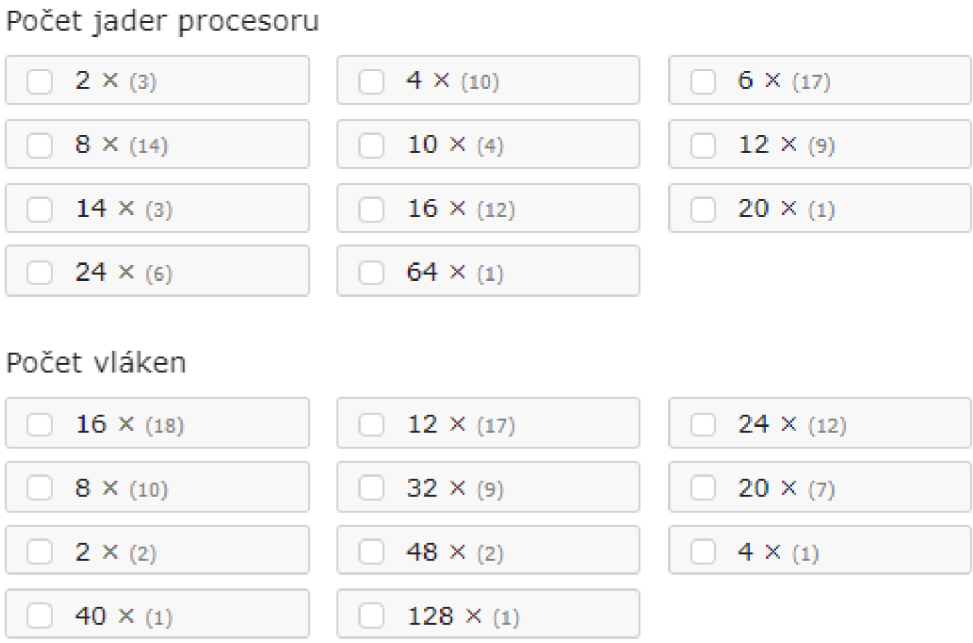
Jedním z těchto obvyklých případů referencí jsou tzv. kategorie, které umožňují organizovat prodávaný produkt do stromové struktury, jež pomůže zákazníkovi se lépe orientovat v nabídce obchodu. E-shop obvykle definuje seznam základních – kořenových kategorií. Každá z kategorií může mít maximálně jednu nadřízenou kategorii (předka) a žádnou nebo více podřízených kategorií (potomků). Pro ilustraci je na obrázku níže část této hierarchické struktury kategorií z e-shopu alza.cz, na níž se nachází kořenová kategorie "Počítače a notebooky", ta má jako jednoho z potomků (pro přehlednost je uveden pouze jeden) kategorii "Komponenty" a tak dále. Podstatné ovšem je, že do kategorie může patřit libovolný (i žádný) počet jak produktů, tak i dalších kategorií. Produkt sám o sobě může spadat do více kategorií, ale teoreticky by nemusel být v žádné.

Obrázek 1 - Stromová struktura výseče kategorií z e-shopu alza.cz. Zdroj: [autor]

Tohoto efektu je v evitaDB možné dosáhnout tak, že entita představující produkt patřící například do kolekce *Produkt* bude obsahovat reference na hierarchické entity označující kategorie – například *Kategorie*, v nichž se má zobrazovat (patří přímo do nich). Použití referencí je velmi flexibilní a umožňuje docílit třeba i podobné logiky u entit stejného typu. To by se na příkladu dalo demonstrovat třeba v případě, že by e-shop prodával produkty v podobě kompletů (produkt skládající se z variabilního počtu dílčích produktů) – příkladem může být koupě stolu reprezentovaného samostatně koupitelnými částmi – deska, nohy.

Mezi další funkce nacházející se na mnoha e-shopech patří uživatelem parametrizované filtrování zobrazených produktů. Příklad takové funkcionality je možné vidět na obrázku níže, kde je zobrazena část filtrace procesorů z e-shopu alza.cz. Z obrázku je možné vidět dvě samostatné skupiny aplikovatelných parametrů, se kterými je uživateli umožněno manipulovat. Vedle samotných parametrů se nachází i hodnota představující počet zobrazených entit, které ve výpisu zbydou po vybrání daného parametru. Zajímavým faktem je chování tohoto filtračního okna při výběru položky z kategorie “Počet jader”. Při výběru libovolného zaškrtávacího pole z této skupiny jsou omezeny možnosti z druhé skupiny “Počet vláken”, což je způsobeno tím, že mezi skupinami je aplikována booleovská operace AND (a zároveň). Většina možností je odstraněna proto, že neexistuje žádný produkt obsahující zároveň oba zvolené parametry. V rámci stejné skupiny se mezi parametry aplikuje boolovská operace OR (nebo), jež naopak rozšiřuje aplikovatelné filtry z jiných skupin.

V evitaDB mají tyto filtry vlastní pojmy, jednotlivé položky se nazývají *facety*, které mohou spadat do některé z existujících skupin nazvaných *facet group*. Facety i jejich skupiny jsou v rámci implementace realizovány jako obohacené reference na jiné entity, samy o sobě jsou však entitami odlišného typu s vlastními daty. V rámci dotazu na databázi je možné facety vybrané uživatelem specifikovat v samostatném kontejneru nazvaném userFilter, jenž slouží pro odlišení filtrů specifikovaných uživatelem a aplikací. Lze v něm specifikovat libovolný počet filtračních podmínek, které mohou být v rámci výpočtů na databáze upraveny nebo potlačeny. Příkladem může být omezení cenového rozsahu či hodnota některého z intervalových atributů (např. šířka od/do). [18]



Obrázek 2 - Ukázka filtrace produktů podle parametrů na e-shopu alza.cz. Zdroj: [autor]

Vztahy mezi parametry (facety) či skupinami parametrů je možné snadno měnit (zaměnit AND/OR operace), či je možné jejich význam negovat (booleovská operace NOT). Do budoucna je plánována i podpora exklusivity výběru facetů ve skupině, což odpovídá jejich zobrazením pomocí komponenty výběru (radio select) místo zde uvedené nabídky zaškrtávacích polí (checkbox select). Více o těchto možnostech je rozepsáno v kapitole 3.7. Cílem kvalitně provedeného parametrického vyhledávání je nabídnout uživateli informaci o omezení/zúžení výběru dle jeho preferencí ještě před tím, než provede vyhledání produktů, a tudíž předejít vyhledáváním s prázdnou nabídkou, kdy vybranému filtru neodpovídá žádný z produktů.

3.4.4. Ceny

Ceny jsou nedílnou součástí jakéhokoliv e-commerce řešení. V základních rysech se odlišují od dříve zmíněných přidružených dat například tím, že jim není předepsáno žádné schéma, kterým se musí řídit. Jelikož evitaDB cílí i na B2B e-shopy[4], tak bylo nutné vyhovět i dosti specifickým potřebám. V této sféře je běžné, že jeden produkt může mít i několik desítek cen v různých cenících pro různé zákazníky a partnery, přičemž ne všechny musí být reálně určené k prodeji. Příkladem takových cen může být obvyklá cena produktu na trhu nebo původní cena před zlevněním pro získání většího zájmu klientů; tyto ceny nejsou nicméně určeny ke konečnému prodeji. Z tohoto důvodu lze u každé ceny specifikovat, zda se jedná o cenu prodejní. Neprodejní ceny je možné v určitých vnitřních procedurách, například pro výpočet seznamu produktů v určitě cenové hladině, ignorovat. S cenami dále souvisí i možnost nastavení odlišných úrovní DPH pro každou cenu dle potřeby, časovou platnost ceny specifikovanou intervalem nebo umožnění odlišného výpočtu prodejní ceny pro složitější případy, jakými jsou např. komplety produktů nebo tzv. master produkty. Jako komplet je myšlen produkt skládající se z částí v podobě dílčích produktů, jako master produkt je označován obecný (většinou neprodejný) produkt, jenž může být v e-shopu k dispozici v několika variantách (například v odlišných barvách). Cena může obsahovat identifikátor přidružené entity, na niž se tímto způsobem odkazuje, a na úrovni entity lze nastavit, jak tato konkrétní entita řeší výpočet cen, které na ni mohou odkazovat. Vytvořeny byly tři základní způsoby výpočtu cen: [9, 19]

- Žádný (None): ID referující na cenu entity nejsou brány v potaz, výchozí hodnota;
- První výskyt (First occurence): navrácena je první cena po sestupném seřazení všech cen se stejným ID referujícím na jinou entitu podle jejich priority;
- Suma (Sum): bude navrácena suma všech cen se stejným ID, jenž referuje na jinou cenu.

Způsob *First occurence* je využitelný například v případě zobrazení jedné varianty produktu tak, aby se u ní ukázala nejnižší cena ze všech jí referovaných variant. Metodou *Sum* lze docílit kupříkladu zobrazení produktu skládajícího se z více menších samostatných kusů, přičemž zobrazená cena odpovídá součtu cen všech dílčích částí.

Cena je unikátně identifikována trojicí parametrů, konkrétně číselným identifikátorem, ceníkem a měnou, v níž jsou uvedeny částky. V rámci dotazu, jenž obsahuje jakýkoliv filtr na cenu, je nutné předat ceník a měnu, pro kterou má být cena platná, jinak dotaz nebude

[4] B2B – e-commerce řešení (e-shopy) zaměřující se na obchodní vztahy mezi firmami, alternativou je B2C, jenž cílí na koncové zákazníky

úspěšně proveden a skončí chybou. Pokud je platnost některých cen omezena pouze na určitě časové období, je nutné v dotazu použít i konkrétní časový okamžik, pro který má být výpočet cen proveden. Jak tento celkový popis napovídá, řešení cen na úrovni evitaDB je velmi složitý, nicméně i tak si je databáze schopna poradit s výpočtem prodejních (a získáním ostatních možných) cen velice rychle i při jejich velmi velkém počtu (milióny cen). [9]

3.5. Schéma

V předchozích podkapitolách byl již několikrát zmíněn termín "schéma". Vhodně a pečlivě zpracované schéma může být nejen užitečné a vhodné pro zajištění konzistence a popisu struktury dat v databázi, ale mohou z něj těžit i další API poskytovaná databází. Jedním z možných využití je automatické vygenerování svých "protokolově" specifických schémat a dokumentací. Tím je možné ušetřit velké množství času vývojářům, kteří by stejně tato schémata pro svá API museli napsat. Z odvozených API schémat lze často generovat i části programového kódu pro různé programovací prostředí, které koncový vývojář již jen použije pro specifické potřeby svého projektu. Strukturálně lze schémata rozdělit do dvou hlavních skupin: do první lze zařadit CatalogSchema a do druhé EntitySchema. Instance typu CatalogSchema je používána katalogem, jemuž schéma předepisuje možné globální atributy opět v podobě dílčích schémat, a jenž zároveň uchovává i všechna schémata pro existující kolekce entit, které v katalogu mohou existovat. Jak z názvu vyplývá, EntitySchema bude podobným způsobem předepisovat pravidla kolekcím entit. Z hlediska možných nastavení je ale mnohem bohatší a flexibilnější, je možné zde totiž nastavit tzv. evoluční módy schématu, které mohou schématu umožnit celkovou nebo částečnou adaptaci na data ze vkládaných entit. V základní konfiguraci, tedy při tvorbě prázdného schématu, jsou všechny typy evoluce povoleny a je tedy velmi snadné a relativně bezpracné rovnou začít vkládat data do databáze a nechat si schéma vytvořit samotnou databází podle struktury vkládaných dat. Tento přístup má ale i své nevýhody, přičemž hlavní již byla zmíněna v souvislosti s generováním API používajících terminologii zákazníka. [20]

Na schématu entit lze nastavovat vlastnosti kolekce, jako například automatické generování primárních klíčů pro nově vložené entity, označení kolekce jako hierarchické nebo povolení kolekci mít přiřazené ceny. V rámci kolekce entit obsahuje také seznamy povolených Locale a Currency. [20]

Schéma entit obsahuje i schémata podporovaných atributů, asociovaných dat a referencí; je vhodné zopakovat, že ceny schéma nemají – pokud jsou ceny v dané kolekci

povoleny, jsou omezeny pouze výčtem měn. Tato jednotlivá schémata již byla popsána výše v podkapitolách 3.4.1, 3.4.2 a 3.4.3.

Schéma lze dle potřeb v čase upravovat. Změn je možné docílit pomocí tzv. mutací, jež představují atomické změny nad schématem. Vzhledem ke komplikovanosti ruční tvorby mutací je práce s nimi obalena do tzv. EntitySchemaBuilderů, které budou představeny v následující podkapitole. [20]

3.6. Mutace

Mutace v evitaDB slouží k provádění jakýchkoliv operací, které vedou k modifikaci uložených dat na disku, mezi něž patří vytváření, editace i mazání libovolných objektů. Výsledné řešení umožňuje ze strany klienta databáze na server poslat pouze seznam konkrétních změn a vyhnout se přenášení nezměněného obsahu. Tyto změny jsou zapsány do Write-Ahead-Logu[5] (WAL) a databáze vyhodnotí jejich potenciální konflikty s dalšími změnami, které byly paralelně odeslány ostatními klienty databáze. Díky tomu, že klient posílá pouze změny, šetří se jak objem dat přenášených po síti, tak i analýza na straně serveru, jenž nemusí provádět žádnou dodatečnou analýzu, která z dat poslaných klientem jsou skutečně změněná, a která nikoliv. [22]

Používání WAL je mezi databázovými stroji běžné, jelikož zprostředkovává hned dvě ze čtyř ACID vlastností: Atomicitu a Durabilitu (trvanlivost). Dále je také využívána tzv. Snapshot izolace, v níž při vytvoření transakce vznikne Snapshot (snímek) stavu databáze – transakce nevidí změny, jež nastaly po jejím vytvoření. Po jejím provedení (operace COMMIT) budou přijaty pouze ty změny, které nejsou kontradikovány jinými paralelně provedenými transakcemi, k nimž došlo od vytvoření snímku databáze. [22, 23]

Aby se předešlo nadměrnému manuálnímu vytváření mutací, což může být poněkud zdlouhavé, vytváří některé operace (například mazání) mutace na základě požadavků implicitně na serverové straně a není to tedy nutné řešit manuálně, postačí pouhé použití dostupných metod, jež dává k dispozici samotná databáze. Pro usnadnění manipulace s mutacemi slouží také tzv. Buildery, s jejichž pomocí lze vývojáře odstínit od vlastní mutační logiky. Pomocí připravených metod lze snadno nastavit specifické požadované změny, přičemž po dokončení jejich nastavení na instanci builderu lze zavolat metodu *toMutation*. Ta pro použitý typ builderu

[5] WAL – běžně používaný koncept při provádění transakcí na databázi, funguje na bázi ukládání prováděných změn mimo hlavní (trvalé) úložiště, odkud jsou poté zpracovány a propsány do databáze. [21]

sestaví jediný objekt obsahující všechny mutace, které byly vytvořeny na základě aplikovaných změn ve srovnání s jeho počátečním stavem. Jejich typů je k dispozici celá řada, přičemž jsou od sebe odlišeny výsledným sestaveným objektem, tím, jestli má být sestaven nový objekt (při vkládání nové věci do databáze), nebo jestli má být výsledkem pozměněná verze již existujícího objektu. [20, 22]

Ve stromu níže je uvedena zjednodušená podoba stromu základních typů mutací společně s uvedením zástupce z dané kategorie, které lze v evitaDB provádět. Názvy odpovídají rozhraním v kódu s hlavním účelem zformovat oddělené skupiny mutací, aby bylo možné zajistit, že by například do metody měnící schéma entity přišla mutace na vkládání entity. Tato rozhraní jsou realizována velkým množstvím tříd vždy reprezentující právě jednu změnu uchovávané struktury, na níž má být aplikována.

- Schéma [20]
    - TopLevelCatalogSchemaMutation: např. CreateCatalogSchemaMutation – vytvoří nový katalog;
    - LocalCatalogSchemaMutation: CreateEntitySchemaMutation – slouží ke vzniku nové kolekce entit;
    - GlobalAttributeSchemaMutation: CreateGlobalAttributeSchemaMutation – zaregistruje nové schéma globálních atributů;
    - EntitySchemaMutation: ModifyEntitySchemaMutation – upraví již existující schéma entit daného typu;
    - AttributeSchemaMutation: RemoveAttributeSchemaMutation – odstraní ze schématu entitního typu schéma zadaného atributu;
    - ReferenceAttributeSchemaMutation: ModifyAttributeSchemaTypeMutation – na atributu nacházejícím se na reference provede požadovanou změnu akceptovaného typu uložitelných hodnot;
    - AssociatedDataSchemaMutation: ModifyAssociatedDataSchemaNameMutation – změní název uloženého schématu nacházejícím se v kolekci entit.
- Entity [22]
    - EntityMutation: EntityUpsertMutation – vloží či upraví (v závislosti na existenci entity se stejnými identifikačními prvky) entitu do kolekce (databáze), tato mutace obsahuje seznam mutací k provedení, jež jsou typu *LocalMutation*;
    - LocalMutation: UpsertPriceMutation – vloží či upraví (v závislosti na existenci ceny se stejnými identifikačními prvky) cenu na entitu;

- SchemaEvolvingLocalMutation: HierarchicalPlacementMutation – modifikuje hierarchické chování entity; pod toto rozhraní spadají všechny mutace, které mohou za předpokladu povoleného evolučního módu automaticky rozšířit schéma při vkládání, resp. úpravách entity.

Jako jeden z mechanismů pro ulehčení práce s mutacemi v rámci provádění transakcí byl zaveden koncept verzování realizovaný rozhraním *Versioned*, jenž byl inspirován technikou Optimistic Concurrency Control[6]. Verzi v podobě celočíselné informace si udržuje každý objekt, který může být změněn pomocí mutací a toto číslo umožňuje vyhodnotit možný konflikt mutací při jejich aplikaci. Disponují jí tedy všechny typy schémat, samotné entity i všechna její přidružená data, atributy počínaje a cenami konče, viz. kapitola 3.4. [25]

3.7. Query

Provádění složitých vyhledávacích dotazů na databázi patří k jejím primárním účelům. Pro provádění těchto dotazů byl vytvořen koncept realizovaný třídou Query, která obsahuje čtveřici parametrů, pomocí nichž vzniká nová instance dotazu. Tyto části dotazu představují v prováděném dotazu omezení a pravidla kladená na výsledně vrácená data. Všechny databázové funkce použitelné při vyhledávání jsou v evitaDB definovány rozhraním Constraint. Pro jejich jednodušší použití z programovacího jazyka Java je možné použít statické metody na rozhraní *QueryConstraints*, které v kombinaci se statickým importem umožní kratší a přehledný zápis. Níže jsou vysvětleny všechny základní části dotazu společně s vysvětlením, k čemu se používají a co smí obsahovat: [18]

- collection() – přijímá název kolekce, nad kterou bude dotaz proveden,
- filterBy() – představuje kontejner přijímající libovolnou instanci filtrační podmínky implementující rozhraní FilterConstraint, slouží k omezení vrácených entit splňujících specifikované podmínky,
- orderBy() – kontejner, v němž je možné specifikovat jedno nebo více pravidel pro třídění vrácených entit ve formě tzv. OrderConstraint,
- require() – kontejner, do něhož lze předat libovolný počet rozšiřujících požadavků implementujících rozhraní RequireConstraint, pomocí kterých lze například upravit

[6] Optimistic Concurrency Control – technika používaná při transakčním zpracování, jenž bez nutnosti synchronizace s úložištěm dat dovoluje posílat ke zpracování i větší počet transakcí, které jsou před jejich aplikováním a zavedením změn do databáze validovány – v případě konfliktu jsou problémové transakce odmítnuty. [24]

navrácená data, nastavit bohatost navrácených entit z hlediska obsahu jejich přidružených dat, upravit chování výpočtu facetů, definovat požadavky na stránkování výsledků nebo specifikování některého z dalších dodatečně vypočtených výsledků (*ExtraResults*).

V následujícím seznamu se nacházejí vybrané constrainty spadající pod zmíněná rozhraní pro použití v databázových dotazech s vysvětlením jejich efektu na vrácená data: [18]

- Collection – specifikuje kolekci, v níž bude daný dotaz vyhledávat, v případě její absence ji lze nahradit filtrační podmínkou na globálně unikátní atribut;
- FilterConstraint – představuje různé podmínky sloužící k omezení počtu nalezených záznamů, např:
    - and() – kontejner akceptující variabilní počet parametrů v podobě dalších filtračních podmínek, přičemž při aplikování musí platit všechny současně,
    - or() – kontejner přijímající filtrační constrainty, přičemž postačí, aby byla splněna alespoň jedna z nich,
    - not() – možné předat právě jednu filtrační constraintu, jejíž význam bude znegován,
    - attributeContains() – slouží pro výběr takových entit, které mají zadaný parametr s hodnotou obsahující část předaného textového parametru,
    - facetHaving() – filtruje entity, které mají referenci zadaného typu s alespoň jedním ze zadaných primárních klíčů, alternativně lze uvést i množinu filtračních podmínek,
    - hierarchyWithin() – navrátí pouze entity, které mají referenci zadaného hierarchického typu a zároveň samy z hlediska hierarchie spadají do stromu dané hierarchické entity specifikované primárním klíčem nebo jinou filtrační podmínkou,
    - priceBetween() – vyfiltruje entity, jejichž cena spadá do intervalu specifikovaného jedním nebo dvěma vstupními parametry,
    - referenceHaving() – kontejner umožňující filtraci nad referencemi specifikovaného typu,
    - userFilter() – kontejner obsahující požadavky k filtraci dle parametrů zadaných uživatelem (například z webového rozhraní);
- OrderConstraint – definuje možnosti, pomocí kterých lze přizpůsobit pořadí navrácených položek:

    - attributeNatural() – umožňuje seřadit entity podle hodnoty specifikovaného atributu dotazované entity, je možné pomocí výčtového typu *OrderDirection* změnit směr řazení vzestupně nebo sestupně (ASC/DESC),
    - priceNatural() – seřadí entity dle spočtené priority navrácených cen, opět možné použít *OrderDirection* (ASC/DESC),
    - referenceProperty() – kontejner přijímající název reference, podle jejíhož atributu budou entity řazeny, a libovolný počet řadících constraint;
- RequireConstraint – požadavek na změnu formy navrácených záznamů, případně umožňuje spočítání dodatečných dat (*ExtraResults*):
    - entityFetch() – při jejím použití je možné specifikovat požadovanou bohatost navrácených entit zadáním objektů realizujících rozhraní *EntityContentRequire*, například předáním parametru attributeContent() budou navrácené entity typu *SealedEntity* a budou obsahovat atributy; při nespecifikování constrainty entityFetch() budou navráceny pouze identifikátory entity reprezentované typem *EntityReference*,
    - dataInLocales() – lze specifikovat lokalizace (jazyky), v nichž mají být data u navrácených entit lokalizována,
    - facetGroupsDisjunction() – umožní u zadaného entitního typu odpovídající skupině facetů změnit logickou vazbu pro filtrační chování facetů mezi těmito skupinami, je nutné zadat i identifikátory skupiny, na jejíž facety bude tato logika aplikována,
    - page() – přijímá dva parametry v podobě stránky a její velikosti, dle zvolených argumentů bude vrácen odpovídající "výsek" entit splňujících ostatní podmínky v dotazu,
    - strip() – lze specifikovat (offset) počet entit k přeskočení od počátku seznamu všech entit, které splňují filtrační podmínky, a počet entit k navrácení - alternativa ke constraintě *page()*,
    - useOfPrice() – na vstupu je možné pomocí výčtového typu nastavit, nad jakou částkou ceny mají být vyhodnocovány filtrační podmínky – zdali má být pro výpočet použita cena s daní, nebo bez daně.

V sekci RequireConstraint byly vynechány výsledky patřící do skupiny *ExtraResult*, které zde budou vysvětleny samostatně. Celkově evitaDB nabízí 6 různých typů spadajících do této kategorie. Jako první lze zmínit *QueryTelemetry*, který obsahuje struktur zpracování dotazu

na straně databáze dobu trvání dílčích kroků v průběhu výpočtu dotazu, jenž je alternativou k EXPLAIN v relačních databázích.

K dispozici jsou také dva typy požadavků (RequireConstraint) umožňující výpočet histogramů hodnot vyfiltrovaných entit. Databáze umožňuje výpočet histogramů, které reflektují četnost navrácených entit v různých cenových hladinách (*PriceHistogram*) a konkrétních hladinách číselných atributů (*AttributeHistogram*). Takto vypočtené histogramy lze snadno zobrazit u posuvníků sloužících k úpravám příslušných parametrů v uživatelském rozhraní a lze dosáhnout zajímavé funkcionality, kterou disponuje velmi malá část aktuálně provozovaných řešení. [18]

Dále existuje požadavek typu *FacetSummary*, které slouží k získání statistik o všech facetech (parametrech) entit odpovídajících filtrovacím podmínkám dotazu. V základním nastavení je pro každý facet spočítán počet na něj se odkazujících entit. Přes argument požadavku je možné nastavit alternativní režim, který u facetů zobrazí i rozdíl (kladné nebo záporné číslo) v počtu zobrazených entit, k němuž dojde při jeho zařazení do filtru, respektive zrušení jeho výběru. Tato funkce je obsažena ve velkém množství e-shopů, ale pouze v prvním, jednodušším režimu. Zobrazování rozdílů u jednotlivých facetů ještě před jejich výběrem není až tak rozšířené a vídané kvůli výpočetní náročnosti především na e-shopech s větším počtem nabízených produktů, a to i přes nesporný přínos pro použitelnost řešení pro koncového uživatele. Jedním z důležitých faktorů, proč lze tuto funkci vídat na tak malém počtu e-shopů, je fakt, že tato funkce není vestavěnou součástí SQL či NoSQL databází, má netriviální komplexitu a vyžaduje ruční naprogramování na straně dodavatele e-commerce řešení. [18]

Poslední dva z dostupných rozšiřujících požadavků jsou spjaté s hierarchickými strukturami, konkrétně se jedná o *HierarchyStatistics* a *HierarchyParents*. První ze zmíněných slouží k výpočtu entit, na něž se odkazují všechny vyfiltrované entity dotazu. Tato funkce se používá především pro účely vykreslování hierarchických menu kategorií. Vypočtené menu respektuje hierarchická filtrační omezení z *filteryBy()*, tzn. některé části stromu mohou být z výpočtů vyloučeny. Podstatné je také zmínit, že se interně jedná o rekurzivní strukturu, není tedy dopředu známo, kolik úrovní bude spočítaný hierarchický strom obsahovat, ačkoliv si může vývojář s použitím dalších argumentů a vnitřních nastavení požadavku vypočtený strom dle potřeb omezit či rozšířit. Bez zadání dalších parametrů by každá z hierarchických entit obsahovala seznam pouze primárních identifikátorů entity, na které se vyhledané entity odkazují. Při zadání dodatečných *EntityContentRequire* požadavků pro obohacení entity budou tyto primární klíče nahrazenými celými entitami s vyžádanou sadou dat. Zároveň se

ve vypočteném stromě u každé hierarchické entity nachází i další statistické hodnoty, které se dají využít v uživatelském rozhraní. Jedná se o počet entit vrácených aktuálním dotazem, který pod konkrétní uzel stromové struktury spadá, a také počet podřízených uzlů stromu, v nichž byla nalezena alespoň jedna entita. Tato funkce může být nápomocná při vykreslování dynamického menu ve webovém rozhraní e-shopu s možností některé jeho části vynechat. Velmi podobného, ale opačného, efektu lze dosáhnout pomocí požadavku *HierarchyParents*. Ten u hierarchických entit vrátí seznam předků ve formě hierarchických entit, na něž se odkazuje pomocí referencí. Tato informace může sloužit například při vykreslování drobečkového menu na webu reprezentujícího zařazení produktu v kategorii nebo skupině. [18]

Na ukázce níže je příklad složitějšího dotazu na evitaDB databázi, konkrétně se jedná o dotaz na kolekci typu "Product". Ve *filterBy()* podmínce je pomocí kontejneru *and()* požadováno, aby všechny z následujících podmínek platily zároveň pro každou navrácenou entitu: musí mít ceny v české koruně, které jsou z ceníku "vip" nebo "basic" – přičemž první zmíněný má vyšší prioritu, musí být aktuálně platné a v cenovém rozmezí 1–10 korun. Zároveň musí odkazovat na některou z entit typu "Category", která v hierarchickém stromě spadá pod kategorii s primárním klíčem 4. Od uživatele jsou navoleny filtry pro zobrazení pouze položek majících atribut "capacity" s hodnotou větší nebo rovno 10 a zároveň mají i některý facet typu "parameter" s primárními klíči 1, 3 nebo 7. Navrácené entity budou seřazeny sestupně podle výsledných cen, přičemž z výsledného seznamu se jich vrátí prvních 20. Entity budou disponovat atributy, cenami a referencemi typu "brand" se všemi atributy na této relaci. Spočítány a vráceny budou také dva ExtraResulty, histogram rozdělující četnost entit podle jejich cen a shrnutí a statistiky facetů s režimem pro vypočítání rozdílů v počtu vrácených entit při jejich zařazení do filtru.

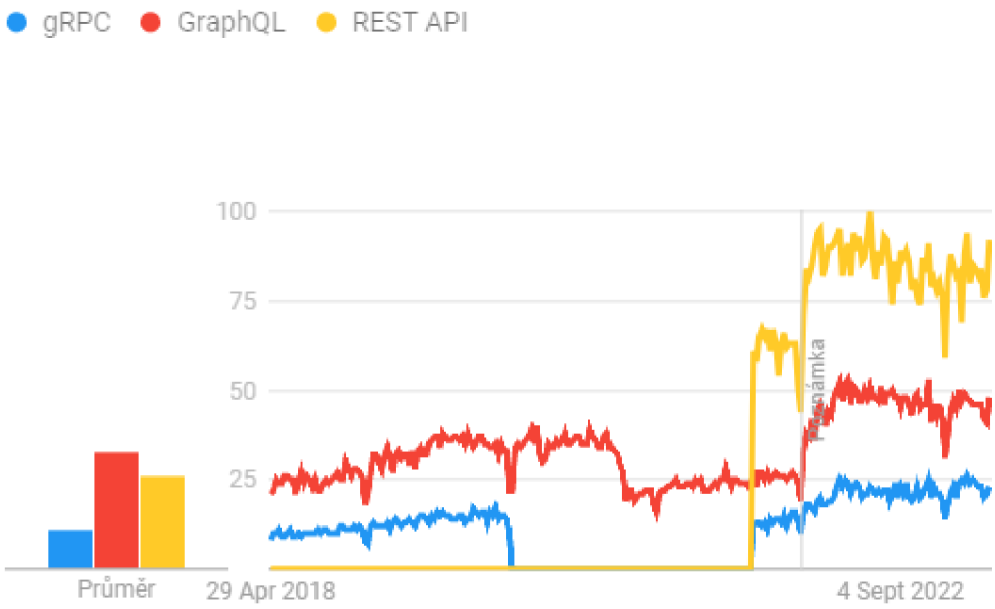
```
query(
   collection("Product"),
   filterBy(
      and(
         priceInCurrency(Currency.getInstance("CZK")),
         priceInPriceLists("vip", "basic"),
         priceValidNow(),
         priceBetween(BigDecimal.ONE, BigDecimal.TEN),
         hierarchyWithin("Category", entityPrimaryKeyInSet(4)),
         userFilter(
            attributeGreaterThanEquals("capacity", 10),
            facetHaving("parameter", entityPrimaryKeyInSet(1, 3, 7))
         )
      )
   ),
   orderBy(
      entityProperty(priceNatural(OrderDirection.DESC))
   ),
   require(
```

```
        page(1, 20),
        entityFetch(
           attributeContent(),
           referenceContent("brand", entityFetch(attributeContent())),
           priceContent()
        ),
        priceHistogram(20),
        facetSummary(FacetStatisticsDepth.IMPACT)
    )
);
```

Ukázka kódu 1 - Příklad složitějšího query na evitaDB databázi. Zdroj: [autor]

4. Rozbor a porovnání protokolů (API)

V minulé kapitole byla představena evitaDB databáze, byly vysvětleny její stěžejní principy fungování a proběhlo také vymezení základních pojmů, jejichž znalost je pro návrhovou a implementační část této práce považována za nezbytnou. Zmíněny byly i plány uvést do provozu trojici API, které by měly operovat nad databází a poskytnout jejich konzumentům co nejlepší možnosti pro vývoj. V současné době jsou dvě z nich (REST API a GraphQL) plně funkční, přičemž byly navrženy co nejpodobnějším způsobem tak, aby byla zajištěna jak jejich snadnější údržba a realizace budoucích změn v samotné databázi, tak i usnadnění přechodu z jeho API na druhé při znalosti principů, na nichž je evitaDB založena. Cílem této kapitoly je popis základních vlastností těchto API, jejich obsluhy, manipulace s daty a také jejich obecné výhody a nevýhody, a to včetně třetího a zatím neimplementovaného gRPC API. V podkapitolách věnujících se jednotlivým API budou rozebírány jejich teoretické principy, v závěru bude pak rozebrána jejich charakteristika a zhodnocení přínosů ve spojení s evitaDB. U gRPC, které ještě v provozu není, proběhne shrnutí původních požadavků na výsledné API a zhodnocení jejich relevance s možnostmi samotného gRPC vzhledem k programové architektuře evitaDB databáze. V závěru kapitoly dojde k finální specifikaci implementace API uzpůsobené zjištěným principům fungování gRPC.



Graf 1 - Porovnání vyhledávanosti řešených API z Google Trends. Zdroj: [26]

Na výše uvedeném grafu získaném z Google Trends představuje popularitu jednotlivých API z hlediska jejich vyhledávanosti přes nejpoužívanější vyhledávací engine od společnosti

Google. Jasně z něj vyplývá, že největšímu zájmu se těší REST API, s přibližně poloviční oblibou druhé místo zaujímá GraphQL a třetí místo obsadilo gRPC s poloviční vyhledávaností oproti předchozímu zmíněnému API. Toto srovnání je spíše informativní pro získání lepšího povědomí o popularitě nejznámějších řešení pro budování API, výsledek z hlediska kvality provedení nebo možností daného API nemusí být zcela směrodatný. Každé z nich má totiž trochu odlišné zaměření a cílí na odlišné scénáře, případy užití či použité technologie, jež je mají provozovat nebo konzumovat. Žádné z API samozřejmě není dokonalé, každé má ve srovnání s ostatními své vlastní přednosti i slabiny, a i typické scénáře použití.

V úvodu do této problematiky je také podstatné zmínit, že ať už je řeč o kterémkoliv ze zmiňovaných typů API, vždy jsou provozovány na některém z HTTP serverů, které slouží ke zpracování a odbavení požadavků přes HTTP protokol. Těchto serverů existuje velké množství, výběr může značně ovlivnit výkon na něm provozovaných služeb, a to včetně API. Před počátkem vývoje samotných API proběhlo evaluační testování deseti nejznámějších HTTP serverů, které lze provozovat na JVM platformě. Toto testování bylo zaměřeno na výkon webového serveru z hlediska propustnosti (počet požadavků za sekundu). Z testování vyšel jako jeden z nejrychlejších a nejjednodušeji integrovatelných HTTP server Undertow, jenž byl také vybrán pro implementaci evitaDB API. Mezi hlavní cíle patřila především rychlost a jednoduchá integrace, ve druhé řadě pak bylo požadováno, aby všechna API mohla běžet na téže adrese na stejném portu. [27]

Obě již dokončená API na té nejobecnější úrovni sdílejí základní rysy z hlediska provozu a komunikace. Obě API odpovědi na dotazy standardně posílají ve formátu JSON, pro komunikaci používají protokol HTTP ve verzi 1.1, ale přistupují odlišně k používání HTTP metod a status kódů. Níže je uveden seznam základních nejpoužívanějších HTTP metod, které jsou určené na provádění CRUD[7] operací:

- GET – odpovídá operaci `Read`, slouží k získávání, respektive čtení dat,
- POST – jejím účelem je vkládat nová data, provádí `Create` operaci,
- PUT – je používána k úpravě záznamů, ekvivalent operace `Update`,
- DELETE – zajišťuje a obsluhuje požadavky pro mazání dat.

Zmíněné HTTP status kódy lze ve stručnosti představit tak, že jsou standardizovaným způsobem sloužícím pro předání informace o výsledku dotazu na server, přičemž jejich hodnota přichází klientovi vždy společně se získanou odpovědí. Kódy se dělí do pěti skupin podle toho,

[7] CRUD zkratka složená z názvů 4 základní operací (Create, Read, Update, Delete)

jaký je výsledek zpracování klientova požadavku, a každá ze skupin má alokovaných 100 míst pro konkrétní kódy. První skupina (100–199) značí odpověď informačního charakteru, druhá skupina (200–299) informuje o úspěšném provedení požadavku, třetí (3XX) pak klientovi sděluje, že došlo k přesměrování či přesunutí požadovaného obsahu. Čtvrtá skupina (4XX) informuje o chybách na klientské straně, pátá (5XX) o chybách na straně serveru. Mezi nejčastěji používané kódy patří například: 200 (úspěšný požadavek), 201 (úspěšně vytvořený záznam), 400 (špatný dotaz ze strany klienta), 401 (chybějící přihlašovací údaje), 403 (nedostatečná oprávnění), 404 (obsah nenalezen), 500 (chyba na straně serveru). [28]

Formát JSON neboli JavaScript Object Notation, jak již delší tvar názvu napovídá, má svůj původ zakořeněný v jazyku JavaScript, nicméně se postupem času stal univerzálním, jazykově nezávislým a v dnešní době i nejrozšířenějším formátem na internetu sloužícím například pro serializaci dat. Je hojně používaný při ukládání dat nebo při práci s webovými API. Podporované typy vychází z jazyka JavaScript, patří mezi ně: [29]

- *Number*, jenž může uchovávat celá i desetinná čísla,
- *String*, do něhož lze vkládat libovolný počet znaků (textové řetězce),
- *Boolean* obsahující hodnoty true/false
- objekt reprezentovaný množinou vlastností realizovaný dvojicemi klíč-hodnota (klíč v textové podobě představuje název vlastnosti, hodnota libovolného podporovaného typu),
- pole libovolných podporovaných elementů,
- hodnota *null* značící prázdnou, resp. nezadanou hodnotu.

Na ukázce níže je příklad objektu serializovaného do formátu JSON reprezentujícího osobu. Je na ní demonstrováno použití všech obecně podporovaných datových typů, kterých JSON pro zachování kompatibility moc nemá, viz. výčet výše.

```
{
    "name": "Joe Williams",
    "age": 30,
    "employed": true,
    "salary": 4523.25,
    "car": null,
    "address": {
        "streetAddress": "6W 62nd St",
        "city": "New York",
        "state": "NY",
        "country": "USA",
```

```
        "postalCode": "10023"
    },
    "hobbies": [
        "swimming",
        "running",
        "reading"
    ],
    "skills": {
        "html": "advanced",
        "css": "advanced",
        "javascript": "intermediate"
    }
}
```

Ukázka kódu 2 - Ukázka objektu v JSON formátu. Zdroj: [autor]

Na uvedené ukázce výše si lze povšimnout, že u vlastnosti *salary* je číselná hodnota předána ve tvaru s plovoucí desetinnou čárkou. S čísly je ve formátu JSON spojen problém, který vychází z jeho původu, tedy jazyku JavaScript. Jak již bylo výše uvedeno, pro všechny typy čísel je používán JavaScriptový datový typ *Number*, který je ovšem omezen velikostí 53 bitů, takže pomocí něj není možné přesně pokrýt běžně používané číselné typy v jiných programovacích jazycích, které tuto velikost mohou přesáhnout. Příkladem z prostředí Javy je číselný typ Long (64 bitů), Double (64 bitů) nebo BigDecimal (s prakticky neomezenou šířkou). Další možností reprezentace velkých/přesnějších čísel je obalit hodnotu do textového řetězce, což umožní zachovat absolutní přesnost serializované hodnoty. Převod na koncový číselný typ, je pak na záležitosti možností platformy, na které je JSON formát parsován. [30]

4.1. Výhody a nevýhody REST API

REST API je dozajista z existujících formátů API nejpoužívanější a nejrozšířenější. V zásadě funguje na synchronní bázi – na server přijde požadavek, ten je zpracován a následně je vrácena požadovaná odpověď, obvykle ve formátu JSON; alternativou může být například formát XML, který ale v tomto směru v průběhu let ztrácí na popularitě. [31]

Mezi výhody patří jednoduchost jak samotné implementace API na straně serveru, tak i jeho používání z role klienta. Dále pak přímočarost a univerzálnost a obecné povědomí, což pravděpodobně souvisí i se stářím REST API, které vzniklo v roce 2000. V dnešní době je velmi jednoduché vytvořit základní API, například pro zprostředkování dat k webové či mobilní aplikaci. V některých technologiích stačí pouze pár řádků na vystavení koncového bodu (endpointu) vytvářeného API, který lze stejně rychle zavolat z klientské strany pro docílení komunikace mezi oběma stranami. Ve srovnání s ostatními zmiňovanými API se těší

jednoznačně největší podpoře z hlediska technologií a programovacích jazyků. Lze jej jak vystavit, tak i konzumovat téměř z jakékoliv technologie, protože plně využívá funkcí a vlastností HTTP protokolu. Na klientské straně existuje celá řada nástrojů (jako například *curl*) pro konzumaci API přímo z příkazové řádky nebo dokonce i v omezené míře přes běžný webový prohlížeč, které umožňují s API komunikovat bez nutnosti programování aplikace. [31]

Jako nevýhody lze uvést hned několik příkladů. Za odeslaným objektem v odpovědi většinou stojí nějaký datový model, například v podobě třídy popisující strukturu dat k serializaci a následnému zaslání uživateli. Klientská strana jako odpověď vždy obdrží model definovaný a specifikovaný vývojářem API, nelze si tedy zažádat pouze o jeho část. Pokud to API umožňuje, jedná se již o vlastní nadstavbu nad REST protokolem, která není nijak specifikovaná. Stejně tak si nelze jednoduše v rámci jednoho dotazu na server říct o více zřetězených dat (dávkové zpracování různých zdrojů). Pro dosažení této modularity by bylo nezbytné vystavit velký počet koncových bodů pro všechny možné scénáře, kterých může být velmi mnoho, a i tak to nemusí pokrýt všechny scénáře daného uživatele.

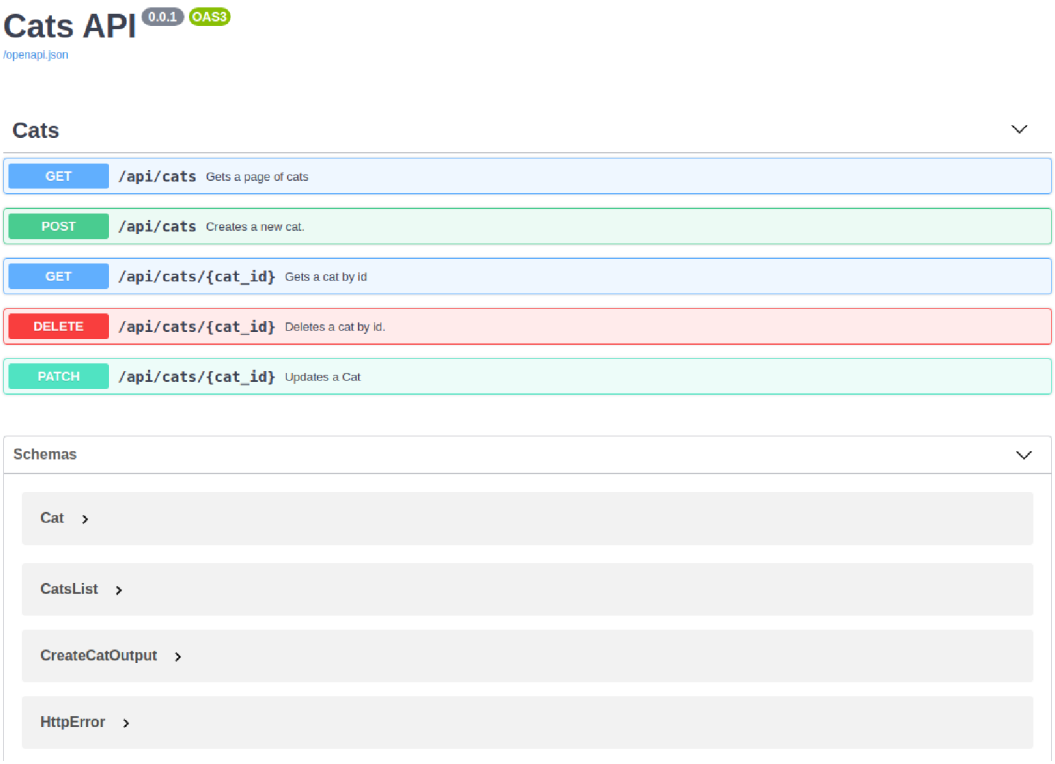
Na níže uvedené ukázce je jednoduchý JSON objekt, jenž obsahuje pole objektů skládajících se ze dvou vlastností: *firstName* a *lastName*. Pro lepší zdůraznění byly na ukázce červeně vyznačeny nadbytečné redundantní názvy vlastností, které jsou s JSON formátem obecně spjaté, tedy zbytečné a opakované specifikování názvů proměnných v rámci kolekcí stejných typů objektů.

```
{
    "buddies": [
        {
            "firstName": "John",
            "lastName": "Smith"
        },
        {
            "firstName": "Ray",
            "lastName": "Johnson"
        },
        {
            "firstName": "William",
            "lastName": "Jones"
        }
    ]
}
```

Ukázka kódu 3 - Ukázka redundancí obsažených v JSON formátu. Zdroj: [autor]

Pro API architektonicky stavící na REST principech je možné vygenerovat specifikaci OpenAPI poskytující jasnou a přehlednou dokumentaci obsahující pro každý jeden endpoint

definovaný popis, vstupní i návratový datový model a seznam všech možných návratových HTTP status kódů. Tato specifikace může být také vstupem pro různé testovací nástroje, které mohou sloužit pro testování již existujícího API nebo mohou i na jejím základě vytvořit celé API, proti němuž lze do určité míry vyvíjet klientské aplikace. [32]



Obrázek 3 - Příklad podoby OpenAPI specifikace. Zdroj: [33]

Podstatné je také zmínit, kdy konkrétně lze o API říct, že se jedná o REST API. Tímto označením lze identifikovat architektonický styl návrhu API, který je definován množinou zásad, principů a omezení. Jestliže navrhnuté REST API splňuje všechny ze stanovených podmínek, lze jej označovat jako RESTful API. Vhodné je také podotknout, že ne v každém případě lze REST API navrhnout tak, aby splňovalo všechny požadované zásady, ačkoli jejich splnění vede k jednoduchému, škálovatelnému a flexibilnímu API. Níže je seznam základních principů, na nichž REST API stojí. [31]

- Jednotná rozhraní: používáním obecného přístupu z hlediska veřejných kontraktů lze dosáhnout zjednodušené architektury.
- Klient-Server: použití tohoto návrhového vzoru lze zajistit odstínění klienta od serveru, což zajistí možnost jejich vývoje nezávisle na sobě – lze spoléhat čistě na rozhraní v podobě kontraktů, která vymezují nutné funkcionality, které server musí implementovat, a tudíž se na jejich existenci může klient spolehnout.

- Bezstavovost: RESTful API by nemělo z perspektivy uživatele uchovávat v rámci jejich opakované komunikace vnitřní stav, jeho požadavky by měly obsahovat veškeré informace a úkony, které mají být serverem provedeny bez znalosti jejich předchozí komunikace.
- Možnost ukládání a znovupoužitelnosti odpovědí (cachovatelnost): odpovědi mohou být jednotným standardizovaným způsobem v modelu označeny jako znovupoužitelné; aby mohly být použity znovu, například v dalších dotazech, REST API těží na velké závislosti na HTTP protokolu, jenž sám o sobě podporu pro cachování nabízí.
- Vrstvený systém: logiku API lze segmentovat na menší samostatné prvky, které mezi sebou účelově komunikují, ale klient komunikuje pouze s API – o tom, že jeho požadavek vyvolá akce na více různých serverech, na nichž je například prováděn či zpracováván jeho dotaz, nemá tušení.

V poslední řadě tu je ještě jedna nepovinná zásada umožňující REST API navrátit kromě klasických odpovědí v používaných formátech i spustitelný kód pro vykonání určitých změn na straně klienta [31].

Níže jsou uvedeny ukázky dotazů na REST API nacházející se na hypotetické adrese *www.example.com* za předpokladu, že v úložišti, jenž uchovává vložené záznamy o osobách, viz. Ukázka kódu 2, existuje interní číselný identifikátor každého záznamu a zároveň na zmíněné adrese existují a jsou veřejně vystavené koncové body, a to přímo na kořenové adrese.

- Pomocí GET metody: *http://example.com/persons/5*.
    - U GET metody se parametry objevují přímo v adrese. Mohou být přímou součástí adresy, jako například hodnota `5` v příkladu výše, nebo se může jednat o query parametry, kterých může být neomezeně, definují se tak, že za konec základní URL adresy je přidán tzv. "query string" – tj. otazník následovaný názvem parametru s rovnítkem a konkrétní hodnotou. Další parametry jsou definovány stejným způsobem, pouze počínaje druhým se otazník nahrazuje znakem `&`.
- Pomocí POST metody: *http://example.com/persons/*.
    - U POST metody není většinou potřebné zadávat parametry, atributy zaslaného objektu jsou poslány v těle požadavku ve tvaru JSON objektu.

4.2. Výhody a nevýhody GraphQL

Ve srovnání s REST API je GraphQL výrazně novější technologií, jeho první stabilní verze byla vydána společností Facebook (dnes Meta) v roce 2015. K problematice zprostředkování dat ze serveru klientům se také postavilo velmi odlišným způsobem. Jak název této technologie napovídá, její název a principy úzce souvisí s matematickým oborem Teorie grafu, v němž je možné pomocí grafů skládajících se z uzlů a hran modelovat víceméně jakoukoliv datovou strukturu. Nedílnou součástí GraphQL, kromě samotné implementace API, je i vlastní dotazovací jazyk, v některých aspektech podobný formátu JSON, který je obohacen o větší flexibilitu a možnosti specifikace požadovaných dat a jejich formátu v odpovědi. GraphQL také, kromě samotného definování dotazů, umožňuje specifikaci datových a objektových typů, které slouží k návrhu schémat, pomocí nichž je modelována celá struktura API. Schéma se definuje zvlášť s vlastním jazykem, který JSONu podobný už není. [34, 35]

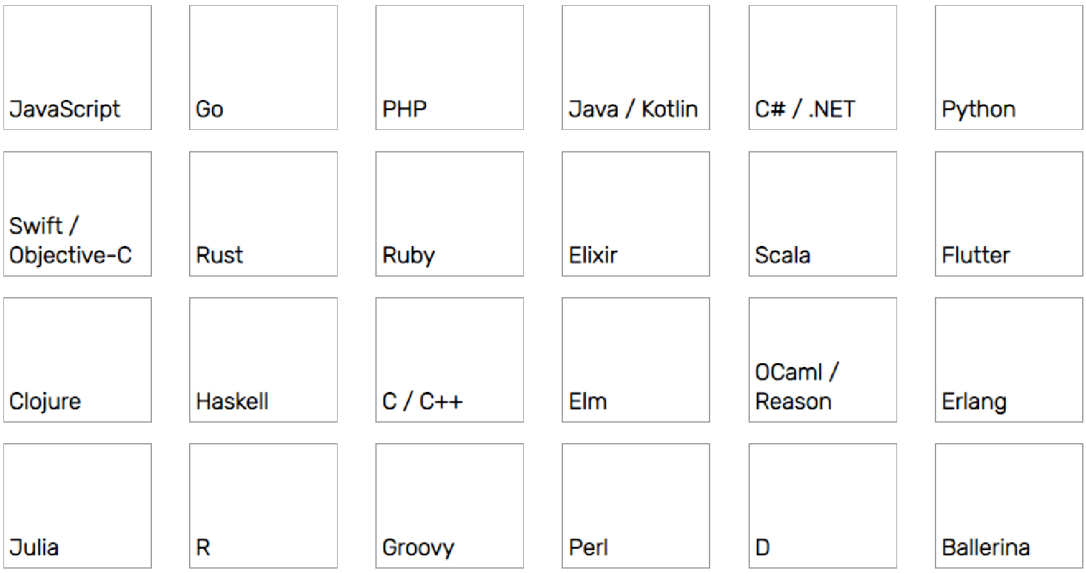
Právě se schématy souvisí jedna z hlavních výhod, mezi něž patří introspekce vystavená na stejném endpointu jako samotné API, díky níž mají klienti přístup ke GraphQL schématu i dokumentaci. Standardizovaného způsobu pro získání schématu API využívá v řadě vývojářských nástrojů také funkce IntelliSense, jež je také známá také jako doplňování kódu či našeptávání. S její pomocí si je možné nechat zobrazit všechny dostupné a v daném kontextu použitelné vlastnosti či klíčová slova odpovídající obsahu schématu. [36] Z hlediska možného návrhu API spojeného s jeho dokumentací je také klíčovou výhodou možnost dynamického sestavování schémat ze spuštěné aplikace. Tím je možné dosáhnout podobného výsledku jako u REST API s generováním OpenAPI specifikace, nicméně GraphQL je (při použití z vhodné technologie) v tomto ohledu přívětivější a nabízí lepší možnosti pro dosažení požadovaného výsledku. Celý tento proces je zde znatelně jednodušší kvůli lepší podpoře dynamičnosti a adaptace na provozovanou doménu. Mezi další výhody spojené se schématem je také na místě zmínit validaci vstupů (například dotazů) na straně uživatele (za předpokladu používání chytrého IDE), jenž zamezuje zasílání nevalidních požadavků na server a přímo informuje o vyskytujících se chybách rozporujících zásady definované ve schématu, ať už jsou jakéhokoliv charakteru [37]. V porovnání s REST API je GraphQL obecně efektivnější, neboť jeho potenciál umožňuje dvě zásadní vlastnosti, které REST nenabízí. Jmenovitě se jedná o možnost si na klientské straně pomocí jazyka definovaným GraphQL technologií přesně vyžádat, které z vlastností požadovaného objektu mají být navráceny, což principiálně zmenšuje a zefektivňuje celou komunikaci, což u REST API není implicitně možné. Druhá ze zmíněných výhod, která také souvisí s komunikací, je možnost v rámci jednoho požadavku

na server definovat více dotazů, na něž v odpovědi obdrží, v podobě jediného JSON objektu, dílčí vnořené objekty představující požadované výsledky.

Mezi nevýhody lze zařadit například odlišný způsob vracení chyb ze serveru uživateli. Jak již bylo řečeno, REST API pro chybové zprávy používá srozumitelné HTTP status kódy, GraphQL nikoliv. To může být pro nového programátora používajícího tuto technologii, a to ať už z pozice klienta, nebo serveru, dost zmatečné. Všechny požadavky se v GraphQL obvykle posílají metodou POST a odpověď na ně vždy (když je server dostupný) vrací HTTP status kód 200 (OK), a to i v případě, že se při zpracování vyskytly nějaké problémy či chyby. Z pozice konzumenta tohoto API tedy nestačí kontrolovat status kód z odpovědi, ale je nutné kontrolovat existenci pole *errors*, v němž by byly případné chyby zaznamenány. [38, 39] Další nevýhodou může být například fakt, že celé API je zakryté pouze jediným koncovým bodem, přes který proudí veškerý provoz. To zesložiťuje možnosti škálování, protože neexistuje jednoduchá cesta, jak různé endpointy rozdělit na různé fyzické servery a dosáhnout tak alespoň částečného horizontálního škálování. Problematickou oblastí je také zabezpečení, v němž má REST díky možnosti separace endpointů na více URL adres jednoznačnou výhodu. Naopak GraphQL většinou využívá pro API pouze jediný koncový bod, a proto je nutné implementovat zabezpečení buď na úrovni API, nebo na jiné vrstvě aplikace. Z hlediska cachování má GraphQL opět ve srovnání s REST API nevýhodu, a to i přes vestavěné způsoby pro implementaci cachování dotazů. REST využívá unikátních URL adres pro každý z typů požadavků a používá standardizované HTTP metody, je možné navrácené výsledky snadno zavést do mezipaměti na různých síťových prvcích, což pro GraphQL neplatí. [39, 40] Jako poslední nevýhodu lze zmínit větší množství času potřebné k návrhu a implementaci GraphQL API v porovnání s REST přístupem. GraphQL bývá ve většině případů ve všech zmíněných parametrech srovnání náročnější jak z časového hlediska, tak i obtížnosti. Tím je ale zaplacena daň za širokou škálu výhod (většina z nich se nachází na straně klienta), které jsou jím, ve srovnání s REST API, nabízeny. [39]

GraphQL API je možné provozovat z velkého počtu programovacích jazyků, pro něž je dostupných mnoho oficiálně podporovaných knihoven. Pro potřeby tvorby samotného API na serverové straně je nezbytné některou z nich použít. Pro konzumaci API na straně klienta se lze bez knihovny obejít, jelikož komunikace, jak již bylo vysvětleno, je realizována běžnými HTTP požadavky, pomocí nichž lze API bez problémů používat z téměř jakéhokoli jazyka. Tímto přístupem je ovšem vývojář připraven o většinu výhod, které GraphQL přináší, jako například zmíněná validace vstupu, dokumentace, našeptávání povolených tokenů při psaní

dotazů, což programátorům bez zkušeností s touto technologií práci velmi komplikuje. Silné stránky GraphQL lze využít při použití některé z knihoven i na straně klienta. Ty jsou k dispozici pro širokou škálu programovacích jazyků, jejichž seznam je na obrázku níže. [41]



Obrázek 4 - Seznam programovacích jazyků s podporou GraphQL. Zdroj: [41]

Jak bylo zmíněno výše, z HTTP metod se v GraphQL nejčastěji používá metoda POST poskytující dotaz k provedení v těle HTTP požadavku. Pro odlišení typů požadavků z hlediska čtení/zápisu dat byla stanovena konvence, že dotazy provádějící čtení jsou označovány pojmem *query* (dotaz). Druhá skupina dotazů, ve které dochází k manipulaci s daty, je nazývána slovem *mutation* (mutace). Ve struktuře se kromě klíčového slova nijak neliší, u obou je možné specifikovat vstupní parametry a jako odpověď vracet vývojářem API specifikovaný objekt. V případě mutace se vrací jeho nová aktualizovaná podoba (ať už byl upravován, nebo vkládán nový). [42] Zároveň GraphQL poskytuje možnost přihlášení se k odběru (subscription) změn prováděných nad jedním či více záznamy. Skrze tento mechanismus lze docílit asynchronní komunikace se schopností automaticky dostávat informace o změnách ve sledovaných datech pomocí WebSocketů. [43]

Níže se nachází ukázka kódu týkající se definice nezbytných struktur k definici dotazu sloužícího k vyhledání člověka podle jeho číselného identifikátoru (všechny z uvedených typů by se nacházely ve schématu).

```
type Query {
  person(id: Int): Person
}

type Person {
  name: String!
  age: Int!
```

```
  employed: Boolean!
  salary: Float
  car: String
  address: Address
  hobbies: [String]
  skills: [Skills]
}

type Address {
  streetAddress: String!
  city: String!
  state: String
  country: String!
  postalCode: String!
}

type Skills {
  skillName: String!
  level: SkillLevel
}

enum SkillLevel {
        BEGINNER
  ADVANCED
  INTERMEDIATE
}
```

Ukázka kódu 4 - Definice typů pro vyhledání nad ukázkovým JSON souborem. Zdroj: [autor]

S výše uvedenou definicí umístěnou ve schématu by bylo možné provést dotaz se strany klienta určený k odeslání na server s cílem získání specifikovaných dat o člověku. Ukázka tohoto dotazu se nachází níže.

```
query {
  person(id: 5) {
    name
    salary
    skills {
      skillName
      level
    }
  }
}
```

Ukázka kódu 5 - Ukázka query k vyhledání člověka podle ID nad JSON souborem. Zdroj: [autor]

Pro zpracování dotazů na straně serveru musí pro každý definovaný typ existovat implementace tzv. *resolveru* určující způsob, jak reagovat na obdržený dotaz od klienta. U všech existujících typů musí být pro jimi deklarované vlastnosti určen způsob, jak na základě předaných parametrů získat jejich hodnotu.

4.3. Výhody a nevýhody gRPC

Framework gRPC, jenž pochází z dílny společnosti Google, je nejmladším ze zmíněných způsobů provozování API s prvotní vydanou verzí v roce 2016. [44] Na základě porovnání z Google Trends na začátku kapitoly bylo zjištěno, že je také mezi zde rozebíranými protokoly

nejméně vyhledávaný, což je pravděpodobně zapříčiněno například jeho specifickým přístupem ke komunikaci mezi klientem a serverem (na obou stranách většinou figuruje server) a také jeho odklonem od zažitých paradigmat používaných ostatními API, od kterých se liší i jeho zaměřením na komunikaci typu server-server, nikoliv klient-server, jak je jinak běžné. [45]

Jeho největším a nejvyhledávanějším přínosem je jednoznačně rychlost. Podle dostupných zdrojů na internetu má gRPC potenciál být až 7× rychlejší než REST API [46]. Nicméně samotné výkonnostní porovnání bylo provedeno ve srovnání s GraphQL jakožto prvním z vyvíjených řešení. Testování GraphQL bylo provedeno na HTTP serveru Undertow, který byl v rámci testování výkonnosti nejznámějších Java HTTP serverů vyhodnocen jako nejvhodnější. Pro informativní srovnání, gRPC Java knihovna existuje ve dvou provedeních, přičemž hlavní a udržovanější verze implicitně obsahuje HTTP server Netty, na němž je automaticky spouštěn gRPC server. Z Javy tedy není možné gRPC provozovat na Undertow serveru, proto bylo srovnávání provedeno na dvou odlišných HTTP serverech. I přes skutečnost, že Netty vyšlo v evaluačním testu HTTP serverů lépe než Undertow, ke změně nedošlo ze dvou hlavních důvodů: Netty má složitější API (hůře by se s ním pracovalo) a v původní implementaci porovnání testů byla chyba, která byla odhalena až ve chvíli, kdy byla již vytvořena velmi značná část na něm stavících API. [27] V tabulce níže lze vidět porovnání dosažené propustnosti v provedených operacích za sekundu testovaných API, ve kterém gRPC dosáhlo 6× vyššího (lepšího) výsledku, čímž byl praktickým testem potvrzen a ověřen rychlostní potenciál frameworku. [47]

Tabulka 2 - Výsledek výkonnostního testování měřícího propustnost pomocí echo dotazů mezi gRPC API na Netty a GraphQL na Undertow. Zdroj: [autor]

	GraphQL běžící na Undertow	gRPC běžící na Netty
Provedené operace	9 971	59 617

Schopnost gRPC dosahovat takto extrémně vysokého datového toku stojí na několika pilířích. Jedním z nich je způsob definování struktury přenášených dat – k serializaci a deserializaci je používán mechanismus zvaný Protocol buffers. [46] Popis datových struktur by šel připodobnit ke schématům z GraphQL, přičemž ani jedno z API neumožňuje zakomponování generiky a Protocol buffers navíc zamezuje i použití dědičnosti mezi definovanými typy. Popis datových struktur v Protocol buffers se provádí v *proto* souborech. Základem jsou tzv. zprávy (message), které definují seznam datových polí včetně specifikace jejich datových typů. [48] Kromě

datových struktur se v *proto* souborech také definují tzv. služby s definicí vstupních a výstupních zpráv, jinak nazývané také RPC (Remote-Procedure-Call) volání. Pomocí těchto služeb bude probíhat reálná komunikace mezi klientem a serverem. Jak uvedená anglická slova skrývající se za zkratkou napovídají, gRPC funguje na principu přímého volání metod nacházejících se na vzdáleném serveru. Každá ze služeb přijímá i vrací právě jeden parametr v podobě odpovídajícího typu zprávy. Druhým z výkonnostních pilířů je způsob, jakým jsou data posílána po síti mezi klientem a serverem. Pro transport totiž gRPC využívá protokol HTTP/2, přes který veškerá data proudí ve formátu binárních rámců. Tato nová verze HTTP protokolu byla navržena pro minimalizaci režie, zrychlení posílání a přijímání požadavků a obecnou optimalizaci síťového přenosu. Pro bezproblémovou komunikaci je vhodné, aby klient i server disponovali stejnými *proto* soubory definujícími komunikační protokol, aby byly obě strany schopné číst navzájem přijímané zprávy. Klient je v určité míře odolný vůči nově definovaným zprávám a službám na straně serveru, avšak při provedení změn je nutné myslet na zpětnou kompatibilitu. Nutno dodat, že pro použití *proto* souborů v některé z podporovaných technologií musí být tyto soubory nejprve pomocí *protoc* kompilátoru zkompilovány do nativních souborů odpovídajících použitému programovacímu jazyku, operačnímu systému a HW platformě. Na serverové straně musí být realizovány implementace vygenerovaných kontraktů představujících služby nadefinované v *proto* souborech. [46] Jako další výhodu je možné uvést i podporu vzájemného ověřování důvěryhodnosti protistrany na bázi výměny certifikátů mezi klientem a serverem (mTLS), což společně s TLS podporou HTTP/2 protokolu zvyšuje úroveň zabezpečení API. HTTP/2, na němž gRPC staví a sdílí s ním tedy i jeho principy a zásady, povoluje komunikaci pouze v režimu TLS (Transport-Layer-Security), tedy vynucuje alespoň základní bezpečnostní praktiky jako komunikaci. [49]

Vzhledem k tomu, že gRPC posílá po síti binárně serializovaná data, nemůže klient bezproblémově zavolat API odkudkoliv, jak tomu bylo u ostatních vysvětlovaných implementací, jelikož neví, jak přijímaná data číst a interpretovat. To na druhou stranu může být i výhodou přinášející dodatečnou úroveň zabezpečení, jelikož z principu může s API komunikovat, a tedy se ho pokusit kompromitovat, výrazně méně potenciálních útočníků. Za zmínku také stojí i problém s distribucí *proto* souborů a propagování změn ze strany serveru při zavedení nových funkcí. Pro jejich zajištění je totiž nutné, aby obě strany komunikace používaly kompatibilní schéma datových struktur, jakékoliv změny mohou tedy velmi snadno rozbít používané implementace konzumující gRPC API. Distribuce textových verzí *proto* souborů může být zajištěna například jejich vystavením na separátním HTTP endpointu, odkud

si je klient stáhne a zkompiluje. Existuje i alternativa k získání informací o gRPC službách a používaných datových typech s povolením serverové reflexe a vystavení samostatného endpointu pro publikování těchto dat. Tento přístup je velmi podobný introspekci v GraphQL, avšak zde je od něj obecně odrazováno, a to hned z několika důvodů: vystavuje veřejně informace o všech gRPC službách, což může být z hlediska zabezpečení nežádoucí a přidává nadbytečnou komunikační režii snižující výkon. Z toho důvodu by měl být používán především v rámci testování s použitím nástrojů, jako například grpCurl nebo Postman, které nepotřebují přímo programově pracovat se strukturami a službami, jež vznikají při kompilaci *proto* souborů. [50, 51]

Komunikaci lze v gRPC provozovat čtyřmi odlišnými způsoby podle typu a směru toku dat, přičemž každý je určený pro specifické účely. Seznam těchto variant je uvedeny ve výčtu níže [52]:

- Unary – klient odešle požadavek a na něj ihned po zpracování obdrží odpověď;
- Client-streaming – klient zahájí proces, při němž začne posílat proud dat na server a po jeho ukončení a zpracování server pošle jedinou odpověď;
- Server-streaming – klient pošle požadavek a server na něj reaguje otevřením proudu dat, po němž může posílat data, která klient může průběžně přijímat a zpracovávat;
- Bidirectional-streaming – kombinace posledních dvou výše uvedených principů; po obdržení klientského požadavku může jak klient, tak i server asynchronně posílat zprávy bez omezení a nutnosti čekání na příjem odpovědi na předešlé zprávy.

Podpora technologií u gRPC není zdaleka tak velká, jako tomu bylo u ostatních zmíněných API. Ve srovnání s GraphQL jejich celkový počet nedosahuje ani poloviny. Nicméně ty nejznámější a nejpopulárnější z nich podporu mají a jsou pro ně nabízeny, vyvíjeny a udržovány potřebné knihovny. Konkrétně je gRPC možné jak provozovat, tak i konzumovat z následujících programovacích jazyků: C# (.NET), C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python a Ruby. [53]

4.3.1. Protocol buffers

Protocol buffers je strukturovaný a jazykově neutrální formát datových struktur od společnosti Google, jehož první veřejná verze vyšla v roce 2008 [54, 55]. Protocol buffers byl navržen jako efektivnější a kompaktnější alternativa k JSON formátu, jelikož data serializuje do sekvence bajtů (bytes). Při práci s Protocol buffers v kombinaci s gRPC je (de)serializační logika již obsažena v použité knihovně a není ji tedy třeba řešit manuálně. [56]

Schéma v Protocol buffers klade důraz na pořadí, ve kterém jsou data předávána. Nemusí se tedy u každé zprávy uvádět název vlastností, jako tomu je u JSONu, komunikace je o tyto redundantní informace ušetřena. V *proto* souborech je možné uvést specifika a nastavení pro podporované jazyky, jako například projektové umístění generovaných nativních souboru (například Java package či C# namespace). Mezi základní vestavěné datové typy patří například: double, float, int32, int64, bool, string a bytes (reprezentuje pole bajtů). [57]

Tyto datové typy mohou být v případě potřeby, přidáním klíčového slova *repeated* před uvedený datový typ, použity i jako kompoziční typy pro pole či seznamy. Jednoduché datové typy mohou být použity v komplexním datovém typu představujícím mapu či slovník (z různých programovacích jazyků), které reprezentují množinu prvků ve tvaru klíč-hodnota – jeho zápis v Protocol buffers je realizován způsobem *map<typ-klíče, typ-hodnoty>*. Zároveň je možné i v rámci navrhované struktury zprávy do sebe vnořovat objekty a definovat výčtové typy klíčovým slovem *enum*. Existují i speciální datové typy nacházející se v knihovnách od společnosti Google, jako *Any* nebo *Oneof*, které je možné do *proto* souboru importovat a používat je. *Any* slouží k uchování libovolného serializovaného objektu, nevýhodou ovšem je, že subjekt na druhé straně komunikace musí znát definici původní třídy, aby jej mohl úspěšně deserializovat. Klíčové slovo *Oneof* slouží ke specifikaci jakéhokoli výčtu možností, z nichž bude použita (naplněna reálnými hodnotami) buď žádná, nebo právě jedna. Mezi další dodatečné typy patří například *Timestamp,* sloužící k ukládání data a času, obalující typy jako *StringValue* či *Int32Value*, které umožňují přenášet základní primitivní datové typy *string* a *int32* v podobě objektu, a tím volitelně odeslat i prázdnou hodnotu. Posledním ze zajímavých importovatelných typů je typ *Empty*, jehož použití například v parametru volané procedury značí, že jej spouštěná metoda na straně serveru nevyžaduje, respektive nevrací žádná data. [57] Na ukázce níže je definován protokol, pomocí něhož je možné posílat zprávy se stejným obsahem, jako na již zmíněné ukázce JSON objektu na Ukázka kódu 2.

```
syntax = "proto3";
import "google/protobuf/wrappers.proto";

message Person {
  string name = 1;
  int32 age = 2;
  bool employed = 3;
  double salary = 4;
  google.protobuf.StringValue car= 5;
  message Address {
    string streetAddress = 1;
    string city = 2;
    google.protobuf.StringValue state = 3;
    string country= 4;
    string postalCode = 5;
  }
```

```
  Address address = 6;
  repeated string hobbies = 7;
  map<string, LevelOfExpertise> skills = 8;
}

enum LevelOfExpertise {
  BEGINNER = 0;
  INTERMEDIATE = 1;
  ADVANCED = 2;
}

message PersonByIdRequest {
  int32 id = 1;
}

message PersonByIdResponse {
  Person person = 1;
}

service EvitaService {
  rpc GetPersonById(PersonByIdRequest) returns (PersonByIdResponse);
}
```

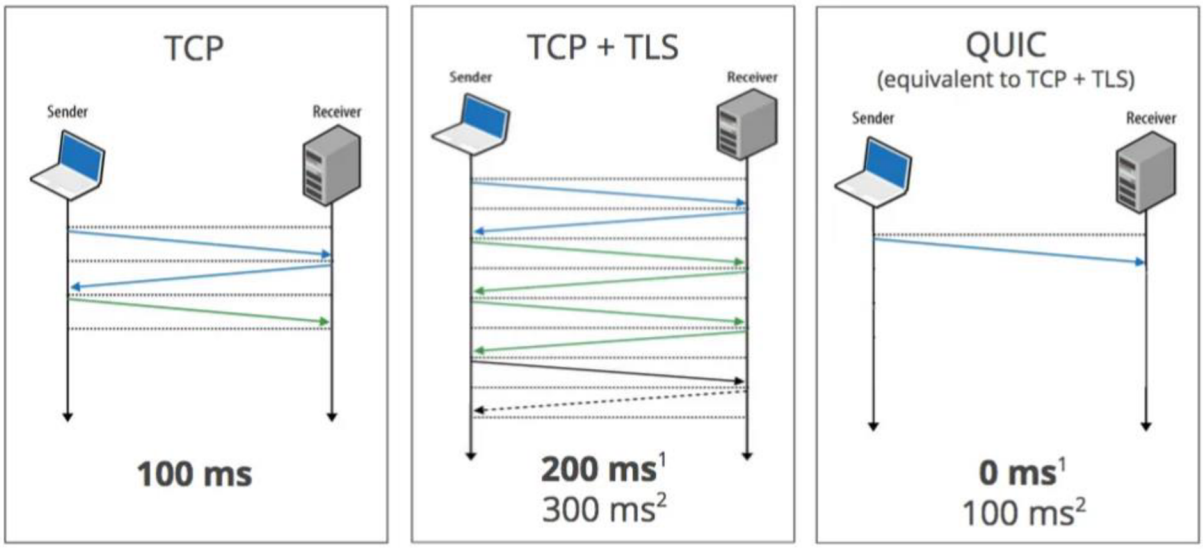
Ukázka kódu 6 - Ukázka definice zpráv a služby pomocí protobuf souboru. Zdroj: [autor]

4.3.2. Potenciál použití novějších verzí HTTP protokolu

V předchozí kapitole byl představen HTTP protokol verze 2, jenž odemkl dveře k mnoha možnostem pro zefektivnění a urychlení komunikace. Přinesl zlepšení na mnoha frontách, nicméně stále je založen na protokolu TCP, jenž má z hlediska maximalizace výkonu určitá omezení, jako například blokování čela fronty (Head-of-line-blocking) fungující na principu zásobníkového zpracování packetů, což značně omezuje výkonnostní potenciál protokolu. [58] HTTP/2 umožňuje částečné řešení tohoto problému na aplikační úrovni pomocí multiplexingu požadavků, na přenosové vrstvě, na níž operuje TCP protokol, problém ovšem zůstává. S řešením nejen tohoto problému přišel v roce 2022 inovativní HTTP/3, jenž tyto problémy vyřešil nahrazením TCP za transportní protokol QUIC, jenž je nadstavbou nad UDP. Hlavní rozdíl mezi TCP a UDP je především v přístupu k navazování spojení – TCP vyžaduje inicializaci spojení před zahájením komunikace, UDP to nevyžaduje. Vynechání inicializační fáze přináší celkové zrychlení komunikace a minimalizaci režie, nevýhodou je však ztráta spolehlivosti při doručování dat. [59]

HTTP/3 s využitím protokolu QIUC kombinuje ty nejlepší vlastnosti obou zmíněných transportních protokolů. Kombinací už tak rychlého HTTP/2 (v porovnání s běžně používaným HTTP /1.1) s bezpečnostní vrstvou TLS přes protokol TCP přináší, kromě již zmíněného problému Head-of-line-blocking, celou řadu zlepšení – především v minimalizaci času potřebného k navázání spojení a v zavedení mechanismů pro kontrolu ztráty dat a správu schopnou řešit přetížení síťové komunikace. [60]

Na obrázku níže se nachází porovnání procesu navazování spojení za použití protokolu TCP, jeho kombinace s bezpečnostní vrstvou TLS a protokolem QUIC. Důležitým faktem je to, že protokol QUIC již implicitně obsahuje bezpečnostní vrstvu a uvedené odezvy je tedy vhodné porovnávat s kombinací TCP + TLS, než základní, jednoduchou variantou TCP. Zároveň u druhého a třetího porovnání jsou uvedeny dvě naměřené hodnoty: první odpovídá času potřebnému k navázání spojení za předpokladu, že spolu již obě strany komunikovaly, druhá reprezentuje náročnost prvního požadavku.



Obrázek 5 - Srovnání rychlosti v navazování komunikace protokolů TCP a QUIC. Zdroj:[61]

Jak lze vidět, HTTP/3 přináší výrazné zrychlení, nicméně jeho aplikování v praxi často není možné kvůli nedostatečné adaptabilitě existujících technologií způsobené jeho nedávným oficiálním vydáním a zařazením mezi standardy. Kupříkladu gRPC, na něž je tato práce především zaměřena, má v tuto chvíli podporu protokolu HTTP/3 pouze ve variantě pro .NET, tedy pouze pro jednu z podporovaných platforem. [62]

4.4. Zhodnocení API z hlediska použití v evitaDB

Tato podkapitola bude věnována krátkému zhodnocení a porovnání již vytvořených API (REST a GraphQL API) z hlediska jejich funkčnosti a obsluhy evitaDB databáze. Proběhne sumarizace zjištěných faktů o gRPC, hlavně s přihlédnutím k původním požadavkům na implementovaná API, následovat bude finální rozhodnutí určující způsob, jímž bude toto API v rámci databáze využito. Je také vhodné zmínit, že pro REST API i GraphQL byl navržen vlastní dotazovací jazyk pro sjednocení stylu filtrace nad databází a také napomáhá klientům konzumovat daná

API. Zároveň schémata do jisté míry zpříjemňují psaní dotazů pomocí našeptávacích funkcí či zvýrazňování klíčových slov v rámci query dostupných v řadě vývojových prostředí.

4.4.1. REST API

Implementace REST API pro evitaDB je ve srovnání s GraphQL řešením poněkud těžkopádnějším a zdaleka ne tak příjemným pro konzumaci klienty. I přes schopnost vygenerování OpenAPI specifikace na míru pro provozovanou doménu nelze OpenAPI jednoduše používat pro snadné zobrazení dokumentace či našeptávání, jelikož je k tomu potřeba dodatečná konfigurace a použití správných nástrojů. Existuje celá řada nástrojů, které umožňují vygenerovat přehlednou dokumentaci s vestavěným testovacím prostředím, ale málokterý z nich podporuje zmíněné našeptávání. Ukázkové generované schéma v OpenAPI formátu pro běžnou e-commerce aplikaci dosahuje ohromného počtu řádků (přibližně 60 tisíc), což může, především na slabším hardwaru, velmi komplikovat manipulaci se zmíněným souborem. Této velikosti dosahuje hlavně kvůli separátním koncovým bodům pro různé jazykové varianty, čímž se sice zjednodušuje práce se samotným API, například linearizací návratové JSON struktury, ale ve schématu je nutné některé definice zduplikovat. Jako pozitivum lze zmínit dynamičnost registrovaných endpointů až na úroveň kolekcí entity.

Pro možnost dynamicky generovat OpenAPI specifikaci bylo nutné vytvořit nad existující knihovnou fasádu ve formě modulu, jelikož použitá knihovna není primárně uzpůsobena k manuálnímu generování OpenAPI schématu; všechno bylo interně uchováváno jako typ Schema (typ zastřešující všechny varianty podporovaných typů), tudíž každá metoda akceptovala jakýkoliv vstup bez možnosti validace. Knihovna je primárně určena ke generování OpenAPI schématu na základě anotací umístěných nad Java třídami, které mají být ve schématu zahrnuty. To při adaptaci na konkrétní klientskou doménu není možné, protože konkrétní datové modely, jež by měly být do OpenAPI zahrnuty (jako názvy katalogů a entity kolekcí), nejsou staticky definované; lze je získat až z běžící databáze ve speciální struktuře specifické pro evitaDB.

Generováním OpenAPI schématu bylo možné přiblížit se přístupu používaném v GraphQL knihovně, jež generování schémat při běhu aplikace dovoluje a má pro to velmi dobré nástroje i objektovou strukturu. Byla zavedena možnost vytvářet OpenAPI specifikaci pomocí vlastních builderů, které zajišťují lepší flexibilitu a zároveň typovou bezpečnost.

Na ukázce níže je tělo REST API požadavku reprezentující query, jenž má být provedeno nad testovacím katalogem *evita* na vystaveném koncovém bodě *https://demo.evitadb.io:5555/rest/evita/brand/query* pomocí HTTP metody GET:

```
{
    filterBy: {
        attributeCodeStartsWith: "a"
    },
    require: {
        page: {
            number: 1,
            size: 20
        },
        entityFetch: {
            attributeContent: "code"
        },
        queryTelemetry: true
    }
}
```

Ukázka kódu 7 - Příklad dotazu na REST API evitaDB databáze. Zdroj: [autor]

4.4.2. GraphQL

Vytvořené evitaDB GraphQL API je pro používání z pohledu vývojáře velmi příjemné. Schéma je generováno na základě databázových schémat a je možné si jej stáhnout z téměř každého IDE podporující práci s GraphQL. Díky aktuálnímu schématu si lze z jakékoliv pozice v query pomocí našeptávání zobrazit pouze aktuálně relevantní akceptovatelné vstupy a výstupy a zároveň je neustále prováděna na straně klienta validace tvořeného dotazu. Vývojář tedy prakticky nemůže vytvořit nevalidní databázový dotaz a při jeho tvorbě nijak nezatěžuje databázový server, protože validace probíhají kompletně na straně klienta vůči předem staženému schématu. Kromě toho je na některých místech výstupního modelu podporována i možnost nastavení formátu, v němž mají být hodnoty ze serveru vráceny – příkladem může být vrácení již naformátované ceny v konkrétní měně.

Za veškerý komfort poskytnutý při jeho používání se, bohužel, platí daň v podobě nižší rychlosti zpracování dotazů a navrácení výsledků. I přes to, že GraphQL je obecně považováno za efektivnější než REST API, vychází z provedených testů toto API jako výrazně pomalejší, a to zhruba o polovinu. Na vině je pravděpodobně serverové parsování vstupů a jejich následná validace. U bohatých objektů může být nevýhodou nutnost všechny vlastnosti objektu, které mají být vráceny, konkrétně vyjmenovat. Neexistuje snadná a rychlá možnost pro vrácení všech (i když některá IDE s tímto problémem pomáhají). Kvůli nutnosti přesně vydefinovat, co má server vrátit, je také problém při práci s rekurzivními strukturami s neznámou hloubkou, kde nelze manuálně specifikovat celou rekurzivní strukturu, jako tomu je například u datového struktury *HierarchyStatistics*. Takové struktury v GraphQL není možné ideálně namodelovat.

Problém však může být do jisté míry řešen pomocí klasické JSON struktury, díky čemuž lze zajistit podporu jakýchkoliv předem neznámých JSON objektů, ale přijde se pak o hlavní přednosti GraphQL, jako je kontextové našeptávání a zvýrazňování klíčových slov a hodnot – vrácený JSON je z perspektivy GraphQL pouhým nestrukturovaným textem.

Na ukázce níže je příklad volání GraphQL API provádějící dotaz nad koncovým bodem nacházejícím se na URL adrese *https://demo.evitadb.io:5555/gql/evita*, konkrétně nad dostupným testovacím katalogem *evita*:

```
query {

  queryBrand (
    filterBy: {
      attributeCodeStartsWith: "a"
    }
  ) {
   recordPage(number: 1, size: 20) {
     data {
       primaryKey
       attributes {
         code
       }
     }
   }
    extraResults {
      queryTelemetry
    }
  }
}
```

Ukázka kódu 8 - Příklad GraphQL dotazu na evitaDB databázi. Zdroj: [autor]

4.4.3. gRPC

Na základě průzkumu možností gRPC protokolu, které jsou uvedeny dále v této kapitole, bylo vyhodnoceno, že gRPC API není vhodné pro návrh klasického klientsky konzumovatelného API, které by bylo schopno uzpůsobit svoji strukturu na proměnlivý obsah získaný z běžící databáze, a to především z následujících důvodů:

- neexistuje rozumný způsob pro přizpůsobení API na databázové schéma – nulová adaptabilita, celé gRPC vychází z fixní specifikace komunikačního protokolu předem; generování této specifikace za běhu by sice pomocí některých komunitních nástrojů dostupných na internetu možné bylo, ale vedlo by k eliminaci většiny nabízených výhod (především rychlost) a bylo by v tomto provedení velmi špatně udržovatelné;
- nulová podpora dědičnosti a práce s generikami efektivně zamezuje jakýmkoliv pokusům o architektonické přiblížení se ostatním API;

- změny v protokolu jsou v porovnání s REST API a GraphQL hůře propagovatelné a implementovatelné na straně klientů z důvodu nutnosti zkompilování pozměněných *proto* souborů a vedly by k jejich častému "rozbíjení".

Kvůli výše zmíněným negativům bylo rozhodnuto o alternativním přístupu k implementaci tohoto API. Jedním z plánovaných úkolů bylo vytvořit v evitaDB Java ovladač, který by umožňoval databázi používat vzdáleně, a to s co nejmenší ztrátou rychlosti, jež gRPC nabízí. Zároveň se zdálo výhodné v ovladači zachovat stejné Java rozhraní jako při používání embedované verze evitaDB. Zprávy definované pomocí Protocol buffers by mohly v dostatečné míře pokrývat variabilitu databází podporovaných datových typů a s dostupnými mechanismy i vyjádřit generické typy, na nichž je evitaDB postavena. Na základě takto popsané struktury pomocí *proto* souborů budou vygenerovány třídy obsahující definované metody jak na serverové, tak i klientské straně. Z principu fungování gRPC bude klient volat metody na své straně, tím zajistí jejich provolání na serveru, čímž lze dosáhnout požadovaného výsledku s vysokým rychlostním potenciálem, kterým se gRPC vyznačuje. Jeho použití také umožní původně neplánovanou, ovšem vítanou možnost implementace ovladačů i pro jiné technologie, než je Java. Výstupem praktické části bude tedy gRPC API, které bude umožňovat vzdálené ovládání databáze co nejpodobnějším způsobem k existujícímu Java API v odlišené technologii, avšak s dodržením zažitých konvencí gRPC API.

Pro závěrečné shrnutí bude ještě uveden důvod, kvůli kterému je gRPC pro použití databázového ovladače vhodnější než REST API či GraphQL. Tak jako má gRPC problém s přizpůsobením se konkrétní provozované doméně, ostatní ze zmíněných API mají problém s obecným dotazováním bez znalosti konkrétní domény, v čemž gRPC naopak vyniká. Realizace ovladačů pomocí GraphQL nebo REST API by musela být vázána na konkrétní doménu narozdíl od gRPC, které díky své obecnosti může být provozováno v jakékoli doméně. Alternativně by bylo nezbytné vzdát se většiny jimi nabízených výhod a veškerou komunikaci provozovat na obecné úrovni bez specifikace konkrétních typů pro doménu, což by znamenalo ztrátu validací, našeptávání či zvýrazňování klíčových slov, bylo by nutné vystačit si pouze s naprostým základem v podobě JSON formátu.

5. Konzumace gRPC protokolu na straně klienta

V této kapitole je uveden souhrn vyžadovaných funkcionalit a dalších požadavků kladených na implementované ovladače. Rozebrány budou také hlavní důvody stojící za jejich vznikem, jejich obvyklé případy použití a proběhne sumarizace přínosů ovladačů při běžném provozu evitaDB v produkčním prostředí. V rámci rozebírané problematiky budou zmíněny také praktické ukázky konkrétního využití gRPC při tvorbě ovladačů.

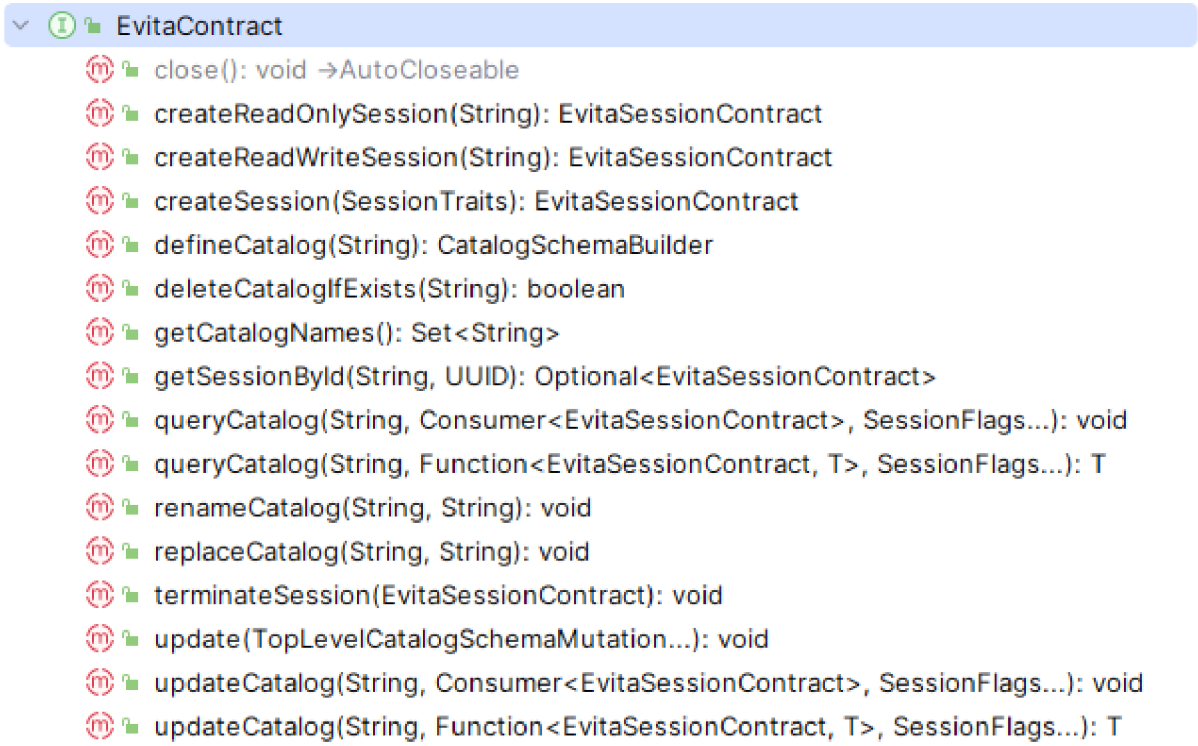
5.1. Požadavky na ovladače

I přes možnost provozovat evitaDB jako součást aplikace, která ji využívá, nejedná se o primární způsob jejího nasazení, jelikož přináší i řadu nevýhod. Typicky bude evitaDB umístěna na separátní fyzický server s vlastními (ideálně nesdílenými) hardwarovými prostředky, nebo s použitím dostupného Docker obrazu umožňujícím běh v izolovaném kontejnerizovaném prostředí. Oddělení aplikace a databáze umožní jejich separátní životní cyklus (spuštění, zastavení, aktualizace atp.), umožňuje databázi i aplikaci provozovat na různých verzích JVM, škálovat obě části samostatně atp. Kvůli tomuto rozdělení databáze a aplikace ale již nebylo možné využívat databázi napřímo v embedovaném režimu, v němž dosahuje svého maximálního výkonnostního potenciálu. Právě pro tyto účely vznikl požadavek na tvorbu ovladače, jenž by umožnil používání metod obsažených v evitaDB Java API a tím zajistil její stejné nativní ovládání i na vzdáleně provozované databázi s co nejnižší ztrátou výkonu. Právě pro tyto účely lze na straně databáze implementovat gRPC API, jenž by mezi ní a jejím uživatelem definovalo komunikační protokol obsahující datové struktury a metody pro vzdálenou obsluhu. Volba gRPC protokolu také otevírá možnosti implementace ovladačů i v jiných technologiích (než je Java).

Vytvořený ovladač by měl uživatele odstínit od gRPC vrstvy a umožnit mu pomocí dostupných Java rozhraní volat metody stejně jako u embedované varianty databáze. Jelikož gRPC nedokáže přímo pracovat s nativními strukturami z evitaDB API, používá pouze jejich reprezentace definované pomocí *proto* souborů. Aby bylo možné zachovat jednotné evitaDB Java API, musí být vytvořen obousměrný převaděč umožňující konverzi mezi oběma zmíněnými typy struktur. S jeho použitím je pak možné v ovladači pracovat včetně používání metod nacházejících se na třídách a rozhraních z evitaDB implementace.

Několikrát již v této práci zazněl pojem evitaDB Java API, avšak bez upřesnění v něm obsažených a nabízených struktur a funkcí. Toto API představuje samostatný Java modul (JAR) skládající se z databází podporovaných datových typů, rozhraní a jimi realizovaných struktur používaných napříč celou implementací. Pro samotné ovládání a manipulaci s databází slouží dvojice níže popsaných rozhraní.

První z nich nese název *EvitaContract*, jenž obsahuje metody pro správu a používání evitaDB na úrovni katalogů (vytváření či úpravy) nebo vytváření a terminaci instancí databázové relace (*EvitaSessionContract*) s možností stanovit její charakteristiky (např. relace je určena pouze pro čtení). Celý výčet dostupných metod se nachází na obrázku níže.



Obrázek 6 - Seznam metod předepsaných rozhraním EvitaContract. Zdroj: [autor]

Na výše zmíněném rozhraní *EvitaSessionContract* lze provádět konkrétní operace na úrovni kolekcí entit. Jmenovitě se jedná například o úpravy entitních schémat, tvorbu a editaci uložených entit a vyhledávání jednotlivých entit specifikováním jejich identifikátorů nebo použitím datové struktury typu *Query* představující strukturovaný dotaz na databázi. Pro provádění změn (mutací) je nezbytné, aby v databázové relaci vznikla tzv. transakce zajišťující izolaci změn vůči ostatním paralelním klientům a zároveň umožňovala detekovat konflikty ve chvíli, kdy je seznam mutací potvrzen tzv. "commit" operací. Níže je uveden seznam předpisů vybraných metod (z důvodu jejich velkého množství) z rozhraní *EvitaSessionContract* s vysvětlením jejich účelu:

```
void close();
```

Metoda předepsaná rozhraním *AutoClosable* způsobí ukončení relace a uvolnění jí používaných zdrojů.

```
Optional<SealedEntitySchema> getEntitySchema(@Nonnull String entityType);
```

Metoda navracející schéma odpovídající předanému typu entity obalené v kontejneru Optional, jenž předchází nutnosti navracet hodnotu *null* (v takovém případě je prázdný).

```
<S extends EntityClassifier, T extends EvitaResponse<S>> T query(@Nonnull
Query query, @Nonnull Class<S> expectedType);
```

Metoda přijímající objekt představující požadovaný dotaz a genericky uvedenou třídu, jež specifikuje typ obdržené odpovědi.

```
Optional<SealedEntity> getEntity(@Nonnull String entityType, int
primaryKey, EntityContentRequire... require);
```

Pokud existuje, pomocí zadaných identifikátorů vrátí entitu se specifikovanou bohatostí.

```
EntitySchemaBuilder defineEntitySchema(@Nonnull String entityType);
```

Definuje parametricky pojmenované schéma, pro které je navrácen *EntitySchemaBuilder*.

```
SealedCatalogSchema updateAndFetchCatalogSchema(@Nonnull
LocalCatalogSchemaMutation... schemaMutation);
```

Přijímá libovolný počet mutací pro úpravu již existujícího schématu katalogu a vrací podobu aktualizovaného schématu po jejich aplikaci.

```
default int updateAndFetchEntitySchema(@Nonnull EntitySchemaBuilder
entitySchemaBuilder);
```

Metoda přijímající v parametru instanci typu *EntitySchemaBuilder*, která obsahuje požadované změny k provedení, navrácena je verze nového schématu.

```
boolean deleteCollection(@Nonnull String entityType);
```

Smaže parametricky specifikovanou kolekci entit, návratová hodnota informuje o tom, jestli byla operace úspěšná.

```
EntityBuilder createNewEntity(@Nonnull String entityType);
```

Metoda, jenž pro v zadané kolekci vytvoří novou entitu, navrací jí odpovídající *EntityBuilder*.

```
SealedEntity upsertAndFetchEntity(@Nonnull EntityMutation entityMutation,
EntityContentRequire... require);
```

Slouží k aplikování předané mutace entity a navrácení aktualizované entity s odpovídající bohatostí.

```
boolean deleteEntity(@Nonnull String entityType, int primaryKey);
```

Smaže identifikátory specifikovanou entitu z databáze.

```
long openTransaction();
```

Slouží k otevření transakce, přičemž návratovou hodnotu představuje její identifikátor.

```
void closeTransaction();
```

Zavře aktuálně otevřenou transakci.

Z hlediska implementace gRPC API je cílem pomocí Protocol bufferů co nejpřesněji reprezentovat všechny potřebné struktury a služby pro obsluhu evitaDB databáze, které

pocházejí z výše popsaných rozhraní *EvitaContract* a *EvitaSessionContract*. Pro ověření správného fungování musí být realizované API pokryto funkcionálními i jednotkovými testy. Po dokončení návrhu a po samotné realizaci serverové části gRPC API vznikne klientský ovladač napsaný v programovacím jazyku Java, jenž bude poskytovat identické metody a využívat stejné datové struktury jako původní rozhraní evitaDB. To bude vyžadovat implementaci převaděčů dat mezi Javou a gRPC. Na jeho základě vznikne v rámci praktické části této práce ovladač postavený na jiné technologii, konkrétně na platformě .NET.

5.2. Specifika gRPC protokolu v .NET prostředí

Programovací jazyky Java a C# (.NET platforma) si jsou v mnoha aspektech velmi podobné, což by mělo implementaci .NET ovladače relativně usnadnit. Četné rozdíly mezi nimi samozřejmě jsou (především na nižší úrovni), takže C# API nebude úplně přesně odpovídat původnímu evitaDB Java API. Pro některé z chybějících konceptů lze sice nainstalovat knihovnu, která by je poskytla, nicméně dílčím cílem při implementaci evitaDB ovladačů je klást důraz na minimalizaci závislostí a využívat nativních prostředků dané technologie. To programátorům používajícím vytvořený ovladač umožní soustředit se čistě na nabízené funkce bez nutnosti dohledávat si dokumentaci k dalším dodatečným knihovnám a jejich strukturám, jež by byly zavedeny čistě kvůli přiblížení se Java implementaci.

Ovladač v jazyku C# sice bude možné implementačně založit na Java ovladači, jenž bude realizován po dokončení gRPC API, nicméně pro zachování příjemného používání bude nezbytné adekvátně navrhnout a vytvořit všechny struktury odpovídající zvyklostem dané platformy, které jsou pro používání ovladače nezbytné. Mezi ně lze kategoricky zařadit všechny z níže popsaných struktur, jež jsou v Java implementaci segmentovány do pěti odlišných modulů:

- všechny databází podporované datové typy včetně výčtových typů,
- struktury reprezentující základní prvky z evitaDB, jako jsou katalogy, kolekce entit, entity s přidruženými daty a schémata,
- rozhraní a jejich konkrétní realizace představující různé typy mutací a builderů,
- zavedení tříd pro dotazování pomocí *Query* včetně dílčích použitelných constraint,
- vytvořit pro všechny používané typy, jenž klient může poslat nebo obdržet z dotazu na API, oboustranné převodníky mezi C# a gRPC message specifikacemi,
- modelové třídy pro reprezentaci odpovědí z API, dodatečné výsledky (*ExtraResults*).

Vzhledem k velkému počtu struktur skrývajícími se za výše uvedenými kategoriemi požadavků na celý ovladač nebudou v rámci této práce implementovány všechny. Výstupem by měl být prototyp ovladače (tzv. MVP – minimal viable product), jenž bude umožňovat správně, rychle a co nejefektivněji komunikovat s gRPC API. Pomocí vlastní implementace rozhraní *EvitaContract* a *EvitaSessionContract* by měl vytvořený ovladač nabízet co nejpodobnější způsob manipulace a práce s databází k existujícímu a stabilnímu Java API.

Navrhovaný prototyp bude obsahovat rámcově všechny zástupce výše uvedených kategorií, ale pro demonstraci funkcionalit nebudou vytvářeny všechny z podporovaných:

- podmínek v rámci Query,
- dodatečných výsledků (*ExtraResults*),
- typů Builderů pro existující struktury,
- mutací,
- metod definovaných výše zmíněnými rozhraními,
- typů NumberRange,
- komplexních datových typů, konkrétně *ComplexDataObject*.

Výsledný prototyp by měl obsahovat množinu funkcí, na jejichž základě lze dojít k závěru, že v této technologii bude možné implementovat celý ovladač se všemi dostupnými funkcemi. Do této množiny spadá možnost založit nové schéma katalogu a vytvořit v něm nové schéma pro kolekci entit. Na úrovni kolekce entity by pak mělo být možné pomocí mutací provést nějakou základní změnu, například přidání nových atributů do schématu. Při existenci schémat by mělo být možné vložit do kolekce novou entitu, jejíž obsah musí dodržovat schématem specifikované atributy. Pokus o vložení entity s obsahem porušujícím definované schéma musí skončit výjimkou. Jelikož ve výchozím režimu schémat kolekcí entit jsou povoleny všechny evoluční módy, mělo by být také možné vkládat entity s novými, doposud neexistujícími atributy a schémata by se na ně měla automaticky adaptovat. Vložené entity by pomocí ovladače mělo být možné načíst v požadovaném rozsahu a mít pro ně připravenou vhodnou datovou strukturu. Poslední podporovanou funkcí ovladače je použití *Query* struktury pro dotazování. S její pomocí bude možné reprezentovat i složitější dotaz, pro který by pak ovladač měl vracet správná data. Všechny z těchto vyjmenovaných funkcionalit musí být obsaženy ve snadno spustitelném testu ověřujícím správné fungování vytvořeného ovladače.

6. Realizace praktické části

V této kapitole budou uvedeny zajímavé poznatky a problémy, na něž se při vývoji praktické části narazilo a bylo je nutné vhodným způsobem vyřešit. První z podkapitol bude věnována návrhu a realizaci gRPC API, druhá pak na samotnou implementaci ovladače pro evitaDB databázi s využitím jazyka C#. V průběhu kapitoly budou uváděny, v některých případech zjednodušené a zkrácené (z důvodu dlouhých metod či definic), ukázky kódu sloužící k lepší demonstraci a pochopení řešené problematiky. Výsledné zdrojové kódy jsou k dispozici online ve dvou repozitářích na platformě GitHub, v prvním se nachází většina implementované logiky gRPC API[8], ve druhém je umístěn vytvořený prototyp ovladače[9]. Jelikož se jedná o velký firemní projekt, v průběhu i po dokončení realizace praktické části byly některé části API pozměňovány a dále rozvíjeny i jinými zaměstnanci firmy kvůli neustále vyvíjející se evitaDB databázi.

6.1. Implementace gRPC API

Obsah této podkapitoly je zaměřen na vysvětlení dílčích problematik, které bylo nutné v rámci realizace gRPC API řešit. Pro ověření správného fungování byly průběžně tvořeny funkcionální testy, které simulovaly dotazy na API z role klienta. Zároveň tím bylo možné vyzkoušet a následně uzpůsobit formu požadavků, aby měly co nejrozumnější strukturu z hlediska serverového zpracování, ale aby také z klientského pohledu dávaly smysl a nebyly zbytečně složitě realizované.

6.1.1. Základní struktury používané v gRPC knihovně a její zprovoznění

Před samým začátkem je vhodné představit pojmy, na které lze narazit v materiálech soustředících se na gRPC nebo při jeho samotném vývoji. Mezi tyto pojmy patří tzn. kanál (Channel), jehož otevřením vznikne TCP spojení ke specifikovanému API, po němž proudí data mezi klientem a serverem. Pro umožnění komunikace je nutné nad již existující instancí kanálu vytvořit abstrakci zpřístupňující programátorovi protokolem definované metody nacházející se na serveru. Tato abstrakce je v některých programovacích jazycích označována pojmem Stub,

[8] gRPC API - https://github.com/FgForrest/evitaDB/tree/dev/evita_external_api/evita_external_api_grpc - vytvořené funkcionální a jednotkové testy se jsou umístěny v odlišném modulu, přístupné jsou na URL adrese https://github.com/FgForrest/evitaDB/tree/dev/evita_functional_tests/src/test/java/io/evitadb/externalApi/grpc

[9] Ovladač - https://github.com/FgForrest/evitaDB-C-Sharp-client

v jiných bývá také pojmenována přímo jako klient fungující nad konkrétní službou definovanou v souborech typu ProtocolBuffers. V poslední řadě bude zmíněn tzv. Interceptor, jenž může být implementován jak na klientské, tak i serverové straně komunikace. S jeho pomocí lze specifikovat logiku, jež bude provedena před samotným zpracováním zaslaného, resp. přijatého, dotazu.

Zprovoznění gRPC v projektu evitaDB, která používá programovací jazyk Java, bylo obtížnější především kvůli konfiguraci potřebného Maven pluginu. Ten obsahuje Java verzi *protoc* kompilátoru, jenž slouží ke kompilování *proto* souborů do nativních struktur. Vyžadované bylo, aby proto soubory nebyly umístěny ve složce se zdrojovými kódy a aby se do ní kompilovaly z nich vycházející Java soubory. V základním nastavení jsou tato umístění kategoricky obráceně (pouze typově, nejedná se o konkrétní složky), což konfiguraci značně ztížilo. Stejně tak bylo požadované zamezení čištění složky, do níž byly kompilovány Java soubory. Tento proces, především při větším počtu definovaných zpráv a služeb v *proto* souborech, totiž značně prodloužil potřebný čas ke kompilaci celého projektu.

6.1.2. Certifikáty

V kapitole 4 byly rozehrány koncepty, na nichž bylo gRPC navrženo, mezi což patřilo využívání pro komunikaci protokolu HTTP/2, jenž implicitně vyžaduje TLS zabezpečení pomocí certifikátu. Stejné zásady tedy platí i pro gRPC, a i když je možné toto API nakonfigurovat a provozovat i bez konfigurace TLS, použití tohoto způsobu v produkčním prostředí je silně nedoporučené, jelikož je tento způsob určen výhradně pro testování.

Na základě této skutečnosti byl při návrhu API provoz bez TLS znemožněn. Pro čistší způsob implementace a umožnění snadného spuštění bylo přidáno nastavení, pomocí kterého při inicializaci server vygeneruje dvojice testovacích certifikátů a jejich privátních klíčů. Nejdříve je vygenerována výše uvedená dvojice souborů pro kořenovou certifikační autoritu, níže jsou podepsány ostatní generované certifikáty určené pro server i klienta. Serverový certifikát je určený pro běžný provoz, klientský může být použitý při zapnutém režimu vzájemného ověřování mezi klientem a serverem (mTLS). Tyto potřebné soubory (kromě klíčů serveru a certifikační autority) by měl mít klient k dispozici, aby mohl s API komunikovat, byly tedy vystaveny na separátním koncovém bodě pomocí HTTP serveru Undertow, na němž jsou provozována ostatní API. [63]

Nastavení pro certifikáty byla zakomponována do konfiguračního souboru evitaDB databáze. Lze v něm nastavit, jestli mají být použité serverem generované certifikáty, nebo

soubory nacházející se v lokaci specifikované v dalších částech konfigurace. Dále pak může být povoleno vzájemné ověřování společně s možností uvést seznam certifikačních autorit, kterým bude, včetně obecně důvěryhodných autorit, dovoleno s API komunikovat. Z pohledu klienta (v rámci testů) bylo nutné vyřešit, jak jej přimět akceptovat certifikát vydaný nedůvěryhodnou certifikační autoritou, jímž se server může prezentovat. V Javě jsou certifikáty řešeny poměrně složitě využitím JKS (Java KeyStore), který uchovává informace o důvěryhodných certifikátech a autoritách odstíněně od certifikátů, jímž důvěřuje operační systém. Java implicitně důvěřuje certifikátům obsaženým jak v JKS, tak i importovaných do operačního systému. Důvod použití JKS bylo automatizovat proces použití nedůvěryhodných certifikátů bez nutnosti zasahovat do operačního systému nebo manuálně vkládat certifikáty do JKS. Nebylo však jednoduché JKS přimět k navázání komunikace s nedůvěryhodným serverem, pro tyto účely vznikla třída, do níž byla delegována logika pro umožnění komunikace s nedůvěryhodnými servery. [63]

Kvůli správnému fungování certifikátů bylo nutné použít dodatečnou knihovnu zprostředkující přístup k nastavením obsažených v Netty knihovně, které obsahují dodatečnou a redundantní logiku pro vytváření gRPC struktur. Ty je nutné využívat jak pro založení serveru, tak i správné nakonfigurování komunikačního kanálu na straně klienta.

6.1.3. Práce s evitaDB relacemi

Pro většinu operací je v evitaDB nezbytné mít k dispozici instanci implementující rozhraní *EvitaSessionContract*, která představuje databázovou relaci. Její koncipované použití bylo původně koncipováno tak, že jedna relace odpovídá jednomu dotazu na databázi, kvůli tomu i sama implementuje rozhraní *AutoClosable* napomáhající jejímu automatickému uzavření, když je specifikována v bloku "try-with-resources". Tímto způsobem ovšem přes API komunikovat nelze, skutečná instance relace na databázi existuje pouze na serveru a klient k ní tedy nemá přímý přístup, pro tyto účely mohou v databázi při použití gRPC API existovat i dlouze otevřené a znovupoužitelné relace.

Pro realizace databázových volání, která potřebují aktivní relaci, bylo využito gRPC interceptorů, jejichž logika je vykonána před provedením požadované metody. Klient před uskutečněním volání metod vyžadující aktivní relaci musí nejdříve zavolat metodu k vytvoření nové relace nad specifikovaným katalogem, přičemž mu jako odpověď přijde identifikátor jeho nové relace. S využitím implementace ClientInterceptor třídy je možné realizovat vlastní

chování, které bude uskutečněné před samotným požadavkem na server, ve kterém klient musí vložit do metadat název katalogu a identifikátoru relace.

Na serveru jsou vždy získány informace z metadat klientského volání, v nichž by měl být obsažen název katalogu a identifikátor jím používané relace. Pokud specifikovaná relace v databázi neexistuje nebo není aktivní a zároveň je pro realizaci požadavku nezbytná (nachází se na *EvitaSessionContract*), požadavek nebude úspěšný a klientovi bude navrácena chyba. Jestliže relace se specifikovaným identifikátorem existuje, její instance bude z databáze získána a nastavena do speciálního kontextu asociovaného s daným volání serverové metody. V rámci následovného provedení implementované metody na serveru je z tohoto kontextu relace získána a na ní provedena požadovaná metoda. Realizace serverového interceptoru je na ukázce níže.

```
@Override
public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(ServerCall<ReqT,
RespT> serverCall, Metadata metadata, ServerCallHandler<ReqT, RespT>
serverCallHandler) {
	final Metadata.Key<String> catalogNameMetadata =
	Metadata.Key.of(CATALOG_NAME_HEADER, Metadata.ASCII_STRING_MARSHALLER);
	final Metadata.Key<String> sessionMetadata =
	Metadata.Key.of(SESSION_ID_HEADER, Metadata.ASCII_STRING_MARSHALLER);
	final String catalogName = metadata.get(catalogNameMetadata);
	final String sessionId = metadata.get(sessionMetadata);
	final Optional<EvitaInternalSessionContract> activeSession =
	resolveActiveSession(catalogName, sessionId);
	if (activeSession.isEmpty() && isEndpointRequiresSession(serverCall)) {
		final Status status = Status.UNAUTHENTICATED
		.withCause(new EvitaInvalidUsageException("Your session is either
		not set or is not active."))
		.withDescription("Your session (catalog: "+ catalogName + ",
		session id: " + sessionId + ") is either not set or is not
		active.");
		serverCall.close(status, metadata);
		return new ServerCall.Listener<>() {};
	}
	if (activeSession.isEmpty()) {
		return serverCallHandler.startCall(serverCall, metadata);
	}
	final Context context = Context.current().withValue(SESSION,
	activeSession.get());
	return Contexts.interceptCall(context, serverCall, metadata,
	serverCallHandler);
}
```

Ukázka kódu 9 - Ukázka části implementace třídy ServerSessionInterceptor. Zdroj: [autor]

6.1.4. Návrh podporovaných datových typů

Při návrhu struktury definující databází podporované datové typy bylo nutné najít optimální řešení, které by bylo snadno použitelné z klientské strany s co nejmenším počtem redundantních definic potřebných typů. V evitaDB je interně uchovávaný seznam podporovaných datových typů, proti němuž se kontroluje validita poskytnutých hodnot. Podobným způsobem byl navržen i gRPC protokol, viz ukázka kódu níže. K realizaci bylo použito klíčové slovo *oneof*

s výčtem všech akceptovatelných typů, přičemž může být nastaven pouze jeden. Pro tento případ bylo ale nutné vytvořit dodatečné typy pro obalující pole hodnot pro každý podporovaný typ, jelikož seznamy a mapy nejsou v kombinaci s *oneof* aplikovatelné. Typ reprezentující databází podporované hodnoty může obsahovat i verzi hodnoty a také její typ, který je realizován výčtovým typem všech podporovaných typů hodnot. Jeho nastavená hodnota je používána při práci se schématy, například při specifikování typu atributu.

```
message GrpcEvitaValue {
	oneof value {
		string stringValue = 1;
		int32 integerValue = 2;
		int64 longValue = 3;
		bool booleanValue = 4;
		GrpcBigDecimal bigDecimalValue = 5;
		GrpcDateTimeRange dateTimeRangeValue = 6;
		GrpcIntegerNumberRange integerNumberRangeValue = 7;
		GrpcLongNumberRange longNumberRangeValue = 8;
		GrpcBigDecimalNumberRange bigDecimalNumberRangeValue = 9;
		GrpcOffsetDateTime offsetDateTimeValue = 10;
		GrpcLocale localeValue = 11;
		GrpcCurrency currencyValue = 12;

		GrpcStringArray stringArrayValue = 13;
		GrpcIntegerArray integerArrayValue = 14;
		GrpcLongArray longArrayValue = 15;
		GrpcBooleanArray booleanArrayValue = 16;
		GrpcBigDecimalArray bigDecimalArrayValue = 17;
		GrpcDateTimeRangeArray dateTimeRangeArrayValue = 18;
		GrpcIntegerNumberRangeArray integerNumberRangeArrayValue = 19;
		GrpcLongNumberRangeArray longNumberRangeArrayValue = 20;
		GrpcBigDecimalNumberRangeArray  bigDecimalNumberRangeArrayValue  =
		21;
		GrpcOffsetDateTimeArray offsetDateTimeArrayValue = 22;
		GrpcLocaleArray localeArrayValue = 23;
		GrpcCurrencyArray currencyArrayValue = 24;
	}
	GrpcEvitaDataType type = 30;
	google.protobuf.Int32Value version = 31;
}
```

Ukázka kódu 10 - Struktura sloužící ke specifikaci hodnot všech jednoduchých datových typů podporovaných evitaDB databází. Zdroj: [autor]

Na ukázce níže je struktura uchovávající hodnoty asociovaných dat, která kromě výše uvedených typů podporují i ComplexDataObject. Ten je na serverové straně serializován do JSON formátu, v němž může být poslán jako běžná textová hodnota. Reálně použitá hodnota je zde obalena pomocí dalšího oneof výčtu, jenž přijímá buď výše uvedený objekt obsahující jednoduchý a podporovaný datový typ, nebo zmíněnou textovou reprezentaci JSON objektu.

```
message GrpcEvitaAssociatedDataValue {

	oneof value {
		GrpcEvitaValue primitiveValue = 1;
		string jsonValue = 2;
	}
	google.protobuf.Int32Value version = 3;
}
```

Ukázka kódu 11 - Zpráva definující podobu struktury pro reprezentaci hodnoty asociovaných dat. Zdroj: [autor]

6.1.5. Formát požadavků a odpovědí obsahující dotaz na databázi

Pro provedení složitých dotazů na evitaDB je potřeba poslat objekt typu Query. Databáze již disponovala parserem, jenž umožňoval převádět tyto objekty z jejich textové reprezentace do požadovaných objektů. Kvůli nedostatku lepších alternativ byl tento způsob zvolen pro gRPC, klient by tedy posílal na server dotazy v textové podobě a server by je s použitím parseru dokázal zpracovat a použít. Z bezpečnostních důvodů (zamezení QL injection[10]) bylo ovšem potřebné rozšířit možnosti parseru o parametrizaci, tedy aby byly namísto hodnot používány zástupné znaky, na jejichž místo by parser na serveru dosadil správné hodnoty. Byla přidána možnost parametry předávat hned ve dvou variantách: v obecném seznamu a v mapě, v níž byl klíčem název parametru specifikovaný klientem. Pro pokrytí ostatních podporovaných typů v dotazu byl zaveden na úrovni gRPC protokolu i nový typ, jenž kromě jednoduchých datových typů podporovaných v databázi umožňoval i specifikaci pěti výčtových typů, například sloužící změně typu ceny (S/BEZ DPH), s níž jsou prováděny výpočty. Níže je uveden příklad navržené struktury v komunikačním protokolu.

```
message GrpcQueryRequest {
	string query = 1;
	repeated QueryParam positionalQueryParams = 2;
	map<string, QueryParam> namedQueryParams = 3;
}
```

Ukázka kódu 12 - Klientský požadavek na vykonání query metody v ProtocolBuffers. Zdroj: [autor]

Odpověď na rozebíraný požadavek je složena z dvojice parametrů reflektujících typ *DataChunk* a množinu dodatečně vypočtených výpočtů (*ExtraResults*). Byly pro ně navrženy optimální struktury s důrazem kladeným na jejich příjemné používání, nabídnutí více způsobů pro získání požadovaných hodnot z konkrétních dodatečných výpočtů a celkové usnadnění převodů mezi gRPC a používaným programovacím jazykem. Návrh často komplikovala omezení ze strany používaného protokolu, jenž například nedovoloval použít definovaný typ jako klíč ve struktuře mapy; je možné použít pouze primitivní datové typy. Na ukázce níže lze vidět reprezentace obou navržených struktur.

[10] QL injection – jeden ze základních útoků na databáze, který cílí na modifikaci zasílaného dotazu

```
message GrpcDataChunk {
	repeated GrpcEntityReference entityReferences = 1;
	repeated GrpcSealedEntity sealedEntities = 2;
	repeated GrpcBinaryEntity binaryEntities = 3;
	oneof chunk {
		GrpcPaginatedList paginatedList = 4;
		GrpcStripList stripList = 5;
	}
	int32 totalRecordCount = 6;
	bool isFirst = 7;
	bool isLast = 8;
	bool hasPrevious = 9;
	bool hasNext = 10;
	bool isSinglePage = 11;
	bool isEmpty = 12;
}

message GrpcExtraResults {
	repeated GrpcHistogram attributeHistograms = 1;
	map<string, GrpcHistogram> attributeHistogram = 2;
	GrpcHistogram priceHistogram = 3;
	repeated GrpcFacetGroupStatistics facetGroupStatistics = 4;
	GrpcHierarchyParentsByReference selfHierarchyParents = 5;
	map<string, GrpcHierarchyParentsByReference> hierarchyParents = 6;
	GrpcLevelInfos selfHierarchyStatistics = 7;
	map<string, GrpcLevelInfos> hierarchyStatistics = 8;
	GrpcQueryTelemetry queryTelemetry = 9;
}
```

Ukázka kódu 13 - Struktury formující odpověď k zaslání klientovi při použití query metody reprezentované pomocí ProtocolBuffers. Zdroj: [autor]

6.1.6. Reprezentace mutací

Mutace jsou v evitaDB velmi závislé na dědičnosti, kterou pomocí ProtocolBuffers nelze modelovat. Jak již bylo v kapitole 3.6 vysvětleno, mutací je v této databázi nabízeno mnoho typů, přičemž každý může pokrývat velké množství konkrétních mutací. Řešením bylo zapomenout na jejich společné rysy a zaměřit se pouze na modelování jednotlivých kategorií. Ty pak obsahují výčty do ní spadajících mutací uvedené v *oneof* bloku, jenž dovolí specifikovat pouze jednu. Na ukázkách níže jsou dva příklady vytvořených typů pro dva typy mutací schémat katalogu. Pro snazší vizualizaci je u prvního typu uvedena pouze část podporovaných mutací.

```
message GrpcLocalCatalogSchemaMutation {
   oneof mutation {
       GrpcModifyCatalogSchemaDescriptionMutation
       modifyCatalogSchemaDescriptionMutation = 1;
       GrpcCreateGlobalAttributeSchemaMutation
       createGlobalAttributeSchemaMutation = 2;
       .....
       GrpcModifyEntitySchemaNameMutation modifyEntitySchemaNameMutation = 17;
       GrpcRemoveEntitySchemaMutation removeEntitySchemaMutation = 18;
   }
}

message GrpcTopLevelCatalogSchemaMutation {
   oneof mutation {
```

```
        GrpcCreateCatalogSchemaMutation createCatalogSchemaMutation = 1;
        GrpcModifyCatalogSchemaNameMutation modifyCatalogSchemaNameMutation =
        2;
        GrpcRemoveCatalogSchemaMutation removeCatalogSchemaMutation = 3;
    }
}
```

Ukázka kódu 14 - Reprezantace různých typů mutací schématu katalogu pomocí ProtocolBuffers specifikace. Zdroj: [autor]

6.1.7. Implementace protokolem definovaných služeb

Na ukázce níže je rozebráno zjednodušené zpracování metody query na straně API (některé části byly pro kompaktnost vynechány). Server obdrží od klienta parametrizované query, viz 6.1.5, které je pomocí parseru převedeno do objektové struktury. Pokud se objekt podařilo sestavit, dotaz zaslaný klientem dával smysl, v opačném případě by došlo k ukončení zpracování požadavku. Následně je z dotazu vytvořena interní reprezentace požadavku, který je použitý k vykonání dotazu databází. Poté dojde k vytvoření gRPC Builderu z obdržené odpovědi, do něhož jsou nastaveny informace o počtu nalezených dat a vrácených dat, načež dojde ke konverzi dodatečně vypočtených výsledků a jejich zakomponování do Builderu. Podle specifikace bohatosti entit následuje výběr potřebného konvertoru, jenž bude použit pro převod reprezentací entit z evitaDB databáze na objekty specifikované v gRPC protokolu, načež dojde k sestavení Builderem nastaveného objektu, který je poslán klientovi.

```
@Override
public void query(@Nonnull GrpcQueryRequest request, @Nonnull
StreamObserver<GrpcQueryResponse> responseObserver) {
	final EvitaInternalSessionContract session =
	ServerSessionInterceptor.SESSION.get();

	final Query query = QueryUtil.parseQuery(
	request.getQuery(),
	request.getPositionalQueryParamsList(),
	request.getNamedQueryParamsMap(),
	responseObserver
	);

	if (query != null) {
	final EvitaRequest evitaRequest = new EvitaRequest(
		query.normalizeQuery(),
		OffsetDateTime.now()
	);

	final EvitaResponse<EntityClassifier> evitaResponse =
	session.query(evitaRequest, EntityClassifier.class);
	final GrpcQueryResponse.Builder entityBuilder =
	GrpcQueryResponse.newBuilder();
	final DataChunk<EntityClassifier> recordPage =
	evitaResponse.getRecordPage();
	final GrpcDataChunk.Builder dataChunkBuilder = GrpcDataChunk.newBuilder()
	.setTotalRecordCount(evitaResponse.getTotalRecordCount())
	.....

	if (recordPage instanceof PaginatedList<?> paginatedList) {
		dataChunkBuilder.getPaginatedListBuilder()
```

```
                .setPageNumber(paginatedList.getPageNumber())
                .setPageSize(paginatedList.getPageSize());
        }
        .....

        entityBuilder.setExtraResults(
                GrpcExtraResultsBuilder.buildExtraResults(evitaResponse)
        );

        final EntityFetch entityRequirement =
        evitaRequest.getEntityRequirement();
        if (entityRequirement != null) {
                .....
                final List<GrpcSealedEntity> sealedEntities = new
                ArrayList<>(recordPage.getData().size());
                recordPage.stream().forEach(e ->
                        sealedEntities.add(
                        EntityConverter.toGrpcSealedEntity((SealedEntity) e))
                );
                entityBuilder.setRecordPage(dataChunkBuilder
                        .addAllSealedEntities(sealedEntities)
                        .build()
                );
                .....
        }
        .....
        responseObserver.onNext(entityBuilder.build());
        }
        responseObserver.onCompleted();
}
```

Ukázka kódu 15 - Implementace query metody na straně gRPC API. Zdroj: [autor]

Na ukázce níže se nachází metoda, která jako parametr přijímá definici logiky obsahující změny v datech. Takové operace musí být vždy součástí transakce, kterou klient může sám otevřít. Metoda nejprve zkusí provést logiku pomocí metody *execute*, která implicitně předanou logiku obalí novou transakci. Pokud její provedení skončí chybou, znamená to, že klient již měl transakci otevřenou a stačí pouze provést požadované změny na předané relaci.

```
private void handleTransactionCall(@Nonnull EvitaSessionContract session,
@Nonnull Consumer<EvitaSessionContract> logic) {
        try {
                session.execute(logic);
        } catch (UnexpectedTransactionStateException ex) {
                logic.accept(session);
        }
}
```

Ukázka kódu 16 - Metoda k zajištění vykonání předané logiky v rámci transakce. Zdroj: [autor]

6.2. Tvorba C# klienta

Tato podkapitola bude zaměřena na problémy související s vývojem ovladačů pro evitaDB komunikujících přes gRPC API, používání gRPC z programovacího jazyka C# a poukázání na nalezené odlišnosti práce s tímto frameworkem. Cílem je vytvoření prototypu ovladače dle specifikací vyjmenovaných na konci 5. kapitoly. Vývoj ovladače bude vycházet z již existujícího Java ovladače a struktur obsažených v základní implementaci evitaDB.

6.2.1. Nastavení Protocol buffers a porovnání práce s gRPC strukturami

Instalace a zprovoznění gRPC bylo z jazyku C#, v porovnání s Javou, značně jednodušší. Pro bezproblémovou práci s gRPC včetně umožnění kompilace *proto* souborů bylo potřeba nainstalovat 3 knihovny: oficiální knihovnu od Googlu pro podporu ProtocolBuffer protokolu, gRPC nástroje obsahující *protoc* kompilátor k získání nativních struktur a v poslední řadě samotnou gRPC .NET knihovnu. Po jejich instalaci stačilo do souboru obsahující nastavení projektu přidat kód na ukázce níže a při spuštění projektu již byly kompilovány *proto* soubory umístěné v nastavené složce.

```
<ItemGroup>
<Protobuf ProtoRoot="Protos" Include="Protos\*.proto" GrpcServices="Client" />
</ItemGroup>
```

Ukázka kódu 17 - Konfigurace gRPC v C# projektu. Zdroj: [autor]

Pro .NET platformu je pouze jedna aktivně vyvíjená a udržovaná knihovna, která se těší velmi rychlé adaptaci na nové standardy, jako například HTTP/3. V porovnání s Java knihovnou má lepší integraci na nativní možnosti jazyka, mezi něž patří například přímé mapování obalujících typů z ProtocolBuffer jako *StringValue* nebo *Int32Value* na jim odpovídající datové typy, které mohou obsahovat i *null* hodnotu (reflektovat neposkytnutí hodnoty v gRPC do zmíněných typů). Jako druhý příklad lze uvést vytváření objektů definovaných na úrovni protokolu, tedy zkompilované výstupy *proto* souborů. Ty lze klasicky vytvářet s požadovanými vlastnostmi s využitím inicializačních závorek namísto konstruktoru. V Javě je každý protokolem definovaný objekt nutné postupně nastavovat pomocí Builderů specifických pro daný typ, což například při větším množství zanořených objektů může být dosti nepřehledné.

6.2.2. Certifikáty

Ve srovnání s výše uvedeným popisem řešení certifikátů v Javě je možnost vlastní validace certifikátů z jazyka C# velmi jednoduchá. Jelikož je v něm pro vytvoření komunikačního kanálu nezbytné poskytnout běžně používaný *HttpClient* disponující velkou škálou nastavení, bylo možné snadno definovat vlastní logiku pro vyhodnocení důvěryhodnosti certifikátu, jímž se server prezentuje. Tento proces se odehrává pouze pokud je v nastavení ovladače specifikováno, že není používán certifikát vydaný důvěryhodnou certifikační autoritou (například Let's Encrypt). Na ukázce níže je implementována ověřovací logika, v níž server porovná skutečně používaný serverový certifikát se souborem, který stáhl ze serverem vystaveného koncového bodu. Pro ověření shody certifikátů postačí porovnání jejich

Thumbprintů, které představují jejich unikátní, na obsahu závislý textový identifikátor. Pokud se shodují, jedná se o stejný certifikát a komunikace se serverem bude být bezpečně navázána.

```
public HttpClientHandler BuildHttpClientHandler()
{
	var handler = new HttpClientHandler();
	if (!TrustedServerCertificate)
	{
		handler.ServerCertificateCustomValidationCallback =
		RemoteCertificateValidationCallback;
	}
	return handler;
}

private bool RemoteCertificateValidationCallback(
HttpRequestMessage message, X509Certificate2? cert, X509Chain? chain,
SslPolicyErrors errors)
{
	var usedCert =
	new X509Certificate2(
	File.ReadAllBytes($"{ClientCertificateFolderPath}/{CertificateUtils.
	GeneratedCertificateFileName}"));
	if (cert == null)
	return false;
	return cert.Thumbprint == usedCert.Thumbprint;
}
```

Ukázka kódu 18 - Ukázka definice vlastní logiky pro akceptování serverového certifikátu. Zdroj: [autor]

6.2.3. Problémy s generikou

Při reimplementaci struktur obsažených v evitaDB v jazyku C# často nastávaly problémy s generikou. Databáze byla na generice ve velké míře postavena a pro rozšíření možností a zjednodušení vývoje se namísto konkrétních generických parametrů hojně využívá zástupný znak v podobě otazníku. Do ním definované generiky lze poslat jakoukoliv třídu, avšak bez znalosti jejího reálného typu mimo spuštěnou aplikaci. Stejně jako běžné generické parametry lze i u otazníku pomocí klíčových slov *extends* a *super* v kombinaci s uvedením typu omezit, jaké typy mohou být akceptovány. V jazyku C# jsou kritéria a omezení na generiku přísnější a obdoba pro tyto zástupné znaky v něm neexistuje. Kvůli tomu bylo nutné některé implementované části buď zjednodušit, řešit kompletně odlišným přístupem, nebo je vynechat.

Jako příklad zmíněného, ne snadno řešitelného, problému je možné uvést chybějící vestavěnou podporu pro zjištění, jestli generický objekt je instancí typu, jenž je jeho nadřazenou třídou, přičemž stejný vztah platí i mezi jejich genericky specifikovanými parametry. Vestavěná metoda sloužící ke zjišťování, jestli je jeden typ instancí druhého, je zabudovaná v jazyku C# a řeší dědičnost pouze u samotných typů. Do vyhodnocení nebere v potaz generické parametry, a jelikož nejsou stejné (i přes existující vztah dědičnosti), nemůže ke shodě nikdy dojít. Tento problém byl vyřešen pomocí dodefinovaní potřebné funkcionality pro třídu *Type* s využitím

„Extension metody“, jež dovoluje přidat vlastní metody na existující třídy i bez přístupu k jejich implementaci. Zmíněné metody jsou uvedeny na ukázce níže.

```
public static class AssignableExtensions
{
	public static bool IsAssignableToGenericType(this Type? givenType, Type?
genericType) {
	if (givenType == null || genericType == null) {
		return false;
	}

	return givenType == genericType
	|| givenType.MapsToGenericTypeDefinition(genericType)
	|| givenType.HasInterfaceThatMapsToGenericTypeDefinition(genericType)
	|| givenType.BaseType.IsAssignableToGenericType(genericType);
}

	private static bool HasInterfaceThatMapsToGenericTypeDefinition(this Type
	givenType, Type genericType)
	{
		return givenType
		.GetInterfaces()
		.Where(it => it.IsGenericType)
		.Any(it => it.GetGenericTypeDefinition() == genericType);
	}

	private static bool MapsToGenericTypeDefinition(this Type givenType, Type
genericType)
	{
		return genericType.IsGenericTypeDefinition
		&& givenType.IsGenericType
		&& givenType.GetGenericTypeDefinition() == genericType;
	}
}
```

Ukázka kódu 19 - Metody sloužící k požadovanému chování při ověřování dědičnosti u objektů s generickými parametry. Zdroj: [64]

Pro konkrétní demonstraci zmíněného problému jsou níže uvedeny dva kusy kódu, které se nacházejí v třídě *PrettyPrintingVisitor*, jež obsahuje logiku z návrhového vzoru Visitor. Ten je v rámci *Query* využíván pro procházení stromu skládajícího se ze specifikovaných constraint a jejich postupnou aplikaci. Právě kvůli této metodě bylo nezbytné sestavit výše vysvětlené metody, jelikož definice volané metody *PrintContainer* vyžadovala jako generický parametr specifikovaný obecným rozhraním *ICoinstraint*, ale aplikované constrainty byly genericky určeny typy jeho potomků. Co z ukázky nemusí být na první pohled zřejmé, *IConstraintContainer* je pouhé rozhraní, jenž je realizováno větším počtem od sebe dědících tříd a rozhraní, od kterých objekty (v ukázce vystupující v proměnné constraint) skutečně dědí.

```
if
(constraint.GetType().IsAssignableToGenericType(typeof(ConstraintContainer<>)))
{
	switch (constraint)
	{
	case IConstraintContainer<IFilterConstraint> filterContainer:
		PrintContainer(filterContainer);
		break;
```

```
        case IConstraintContainer<IOrderConstraint> orderContainer:
              PrintContainer(orderContainer);
              break;
        case IConstraintContainer<IRequireConstraint> requireContainer:
              PrintContainer(requireContainer);
              break;
        default:
              throw new NotSupportedException();
        }
}
```

Ukázka kódu 20 - Část metody pro průchod a aplikaci query constraint. Zdroj: [autor]

```
private void PrintContainer(IConstraintContainer<IConstraint> constraint)
```

Ukázka kódu 21 - Hlavička metody zdůrazňující typ generického parametru. Zdroj: [autor]

6.2.4. Ukládání schémat do mezipaměti

V evitaDB implementaci vystupuje několik struktur definovaných schématem, mezi ty stěžejní patří katalog a kolekce entit. Pro optimalizaci velikosti přenášených dat, u kterých nedochází k častým změnám, bylo možné na straně ovladače vytvořit mezipamět udržující aktuální verzi schémat, aby je nebylo nutné posílat společně s každou entitou. Při převodu dat z gRPC struktur do tříd kopírujících evitaDB implementaci tak postačuje z mezipaměti vytáhnout schéma verze, která je jako jediná z informací schématu definována přímo na entitě, resp. katalogu. Pokud požadované schéma v mezipaměti není, je automaticky proveden další dotaz, jenž zajistí jeho stažení, překonvertování a uložení do mezipaměti, z níž jej bude možné použít při dalších dotazech.

6.2.5. Channel pooling

Pro zvýšení efektivity a výkonu implementovaných ovladačů byl navržen tzv. channel pooling. Ten je reprezentován třídou obsahující kolekci představující frontu, do níž je při inicializaci ovladače vložen zadaný počet komunikačních kanálů, jejichž vytváření (kvůli navazování TCP spojení) je drahé a vyplatí se již vytvořené kanály sdílet a opětovně používat. Při provádění metod dostupných na ovladači je samotné volání API obaleno níže uvedenou metodou, která akceptuje delegát obsahující logiku k provedení na abstrakci vytvořené nad gRPC stubem (klientem). V rámci metody je z vytvořeného ChannelPoolu vyžádán kanál (pokud není žádný k dispozici, je navrácena jeho nově inicializovaná instance), na němž bude vytvořena instance zmíněného klienta, jenž provede požadovanou logiku. Po dokončení je kanál vrácen zpět do poolu (na konec fronty). Tento přístup vedl k výraznému zlepšení výkonu klienta především při provádění paralelních dotazů na databázi.

```
private T ExecuteWithEvitaSessionService<T>(
            Func<EvitaSessionService.EvitaSessionServiceClient, T>
      evitaSessionServiceClient)
{
      var channel = _channelPool.GetChannel();
      try
      {
            return evitaSessionServiceClient.Invoke(new
            EvitaSessionService.EvitaSessionServiceClient(channel.Invoker));
      }
      finally
      {
            _channelPool.ReturnChannel(channel);
      }
}
```

Ukázka kódu 22 - Metoda k aplikaci předané logiky na kanálu získaném pomocí instance třídy *ChannelPool*. Zdroj: [autor]

6.2.6. Dotazování se na gRPC API pomocí Query

V kapitole 6.1.7 byla ukázka obsluhy metody query na straně gRPC API, zde se nachází vysvětlení implementace téže metody ze strany klienta (ovladače) pro získání lepší představy o fungování komunikace mezi gRPC API a ovladačem. Zjednodušená ukázka implementace této metody se nachází níže.

Před samotným zahájením zpracování dotazu je nejprve ověřeno, že je aktuálně používaná relace stále aktivní a že požadovaný dotaz dává z hlediska typu požadované návratové hodnoty smysl. Tento typ, stejně jako očekávaná forma navrácených entit, je specifikován pomocí generických parametrů. Předaná instance typu Query obsahuje metodu převádějící tuto strukturu na již vysvětlenou formu parametrizovaného dotazu, viz. 6.1.5. Z této struktury lze za použití konverteru parametrů vytvořit strukturu požadavku pro vykonání dotazu definovanou v protokolu. Samotný dotaz na API je poslán jako delegát do metody řešící channel pooling (viz. 6.2.5), po obdržení odpovědi proběhne převod dodatečně vypočtených dat do odpovídajících struktur. Podle generický požadovaného typu je zvolen a použit vhodný převaděč gRPC struktur do tříd inspirovaných evitaDB implementací. Zde je vhodné povšimnout si druhého parametru, jenž definuje funkci přijímající dvojici parametrů sloužících k získání požadovaného schématu z mezipaměti, viz. 6.2.4. Pokud by jimi definované schéma v mezipaměti obsaženo nebylo a nepodařilo by se jej získat předanými funkcemi, dotaz by skončil výjimkou.

```
public T Query<T, TS>(Query query) where TS : IEntityClassifier where T :
EvitaResponse<TS>
{
      AssertActive();
      AssertRequestMakesSense<TS>(query);

      StringWithParameters stringWithParameters =
query.ToStringWithParametersExtraction();
```

```
        var request = new GrpcQueryRequest
        {
        Query = stringWithParameters.Query,
        PositionalQueryParams =
{stringWithParameters.Parameters.Select(QueryConverter.ConvertQueryParam)}
        };
        GrpcQueryResponse grpcResponse =
        await ExecuteWithEvitaSessionService(async session => await
session.QueryAsync(request));

        IEvitaResponseExtraResult[] extraResults = grpcResponse.ExtraResults is
not null
        ? QueryConverter.ToExtraResults(grpcResponse.ExtraResults)
        : Array.Empty<IEvitaResponseExtraResult>();

        if (typeof(SealedEntity).IsAssignableFrom(typeof(TS)))
        {
        IDataChunk<SealedEntity> recordPage = QueryConverter.ConvertToDataChunk(
                grpcResponse,
                grpcRecordPage => EntityConverter.ToSealedEntities(
                        grpcRecordPage.SealedEntities.ToList(),
                        (entityType, schemaVersion) =>
                        _schemaCache.GetEntitySchemaOrThrow(
                                entityType, schemaVersion, FetchEntitySchema,
                                GetCatalogSchema
                        )
                )
        );
        return (new EvitaEntityResponse(query, recordPage, extraResults) as T)!;
        }
        throw new EvitaInvalidUsageException("Unsupported return type `" +
        typeof(TS) + "`!");
}
```

Ukázka kódu 23 - Ukázka klientské implementace *Query* metody. Zdroj: [autor]

6.2.7. Test ověřující správnou funkčnost požadavků na prototyp ovladače

Testy je v jazyce C# možné psát ve třech různých testovacích frameworcích, přičemž pro řešený ovladač byl vybrán NUnit. Volba byla učiněna na základě skutečnosti, že tato knihovna je C# adaptací knihovny JUnit, což je nejpoužívanější knihovna pro testování v jazyku Java. V mnoha ohledech si jsou podobné, používají stejná klíčová slova a sdílejí i mnohé koncepty z hlediska stylistiky a samotného psaní testů.

Na ukázce níže je podoba testu, který testuje většinu z dílčích požadavků, které byly kladeny na prototyp ovladače evitaDB databáze, viz. kapitola 5. Z ukázky byly odebrány veškeré komentáře a Asserty ověřující správnost získaných dat pro lepší vizualizaci implementovaných funkcí.

```
[Test]
public async Task
ShouldBeAbleTo_CreateCatalog_And_EntitySchema_AndInsertNewEntity_WithAttribute(
)
{
	CatalogSchema catalogSchema =
	await _client.DefineCatalogAsync(TestCatalog);

	using (var rwSession = _client.CreateReadWriteSession(TestCatalog))
	{
		catalogSchema = rwSession.UpdateAndFetchCatalogSchema(new
		CreateEntitySchemaMutation(TestCollection));
		CreateAttributeSchemaMutation createAttributeDecimalRange = new
		CreateAttributeSchemaMutation(
		AttributeDecimalRange, nameof(AttributeDecimalRange), null, false,
		true, true, false, true,
		typeof(DecimalNumberRange), null, 2);

		catalogSchema = rwSession.UpdateAndFetchCatalogSchema(new
		ModifyEntitySchemaMutation(TestCollection,
		createAttributeDecimalRange));

		var entitySchema = rwSession.GetEntitySchema(TestCollection);
		rwSession.GoLiveAndClose();
	}

	using var newSession = _client.CreateReadWriteSession(TestCatalog);

	var newEntity = newSession.UpsertAndFetchEntity(
		new EntityUpsertMutation(
			TestCollection,
			null,
			EntityExistence.MayExist,
			new UpsertAttributeMutation(AttributeDecimalRange,
			DecimalNumberRange.Between(1.2m, 100))
		), AttributeContent()
	);
}
```

Ukázka kódu 24 - Ukázka testu zobrazující implementované funkce. Zdroj: [autor]

6.2.8. Srovnání použití metod pro dotazování mezi Java a C# implementací

V této podkapitole bude porovnání definice a poslání dotazu k nalezení odpovídajících dat mezi Java implementací (embedovaná verze evitaDB / Java ovladač) a implementovaným prototypem ovladače naprogramovaném v jazyku C#. Dotaz vyhledává v kolekci produktů, přičemž mají být vráceny pouze produkty, jejichž atribut “url” neobsahuje hodnotu “test.com” a zároveň platí alespoň jedna z následujících podmínek: buď jejich primární klíče spadají do množiny (677, 678, 679, 680) nebo musí mít aktuálně platnou cenu alespoň v jednom z ceníků [“basic” nebo “vip”] v českých korunách, přičemž cena musí být v rozmezí 1.2 až 1 000 korun. Vrácené záznamy v podobě EntityReferencí (primárních klíčů) budou seřazeny od nejdražšího k nejlevnějšímu podle vypočtené prodejní ceny. Seznam obdržených entit by se měl skládat z prvních 20 nalezených záznamů a odpověď by měla obsahovat i dodatečně

vypočtený výsledek v podobě objektu QueryTelemetry. Na následující ukázce z programovacího jazyka Java je uveden příklad, v němž je popsaný dotaz poslán na server, přičemž je specifikován požadovaný návratový typ *EvitaEntityReferenceResponse*, jenž reflektuje očekávaný typ vrácených entit.

```
EvitaEntityReferenceResponse referencesResponse =
client.queryCatalog(existingCatalogWithData, session -> {
	return session.query(
	query(
		collection("Product"),
		filterBy(
		and(
			not(
				attributeContains("url", "test.com")
			),
			or(
				entityPrimaryKeyInSet(677, 678, 679, 680),
				and(
					priceBetween(new BigDecimal("1.2"), new
					BigDecimal("1000")),
					priceValidIn(OffsetDateTime.now()),
					priceInPriceLists("basic", "vip"),
					priceInCurrency(Currency.getInstance("CZK"))
				)
			)
		)
	),
	orderBy(
		priceNatural(OrderDirection.DESC)
	),
	require(
		page(1, 20),
		dataInLocales(new Locale("en", "US"), new Locale("cs", "CZ")),
		queryTelemetry()
	)
	),
	EntityReference.class
	    );
	}
);
```

Ukázka kódu 25 - Ukázka volání *query* metody z Java implementace evitaDB. Zdroj: [autor]

Tentýž dotaz je na ukázce níže realizován a poslán na server pomocí prototypu klienta naprogramovaném v jazyku C#.

```
EvitaEntityReferenceResponse referenceResponse =
_client!.QueryCatalog(ExistingCatalogWithData,
session =>
{
	return session.Query<EvitaEntityReferenceResponse, EntityReference>(
	Query(
		Collection("Product"),
		FilterBy(
			And(
				Not(
					AttributeContains("url", "test.com")
				),
				Or(
					EntityPrimaryKeyInSet(677, 678, 679, 680),
					And(
```

```
                                        PriceBetween(1.2m, 1000),
                                        PriceValidIn(DateTimeOffset.Now),
                                        PriceInPriceLists("basic", "vip"),
                                        PriceInCurrency(new Currency("CZK"))
                                    )
                                )
                            )
                        ),
                        OrderBy(
                            PriceNatural(OrderDirection.Desc)
                        ),
                        Require(
                            Page(1, 20),
                            DataInLocales(new CultureInfo("en-US"), new CultureInfo("cs-
                            CZ")),
                            QueryTelemetry()
                        )
                    ));
});
```

Ukázka kódu 26 - Ukázka volání *Query* metody pomocí implementovaného C# ovladače pro evitaDB. Zdroj: [autor]

Jak vyplývá z uvedených ukázek kódu, C# adaptace je originální implementaci velmi podobná. Některé aspekty byly realizovány odlišným způsobem, které jsou v daném jazyku běžně používané. Změny jsou samozřejmě také v pojmenování, které je obsahově stejné, metody ale začínají dle konvencí velkými písmeny. Třída *Locale*, jež je obsažena v Javě, zde byla nahrazena její existující alternativou *CultureInfo* s relativně podobnou množinou vlastností.

7. Závěr

Tato diplomová práce se zabývala in-memory databází evitaDB, která se specializuje na oblast e-commerce. Nejdříve proběhlo představení dané databáze, u níž bylo nutné seznámit se s používanými pojmy a principy a také s možnostmi jejího používání. Ta měla před začátkem této práce hned tři: použití napřímo z programovacího jazyka Java, nebo použití jednoho z dostupných API: REST či GraphQL. Tato API byla navržena s cílem co nejpříjemnější klientské konzumace a tak, aby si byla z hlediska návrhu co nejpodobnější pro umožnění přecházet z jednoho na druhé bez větších obtíží. Bylo také žádoucí, aby se jejich struktura mohla přizpůsobit uživatelem specifikované terminologii použité při návrhu databáze. Plánována byla také realizace třetího typu API přes framework gRPC, jenž je známý svojí velkou rychlostí v mezi-serverové komunikaci. Původním cílem bylo přiblížit jej ostatním již provozovaným API a jimi používaným konceptům pro usnadnění celkové údržby databáze při provádění změn či přidávání nových funkcí.

U každého ze tří zmíněných API byla detailně rozebrána funkcionalita a obvyklé metody jejich používání jak ze serverové, tak i klientské strany. Zároveň byly u každého vyzdvihnuty klady i zápory nejprve z obecného hlediska a následně z jejich (v případě gRPC teoretického) používání při obsluze evitaDB. U žádného z nich nelze říct, že by bylo lepší než jiné, každé má své přednosti a specifické případy užití. GraphQL se vyznačuje především přívětivostí při konzumaci, jelikož při použití podporovaného vývojového prostředí přináší mezi programátory oblíbené funkce, jako je kontextové našeptávání nebo validace na straně klienta. Za zprostředkování těchto funkcí však stojí výrazné snížení výkonu, což bylo potvrzeno výkonnostním porovnáním s REST API, v němž dosáhlo pouhé poloviny provedených operací za sekundu. Z toho vyplývá, že je REST sice dvakrát tak rychlý a jeho použití s sebou přináší celou řadu výhod, například snadno zakomponovatelné prvky pro zabezpečení API a také jeho jednoduchost vystavení i konzumace. Nicméně jeho používání není ani zdaleka tak příjemné pro běžné případy užití, spíše by mohlo najít uplatnění při strojovém používání databáze, kupříkladu při tvorbě scriptů nebo automatizaci některých procesů. Po prozkoumání možností gRPC bylo rozhodnuto, že jej není vhodné implementovat stejným způsobem jako ostatní API, především z důvodu nemožnosti přizpůsobení se dynamické struktuře databáze definované klienty. Vzhledem k jeho vynucenému používání statického a předem definovaného protokolu pro něj bylo nalezeno alternativní využití pro implementaci databázového ovladače.

Na základě tohoto rozhodnutí bylo nezbytné vymezit požadavky na vzniklé ovladače – gRPC mělo na serverové straně vystupovat jako univerzální API, jenž mělo sloužit pro jazykově neutrální komunikaci mezi ovladači a databází. Na základě těchto požadavků byl vytvořen prototyp ovladače v jiném jazyce, než je implementována samotná evitaDB.

Vzhledem ke zkušenostem autora byl pro účel vytváření prvního takového řešení zvolen jazyk C#, v němž bylo na základě specifikovaných požadavků potřebné implementovat všechny nabízené funkce v omezeném rozsahu pro potvrzení funkčnosti prototypu. Mezi tyto funkce spadal návrh schémat, vytváření katalogů a kolekcí entit, do nichž mělo být možné vkládat entity obsahující některá data z jí přidružených dat. Stěžejní částí bylo také zajistit, aby bylo možné vytvářet složité filtrační a výpočetní dotazy nad databází.

Hlavním cílem tohoto prototypu bylo dosáhnout co nejpodobnějšího ovládání databáze přímému používání vestavěného Java API. Díky tomu by mohla být databáze ovládána i vzdáleně, což by umožnilo její provoz i na odlišném stroji, než na kterém by běžela jí používaná aplikace, díky čemuž by bylo dosaženo lepší separace z hlediska správy hardwarových prostředků. Jedním ze stěžejních motivů pro oddělení bylo i umožnění provozovat evitaDB databázi v kontejnerizovaném prostředí pomocí Dockeru.

Všechny z vymezených cílů (požadavků) na realizaci ovladače s využitím gRPC se podařilo úspěšně splnit, výsledkem praktické části je na straně databázového serveru funkční gRPC API kopírující všechny funkce evitaDB Java API. Na klientské straně vznikl prototyp ovladače vytvořený na platformě .NET, který toto API využívá ke vzdálenému volání metod na serveru. Pro ověření návrhu gRPC API byl implementován autorem databáze i Java ovladač využívající společná rozhraní z evitaDB, jenž ověřil, že API splňuje veškeré náležitosti na možnosti komunikace s minimální ztrátou rychlosti. Vytvořený prototyp bude v budoucnu i nadále vyvíjen a rozvíjen, aby jím bylo možné databázi plně obsluhovat.

8. Seznam zkratek

ACID – Atomic-Consistent-Isolated-Durable

API – Application-Programming-Interface

B2B – Business-to-business

B2C – Business-to-consumer

CRUD – Create-Read-Update-Delete

gRPC – g-Remote-Procedure-Call

HTML – Hypertext-Transfer-Protocol

JKS – Java-KeyStore

JSON – JavaScript-Object-Notation

NoSQL – Not-Only-SQL, Non-SQL

POJO – Plain-Old-Java-Object

QUIC – Quick-UDP-Internet-Connections

REST – Representational state transfer

RPC – Remote-Procedure-Call

SQL – Structured-query-language

TCP – Transmission-Control-Protocol

TLS – Transport-Layer-Security

UDP – User-Datagram-Protocol

WAL – Write-Ahead-Logu

XML – Extensible-Markup-Language

9. Seznam použitých zdrojů

[1] FG Forrest, a s. About project. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/en/project-info

[2] NOVOTNÝ, Jan. Research assignment - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/research/assignment/index

[3] NOVOTNÝ, Jan. Research evaluation - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/research/evaluation/evaluation

[4] What Is an In-Memory Database? *Amazon Web Services, Inc.* [online]. [vid. 2023-04-25]. Dostupné z: https://aws.amazon.com/nosql/in-memory/

[5] MEDJAHED, Brahim, Mourad OUZZANI a Ahmed K. ELMAGARMID. Generalization of ACID Properties. In: LING LIU a M. TAMER ÖZSU, ed. *Encyclopedia of Database Systems* [online]. Boston, MA: Springer US, 2009 [vid. 2023-04-25], s. 1221–1222. ISBN 978-0-387-39940-9. Dostupné z: doi:10.1007/978-0-387-39940-9_736

[6] FG Forrest, a s. evitaDB - Fast e-commerce database. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/en

[7] HORNYCH, Lukáš. Documentation - REST. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/connectors/rest

[8] HORNYCH, Lukáš. Documentation - GraphQL. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/connectors/graphql

[9] NOVOTNÝ, Jan. Data model - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/data-model

[10] NOVOTNÝ, Jan. Data types - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/data-types

[11] GUAGENTI, calogero. CZK To IRR: Convert Czech Koruna to Iranian Rial. *Forbes Advisor* [online]. 25. duben 2023 [vid. 2023-04-25]. Dostupné z: https://www.forbes.com/advisor/money-transfer/currency-converter/czk-irr/

[12] RESEARCH, BitMEX. Growth In The Level Of Precision Of Bitcoin Spending. *BitMEX Blog* [online]. 27. leden 2020 [vid. 2023-04-25]. Dostupné z: https://blog.bitmex.com/bitcoin-transaction-output-value-precision/

[13] TRAYSR. Commonly used IETF language tags. *Gist* [online]. [vid. 2023-04-25]. Dostupné z: https://gist.github.com/traysr/2001377

[14] *JDK 17 Supported Locales* [online]. [vid. 2023-04-25]. Dostupné z: https://www.oracle.com/java/technologies/javase/jdk17-suported-locales.html

[15] ISO - ISO 4217 — Currency codes. *ISO* [online]. [vid. 2023-04-25]. Dostupné z: https://www.iso.org/iso-4217-currency-codes.html

[16] NOVOTNÝ, Jan. *Range - evitaDB* [online]. Java. B.m.: FG Forrest. 12. duben 2023 [vid. 2023-04-25]. Dostupné z: https://github.com/FgForrest/evitaDB/blob/1e027e114707d86783cf8603f99e8f1b821794ba/evita_common/src/main/java/io/evitadb/dataType/Range.java

[17] GUPTA, Lokesh. Java record Type with Examples. *HowToDoInJava* [online]. 17. květen 2020 [vid. 2023-04-25]. Dostupné z: https://howtodoinjava.com/java14/java-14-record-type/

[18] NOVOTNÝ, Jan. Query language - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/research/assignment/querying/query_language

[19] NOVOTNÝ, Jan. Price filtering - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/query/filtering/price#price-for-sale-computation-algorithm

[20] NOVOTNÝ, Jan. Schema - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/schema

[21] 30.3. Write-Ahead Logging (WAL). *PostgreSQL Documentation* [online]. 9. únor 2023 [vid. 2023-04-25]. Dostupné z: https://www.postgresql.org/docs/15/wal-intro.html

[22] NOVOTNÝ, Jan. Write data - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/use/api/write-data

[23] FEKETE, Alan, Dimitrios LIAROKAPIS, Elizabeth O'NEIL, Patrick O'NEIL a Dennis SHASHA. Making snapshot isolation serializable. *ACM Transactions on Database Systems* [online]. 2005, **30**(2), 492–528. ISSN 0362-5915. Dostupné z: doi:10.1145/1071610.1071615

[24] HERLIHY, Maurice. Optimistic concurrency control for abstract data types. *ACM SIGOPS Operating Systems Review* [online]. 1987, **21**(2), 33–44. ISSN 0163-5980. Dostupné z: doi:10.1145/24601.24604

[25] NOVOTNÝ, Jan. *Versioned - evitaDB* [online]. Java. B.m.: FG Forrest. 12. duben 2023 [vid. 2023-04-25]. Dostupné z: https://github.com/FgForrest/evitaDB/blob/1e027e114707d86783cf8603f99e8f1b821794ba/evita_api/src/main/java/io/evitadb/api/requestResponse/data/Versioned.java

[26] Google Trends. *Google Trends* [online]. [vid. 2023-04-25]. Dostupné z: https://trends.google.com/trends/explore?date=today%205-y&q=gRPC,GraphQL,REST%20API&hl=cs-CZ

[27] HORNYCH, Lukáš. Choosing http server - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/blog/03-choosing-http-server

[28] *HTTP response status codes - HTTP | MDN* [online]. 10. duben 2023 [vid. 2023-04-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[29] *JSON Data Types* [online]. [vid. 2023-04-25]. Dostupné z: https://www.w3schools.com/js/js_json_datatypes.asp

[30] SÉGURET, Denys. Answer to „Javascript long integer". In: *Stack Overflow* [online]. 26. červen 2013 [vid. 2023-04-25]. Dostupné z: https://stackoverflow.com/a/17320771

[31] What is REST. *REST API Tutorial* [online]. 7. duben 2022 [vid. 2023-04-25]. Dostupné z: https://restfulapi.net/

[32] *OpenAPI Specification - Version 3.0.3 | Swagger* [online]. [vid. 2023-04-25]. Dostupné z: https://swagger.io/specification/

[33] PREVATO, Roberto. *BlackSheep - OpenAPI Docs* [online]. [vid. 2023-04-25]. Dostupné z: https://www.neoteroi.dev/blacksheep/openapi/

[34] *GraphQL | A query language for your API* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/

[35] *Thinking in Graphs | GraphQL* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/learn/thinking-in-graphs/

[36] *Introspection | GraphQL* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/learn/introspection/

[37] *Validation | GraphQL* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/learn/validation/

[38] *Serving over HTTP | GraphQL* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/learn/serving-over-http/

[39] *Advantages and Disadvantages of GraphQL | Stable Kernel* [online]. [vid. 2023-04-25]. Dostupné z: https://stablekernel.com/article/advantages-and-disadvantages-of-graphql/

[40] *Batching | GraphQL Java* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql-java.com/documentation/batching/

[41] *GraphQL Code Libraries, Tools and Services* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/code/

[42] *Queries and Mutations | GraphQL* [online]. [vid. 2023-04-25]. Dostupné z: https://graphql.org/learn/queries/

[43] Subscriptions. *Apollo Docs* [online]. [vid. 2023-04-25]. Dostupné z: https://www.apollographql.com/docs/react/data/subscriptions/

[44] gRPC. *gRPC* [online]. [vid. 2023-04-25]. Dostupné z: https://grpc.io/

[45] About gRPC. *gRPC* [online]. [vid. 2023-04-25]. Dostupné z: https://grpc.io/about/

[46] Understanding gRPC Concepts, Use Cases & Best Practices. *InfraCloud* [online]. [vid. 2023-04-25]. Dostupné z: https://www.infracloud.io/blogs/understanding-grpc-concepts-best-practices/

[47] *FgForrest/EvitaExternalAPIPerformanceTest: The repository contains performance tests that measure performance of Evita API across different protocols - gRPC, GraphQL, REST API* [online]. [vid. 2023-04-25]. Dostupné z: https://github.com/FgForrest/EvitaExternalAPIPerformanceTest

[48] Protocol Buffer Basics: Java. *Protocol Buffers Documentation* [online]. [vid. 2023-04-25]. Dostupné z: https://protobuf.dev/getting-started/javatutorial/

[49] Authentication. *gRPC* [online]. [vid. 2023-04-25]. Dostupné z: https://grpc.io/docs/guides/auth/

[50] SWAYAMRAINA. Answer to „Why do we need to register reflection service on gRPC server". In: *Stack Overflow* [online]. 31. červenec 2020 [vid. 2023-04-25]. Dostupné z: https://stackoverflow.com/a/63190406

[51] *Server Reflection • Akka gRPC* [online]. [vid. 2023-04-25]. Dostupné z: https://doc.akka.io/docs/akka-grpc/current/server/reflection.html

[52] KURUPPU, Danesh. gRPC: A Deep Dive into the Communication Pattern. *The New Stack* [online]. 31. srpen 2021 [vid. 2023-04-25]. Dostupné z: https://thenewstack.io/grpc-a-deep-dive-into-the-communication-pattern/

[53] Supported languages. *gRPC* [online]. [vid. 2023-04-25]. Dostupné z: https://grpc.io/docs/languages/

[54] *Protocol Buffers* [online]. 2022 [vid. 2023-04-25]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Protocol_Buffers&oldid=1112799194

[55] Protocol Buffers. *Protocol Buffers Documentation* [online]. [vid. 2023-04-25]. Dostupné z: https://protobuf.dev/

[56] Overview. *Protocol Buffers Documentation* [online]. [vid. 2023-04-25]. Dostupné z: https://protobuf.dev/overview/

[57] Language Guide (proto 3). *Protocol Buffers Documentation* [online]. [vid. 2023-04-25]. Dostupné z: https://protobuf.dev/programming-guides/proto3/

[58] MORAWSKI, Michał a Przemyslaw IGNACIUK. Influence of Congestion Control Algorithms on Head-of-Line Blocking in MPTCP-based Communication. In: *2019 27th Telecommunications Forum (TELFOR)*: *2019 27th Telecommunications Forum (TELFOR)* [online]. 2019, s. 1–4. Dostupné z: doi:10.1109/TELFOR48224.2019.8971059

[59] JOLY, Jean-Marie. gRPC over HTTP/3. *SafetyCulture Engineering* [online]. 14. červenec 2020 [vid. 2023-04-25]. Dostupné z: https://medium.com/safetycultureengineering/grpc-over-http-3-53f41fc0761e

[60] IYENGAR, Jana a Ian SWETT. *QUIC Loss Detection and Congestion Control* [online]. Request for Comments. RFC 9002. B.m.: Internet Engineering Task Force. 2021 [vid. 2023-04-25]. Dostupné z: doi:10.17487/RFC9002

[61] ZANINI, Antonello. Will Google's QUIC Protocol Replace TCP? *Medium* [online]. 7. prosinec 2021 [vid. 2023-04-25]. Dostupné z: https://levelup.gitconnected.com/will-googles-quic-protocol-replace-tcp-6ed991a0ca1e

[62] *gRPC RFCs* [online]. B.m.: grpc. 18. duben 2023 [vid. 2023-04-25]. Dostupné z: https://github.com/grpc/proposal/blob/c994c3cd4e3baee501e56f35f529675626d5e184/G2-http3-protocol.md

[63] POZLER, Tomáš. Setting up TLS - evitaDB. *EvitaDB - Fast e-commerce database* [online]. [vid. 2023-04-25]. Dostupné z: https://evitadb.io/documentation/operate/tls

[64] Determining if an Open Generic Type IsAssignableFrom a Type. *Glacius* [online]. 10. leden 2022 [vid. 2023-04-25]. Dostupné z: https://glacius.tmont.com/articles/determining-if-an-open-generic-type-isassignablefrom-a-type

10. Seznam obrázků

11. Seznam tabulek

12. Seznam grafů

13. Seznam ukázek kódu

Zadání diplomové práce

Autor: **Bc. Tomáš Pozler**

Studium: I2100077

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Použití gRPC pro protokol ovladače databázového systému**

Název diplomové práce AJ: Database system driver protocol based on gRPC

Cíl, metody, literatura, předpoklady:

Cíl práce: Hlavním cílem práce bude implementace serverové části API v programovacím jazyku Java, které umožňuje klientům komunikovat a ovládat tuto databázi pomocí protokolu gRPC. Následně bude v rámci práce pro toto API vyvinut ovladač (klient) v jazyku C#.

Osnova:

- Úvod
- EvitaDB
- Rozbor a porovnání protokol
  - Výhody a nevýhody REST API
  - Výhody a nevýhody GraphQL
  - Výhody a nevýhody gRPC
- Konzumace gRPC protokolu na straně klienta
  - Požadavky na ovladače
  - Specifika gRPC protokolu v .NET prostředí
- Realizace praktické části
- Závěr

Oficiální web EvitaDB: https://evitadb.io

Zadávající pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Datum zadání závěrečné práce: 26.1.2021