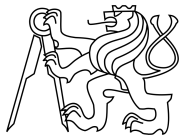
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Návrh a prototypová implementace aplikační vrstvy nad metadatovým úložištěm projektu Manta
Student:	Bc. Ondrej Bernát
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1. Vypracujte rešerši formátů serializace reprezentace grafů.
2. Analyzujte a navrhněte sadu atomických operací nad metadatovým úložištěm projektu Manta. Inspirujte se rozhraními pro grafové databáze, respektujte však logický model projektu Manta [1]. Mezi tyto operace bude patřit export a import grafu (viz bod 1).
3. Analyzujte možnosti návrhu doménově specifického jazyka (DSL) pro komplexní analýzy nad metadatovým úložištěm (analýza bezpečnostních rizik, analýza dopadu změn struktury apod.) a diskutujte možná řešení (vlastní deklarativní jazyk nebo rozšíření programovacího jazyka o příslušné třídy apod.).
4. Dle výsledku analýzy proveďte návrh a prototypovou implementaci aplikační vrstvy, která bude obsahovat atomické operace (bod 2) a DSL jazyk (bod 3).
5. Aplikační vrstvu otestujte realizací alespoň jedné v praxi používané komplexní analýzy.

Seznam odborné literatury

[1] Michal Peroutka: Optimální struktura a indexy modelu metadatového úložiště v grafové databázi. Diplomová práce ČVUT FIT. 2016.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 16. listopadu 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práca

Návrh a prototypová implementace aplikační vrstvy nad metadatovým úložištěm projektu Manta

Bc. Ondrej Bernát

Vedúci práce: Ing. Michal Valenta, Ph.D.

2. mája 2017

Poďakovanie

Chcel by som sa poďakovať Tomášovi Smolíkovi za to, že mi umožnil pracovať na rozvoji projektu Manta a dokázal tak spojiť akademickú činnosť s praxou, čo mi prinieslo mnoho užitočných skúseností, Tomášovi Fechnerovi za mnoho užitočných rád pri implementácii riešenia a Michalovi Valentovi za to, že viedol túto diplomovú prácu.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 2. mája 2017 .....

Odkaz na túto prácu

Bernát, Ondrej. *Návrh a prototypová implementace aplikační vrstvy nad metadatovým úložištěm projektu Manta.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Táto diplomová práca sa zaoberá návrhom a prototypovou implementáciou aplikačnej vrstvy nad metadatovým úložiskom. Metadatové úložisko je implementované formou grafovej databázy so značne zložitým a neprehľadným dátovým modelom.

Cieľom tejto práce je navrhnúť aplikačnú vrstvu tak, aby odstienila užívateľa od zložitých technických detailov a umožnila jednoduché používanie. Aplikačná vrstva musí poskytnúť rozhranie s atomickými operáciami a umožniť import a export serializovaného grafu. Povinnou súčasťou riešenia je aj doménovo špecifický jazyk, ktorý umožní implementáciu zložitých analýz nad metadatovým úložiskom.

Klíčová slova graf, grafová databáza, metadata, logický model, doménovo špecifický jazyk, programové rozhranie aplikácie, Manta

Abstract

This diploma thesis deals with design and prototype implementation of application layer over metadata storage. Metadata storage is implemented as graph database with very complex and vast data model.

The goal of this thesis is to design application layer the way, it will hide complicated technical details from user and provides simple usage. Application layer must provide interface with atomic operations and make possible to import and export serialized graph. Mandatory feature of the solution is also domain specific language (DSL), that will provide support for implementation of complex analysis over metadata storage.

Keywords graph, graph database, metadata, logical model, domain specific language, application programming interface, Manta

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Témou tejto diplomovej práce je zanalyzovať, navrhnúť a vytvoriť prototypovú implementáciu aplikačnej vrstvy nad metadátovým úložiskom Manta. Metadátové úložisko je realizované pomocou grafovej databázy, čo prináša množstvo výziev, ktoré je potrebné pri realizácii aplikačnej vrstvy vyriešiť.

V prvej kapitole zoznamujem čitateľa s projektom Manta a jeho funkciami. Ďalej je tu popísané prostredie dátového skladu, pre ktorý je systém Manta určený a sú tu vysvetlené základné pojmy, ktoré s tým súvisia.

V druhej kapitole sa venujem vysvetleniu cieľa tejto práce a jeho presnejšej špecifikácii a tomu, prečo je potrebné tento cieľ dosiahnuť.

V tretej kapitole uvádzam požiadavky, ktoré musí toto riešenie spĺňať a analyzujem súčasnú situáciu. Ďalej sa pozerám na rôzne možnosti riešenia požiadaviek a spracúvam rešerše serializácie grafu.

V štvrtej kapitole sa venujem návrhu konkrétneho riešenia, výberu najvhodnejších technológií a postupov a rozdeleniu riešenia na funkčné bloky. Ďalej tu uvádzam presné rozvrhnutie atomických operácií a možnosti doménovo špecifického jazyka.

V piatej kapitole rozoberám samotnú realizáciu projektu. Upresňujem tu detaily implementácie, návrhové vzory a rozvrhnutie funkčných blokov na modely tried a závislosti medzi nimi.

V ďalšej kapitole zoznamujem čitateľa s procesom testovania tejto aplikácie. Ďalej sa pozerám na metódy, ktorými je dosiahnutá vysoká kvalita a udržateľnosť zdrojového kódu, čo prispieva k zvýšeniu kvality a zníženiu chybovosti celého softvérového projektu.

V predposlednej kapitole sa venujem samotnému nasadeniu aplikácie do produkčného prostredia a usporiadaniu jednotlivých častí, ktoré medzi sebou komunikujú.

V poslednej kapitole popisujem výsledky riešenia pomocou dokumentácie, ktorá je nutnou súčasťou každého softvérového projektu. Dokumentácia je v tomto prípade obzvlášť dôležitá, pretože aplikačná vrstva umožňuje integráciu s inými systémami.

Kapitola 1

Projekt Manta

Objem dát vo svete zaznamenáva v posledných rokoch obrovský nárast. Predpokladá sa, že počas nasledujúcich piatich rokov objem dát vzrastie každoročne o 20 Exabajtov [1]. Tento trend ešte viac posiluje informatizácia oblastí ako je štátna správa, školstvo alebo zdravotníctvo.

Pre oblasti ako je bankovníctvo, finančné trhy a veľké korporácie je produkcia, spracovávanie a uchovávanie obrovského množstva dát samozrejmosťou už dlhšiu dobu. Na tieto účely slúžia v týchto organizáciách dátové sklady.

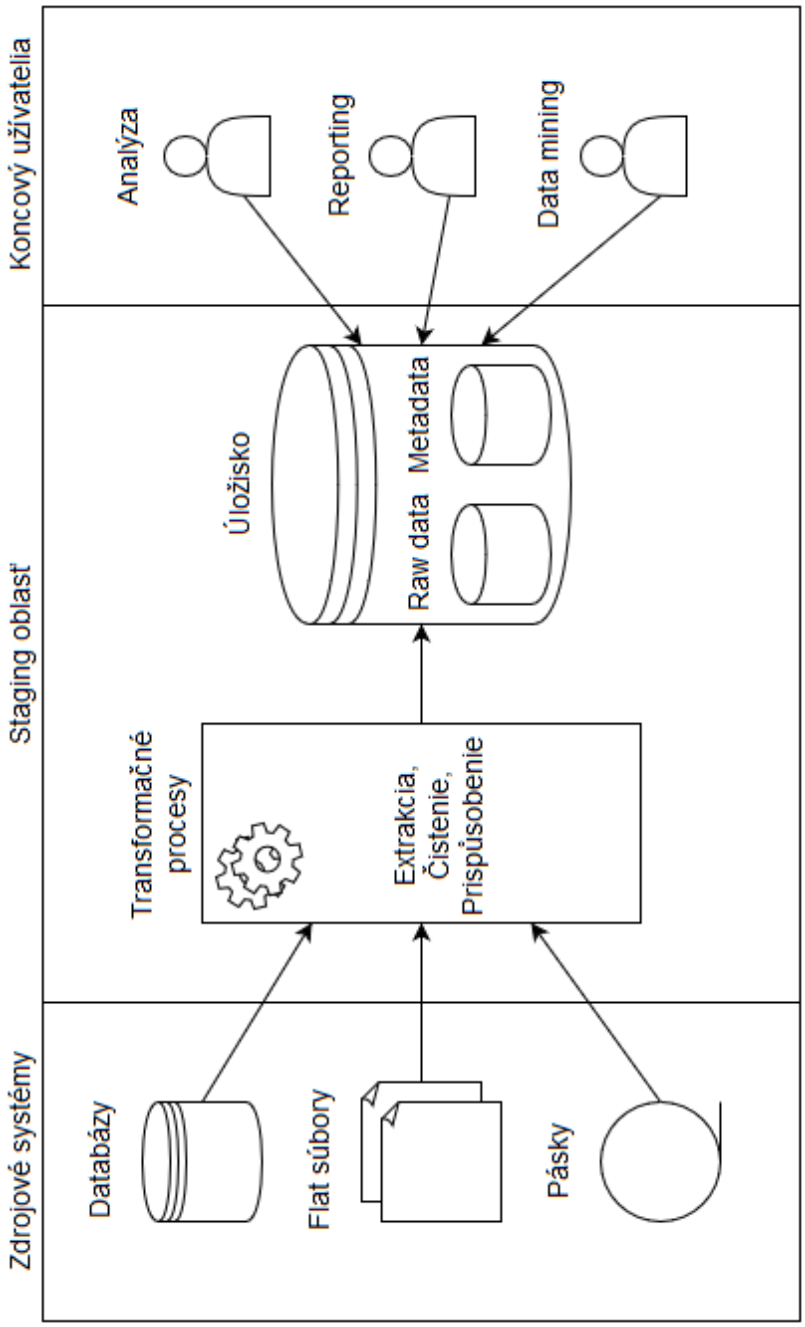
1.1 Dátový sklad

Dátový sklad sa stal nevyhnutným technologickým základom pre množstvo možných riešení ako zvýšiť konkurencieschopnosť a ziskovosť firmy. Princípom je premeniť veľké množstvá vznikajúcich dát na hodnotné informácie a nové pohľady na dáta, ktoré sa dajú využiť manažmentom firmy na prijímanie lepších rozhodnutí.

Stručný popis dátového skladu prevzatý z [2]: "Dátový sklad je systém, ktorý extrahuje, čistí, prispôsobuje a doručuje zdrojové dáta do dimenzionálneho dátového úložiska a následne nad ním podporuje a implementuje dotazovanie a analýzu pre účely rozhodovania."

Dátový sklad má v organizáciách úlohu podporného systému, avšak náklady na chod takéhoto skladu nie sú malé. Oveľa vačšie náklady však vznikajú z prijímania chybných rozhodnutí, ktoré sú spôsobené nedostatkom informácií.

Dáta sú do dátových skladov vkladané dlhé roky a je zrejmé, že získavanie dát z týchto systémov pre analytické účely je príliš zložité. Kvôli tomu je dlhodobý koncept pri budovaní dátového skladu veľmi dôležitý a týka sa množiny štandardných komponentov, ktoré sú znázornené na diagrame dátového skladu, obrázok 1.1.



Obr. 1.1: Príklad štruktúry dátového skladu

1.1.1 Komponenty dátového skladu

Najzákladnejšími a najdôležitejšími komponentami dátového skladu sú dátové procesy, dimenzionálne úložiská a podpora prezentácie dát koncovým užívateľom. Jednotlivé komponenty bývajú fyzicky oddelené, čiže bežia na rôznych strojoch a sú spravované odlišnými ľuďmi.

1.1.1.1 Dátové procesy

Dátové procesy znamenajú akékoľvek spracovávanie dát, samotné dáta sa nemusia nutne meniť, často sa získajú iba metadata. Dátové procesy sa dajú popísať ako získanie dát, prevod dát do vhodnej podoby a ich uloženie do štruktúry, ktorá je vhodná na dotazovanie, v angličtine nazývané ETL, čiže Extract, Transform, Load.

1.1.1.2 ETL

ETL systém je základom dátového skladu. Jeho úlohou je získavanie, úprava a uloženie dát. Správne navrhnutý ETL systém dáta najprv extrahuje zo zdrojových systémov, ktoré môžu byť napríklad databázy alebo rôzne súborové systémy, potom na nich prevedie transformácie za účelom čistenia dát alebo radenia dát a nakoniec ich uloží do cieľového úložiska vo vhodne štruktúrovanej forme.

Použitie ETL výrazne zvyšuje kvalitu dát. Nejedná sa len o jednoduché presunutie dát, ale súčasťou může byť aj napríklad agregácia dát z rôznych zdrojov alebo identifikácia a zahodenie chybných údajov. Dôležité je taktiež uloženie spracovaných údajov v takej forme, aby bolo poskytovanie dát pre koncových užívateľov čo najefektívnejšie.

1.1.1.3 Dimenzionálne úložisko

Dimenzionálne úložisko obsahuje samotné dáta, ktoré sú uložené v štruktúrovanej podobe a metadata, ktoré s uloženými dátami súvisia. Úlohou dimenzionálneho úložiska je poskytovať tabuľky s dátami a metadátami prezentačnej vrstve. Priamy prístup koncových užívateľov do úložiska nie je možný.

Dôvody zakázaneho priameho prístupu:

- Poskytovanie detailnej bezpečnosti na aplikačnej úrovni
- Vytváranie výkonovo náročných indexov
- Poskytovanie nepretržitej dostupnosti služieb
- Garantovanie konzistencie dátových množín

1.1.1.4 Prezentačná vrstva

Priamy prístup do dimenzionálneho úložiska pre koncových užívateľov musí byť zakázaný, ináč môže dôjsť k narušeniu bezpečnosti dátového skladu. Tento prístup je umožnený pomocou prezentačnej vrstvy, ktorá umožňuje analýzu a dotazovanie sa nad dátami a metadatami.

Prezentačná vrstva je obvykle vytvorená pomocou BI (Business Inteligence) serverov a musí byť úzko skoordinovaná s budovaním a spravovaním samotného ETL systému. Úlohou ETL systému je dodávať prezentačnej vrstve modelované tabuľky s dátami a prezentačná vrstva potom poskytuje to, čo vidí samotný koncový užívateľ.

Hlavné úlohy prezentačnej vrstvy:

- Indexovanie tabuliek podľa častých dotazov
- Prevod užívateľských dotazov na technickú formu
- Bezpečnosť na úrovni tabuliek a riadkov
- Podpora užívateľských nástrojov na úrovni metadát

1.2 Metadata

Metadata sú štruktúrované dáta o dátach. K dátam pridávajú dodatočné informácie a uvádzajú ich do určitého kontextu, bez ktorého sú samotné dáta skoro bez informačnej hodnoty.

V rámci dátového skladu sa metadata dajú rozdeliť na procesné metadata a popisné metadata. Procesné metadata sa týkajú procesov extrakcie, transformácie a doručenia. Popisné metadata umožňujú vytváranie dotazovacích nástrojov a funkcií pre vytváranie reportov.

Procesné a popisné metadata sa prekrývajú, ale aj tak sa o nich dá uvažovať oddelene. Procesné metadata dávajú koncovým užívateľom možnosť zistiť, odkiaľ dáta uložené v dátovom sklade pochádzajú a popisné metadata umožňujú vytvorenie slovníka, ktorý obsahuje všetky dátové položky, ktoré sa nachádzajú v dátovom sklade.

1.2.1 Procesné metadata

Reprezentujú štatistiky bežiacich ETL procesov, ktoré sú generované pri behoch dávok. Procesné metadata sú rozhodujúce pre zistenie, či boli dáta v dátovom sklade sprocesované úspešne. Ďalej je ich možné využiť k odhaleniu slabých miest v procese. Obsahujú atribúty ako je počet záznamov, ktoré boli úspešné, počet zamietnutých záznamov, dobu trvania procesu a podobne.

1.2.2 Technické metadata

Reprezentujú technické aspekty dát, obsahujú atribúty ako dátový typ, veľkosť, lineage, výsledky dátových analýz a podobne.

- Súpis systémov - obsahuje definície všetkých systémov v prostredí dátového skladu. Každý systém má definované tabuľky, atribúty a ich dátové typy, ktoré sú použité počas ETL procesov.
- Dátové modely - sú znázornené vo forme diagramov schém a graficky zobrazujú metadata. Aj keď je spojenie tabuliek zobrazené v logickom modeli pomocou dátových tokov, sú tieto grafické modely využité k rýchlej identifikácii vzťahov.
- Dátové definície - sú priradené ku každému dátovému úložisku. Dátová definícia sa skladá z názvu tabuľky, názvu atribútu, jeho dátového typu, rozsahu hodnôt, ktoré může nadobúdať, implicitných hodnôt a podobne.
- Business pravidlá - tvoria podstatu ETL procesov a každé pravidlo musí byť v nejakom procese zakódované. Väčšina business pravidiel býva implementovaná na úrovni aplikačného kódu, ale niektoré sú obsiahnuté v databázových trigroch alebo procedúrach. Takéto pravidlá musia byť začlenené do logického datového mapovania.
- Metadata úloh - úloha je implementáciou dátového mapovania. Tieto metadata sú veľmi dôležité, pretože obsahujú elementy data lineage v dátovom sklade. Je tu zachytená každá operácia od extrakcie, transformácie, filtrovania až po uloženie. Každá operácia má zachytené všetky vstupy a výstupy.
- Transformačné metadata - každá úloha sa skladá z viacerých častí a niektoré tieto časti sa nazývajú dátové transformácie. Transformáciou sa rozumie akákoľvek manipulácia s datami. Transformačné metadata obsahujú informácie o procese transformácie, spôsobe zmeny dát na iné a podobne.
- Dávkové metadata - dávkové metadata obsahujú informácie o spúšťaní viacerých úloh súčasne. Obsahujú plán spúšťania jednotlivých úloh.

1.2.3 Business metadata

Popisujú význam dát z pohľadu koncových užívateľov.

- Business definície - pridávajú dátam zmysel pre koncových užívateľov. Obsahujú fyzické meno tabuľky a atribútu, business meno atribútu a definíciu, ktorá popisuje význam tohto atribútu.

- Informácie o zdrojových systémoch - poskytujú informácie o tom, ku ktorým tabuľkám zo zdrojového systému sa pristupuje v dátovom sklade. Najdôležitejšie súčasti týchto metadát sú mená zdrojových databáz alebo súborových systémov, špecifikácie tabuliek, business definície, business pravidlá, definície chovania pri nevalidných dátach a podobne.

- Dátový slovník dátového skladu - je to zoznam všetkých dátových položiek použitých v dátovom sklade spolu s ich business definíciou. Položky musia byť presne vyšpecifikované, to znamená, musia obsahovať názov položky, dátový objekt, v ktorom sa položka nachádza, schému, v ktorej sa nachádza dátový objekt a nakoniec aj databázu alebo súborový systém, v ktorom je schéma obsiahnutá.

- Logické dátové mapovanie - skladá sa z mapovania zdroja na cieľ, ktoré logicky presne vysvetľuje, čo sa deje s dátami od momentu, keď sú vyextrahované zo zdrojového systému až do momentu, keď sú uložené do cieľového úložiska. Logické dátové mapovanie je zásadná časť metadát a je používané pre vytváranie funkčných špecifikácií. Sú to najzaujímavejšie metadata dátového skladu a sú známe aj pod názvom data lineage.

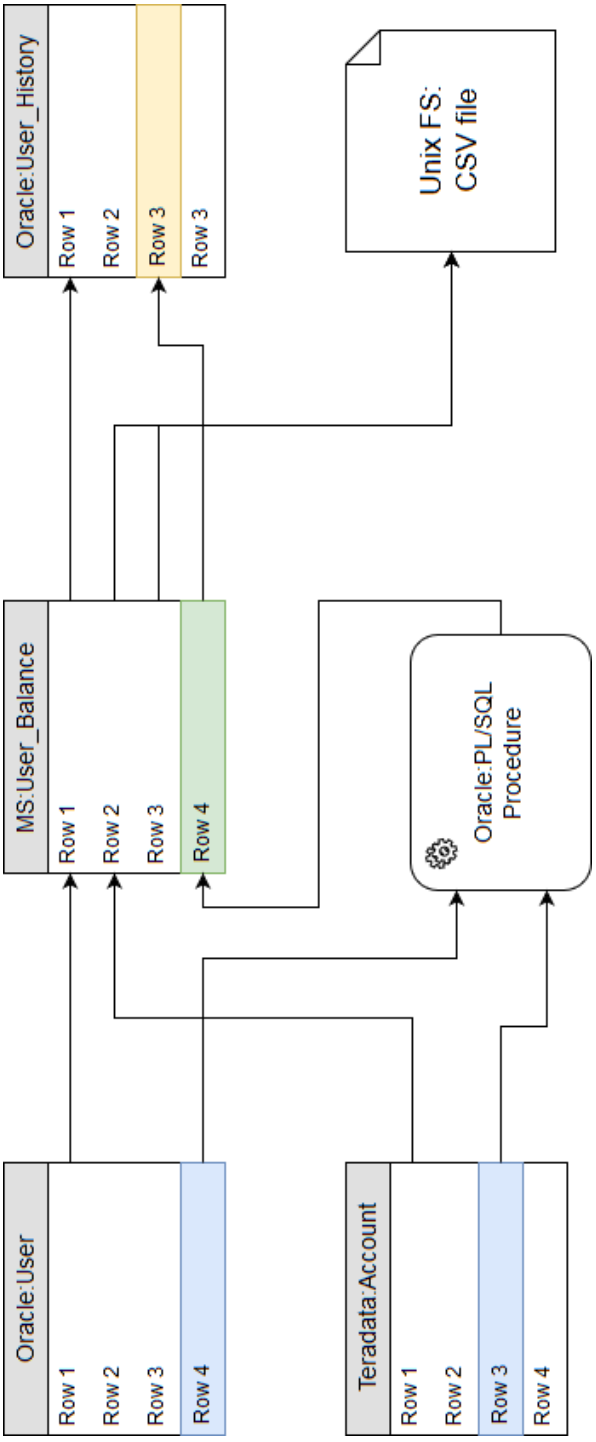
1.3 Data lineage

ETL procesy tvoria najdôležitejšie metadata v dátovom sklade - data lineage. Data lineage sleduje dáta od ich umiestnenia v zdrojovom systéme a presne dokumentuje aké tranformácie sú na nich prevedené pred tým, ako sú uložené do cieľového dátového úložiska. Data lineage tiež zahrňuje definície zdrojového systému a konečného umiestnenia v cieľovom úložisku.

Data lineage pokrýva priame závislosti medzi dátovými atribútmi, ale aj závislosti, ktoré nie sú na prvý pohľad zrejmé a to napríklad, ak nejaký atribút ovplyvňuje transformáciu iných dát nepriamo, napríklad ako súčasť podmienky filtra alebo ako vstupný parameter funkcie.

Na obrázku 1.2 je znázornený príklad data lineage v grafickej podobe ako graf závislostí atribútov jednotlivých dátových objektov. Dátové toky sú znázornené smerovými hranami grafu. Zelenou farbou je vyznačený skúmaný atribút. Žltou farbou je vyznačený cieľový systém a jeho atribút, ktorý je skúmaným atribútom ovplyvnený. Modrou farbou sú vyznačené atribúty v zdrojových sytémoch, ktoré ovplyvňujú (aj nepriamo) skúmaný atribút.

Takéto dátové toky sa využívajú napríklad pri vytvorení analýzy dopadov alebo pri bezpečnostnom audite dátového skladu.



Obr. 1.2: Príklad data lineage

1.3.1 Analýza dopadov

Jedným z dôvodov, prečo je potrebné získavať ETL metadata, je analýza dopadov. Analýza dopadov umožňuje získať zoznam všetkých dátových atribútov v prostredí dátového skladu, ktoré by boli ovplyvnené navrhovanou zmenou. Analýza dopadov musí brať do úvahy všetky funkčnosti ETL procesu, pretože zmeny zdrojových a cieľových dát môžu byť nepretržité a iba ETL proces vie o všetkých zúčastnených dátových elementoch.

Základnou schopnosťou data lineage je v tomto prípade zobrazit dopad zmien, ktoré by nastali v nejakej komponente dátového skladu a získať zoznam všetkých atribútov vo všetkých ostatných komponentách, ktoré by boli touto zmenou ovplyvnené.

Analýza dopadov odpovedá na otázky ako napríklad:

- Ktoré dátové procesy závisia na týchto uložených dátach?
- Je táto tabuľka zo zdrojového systému použitá v dátovom sklade?
- Ovplyvnilo by zmazanie tohto stĺpca v zdrojovom systéme dátový proces?
- Ktoré zdrojové tabuľky plnia túto tabuľku?
- Ktoré procesy a tabuľky v dátovom sklade bude potrebné upraviť, ak sa zmení dátový typ tejto položky?

Odpovede na tieto otázky bez použitia nástrojov pre extrakciu metadát a následné vytvorenie data lineage, by bolo skoro nemožné zodpovedať. Jednou možnosťou by bolo spravovať tabuľky, ktoré by udržiavali údaje o všetkých tabuľkách v dátovom sklade a ich mapovaní medzi sebou (tabuľky metadát), avšak tie by sa museli pri každej zmene v dátovom sklade manuálne upraviť, aby boli aktuálne.

Jedným z takýchto nástrojov, ktoré automatickú extrakciu metadát a následnú vizualizáciu jednotlivých dopadov umožňujú je práve projekt Manta.

1.3.2 Bezpečnostný audit

Ďalším príkladom použitia data lineage je bezpečnostný audit. Veľké organizácie potrebujú mať prehľad o tom, kam sa ich citlivé dáta (napríklad osobné údaje o klientoch) všade dostávajú. V tomto prípade je data lineage ideálnym riešením, pretože po extrakcii metadát a zobrazení dátových tokov je ihneď vidieť, kam sa ktoré dáta dostávajú a je možné prijať bezpečnostné opatrenia.

Projekt Manta je vhodným riešením pre túto situáciu, keďže po extrakcii metadát a vygenerovaní dátových tokov, je možná vizualizácia data lineage a bezpečnostných audit citlivých dát.

1.4 Projekt Manta

Projekt Manta je súbor nástrojov, ktoré umožňujú extrakciu metadát z dátového úložiska (dátového skladu), uloženie metadát, konštrukciu dátových tokov a následnú vizualizáciu data lineage a podporu pre ďalšie analýzy.

Proces spracovania metadát:

- Extrakcia metadát parsovaním ETL skriptov a dátových slovníkov
- Uloženie metadát do metadatového úložiska
- Vizualizácia dátových tokov (data lineage)
- Analýzy nad metadatami (impact analysis)
- Poskytnutie metadát externým systémom pre ďalšie spracovanie

1.4.1 Extrakcia metadát

Pre extrakciu metadát z dátového systému slúžia rôzne parsery. Tieto parsery prehľadávajú zdrojové kódy systémov a vyhľadávajú v nich dátové objekty, akými sú napríklad tabuľky a pohľady a následne transformácie medzi nimi, ako sú napríklad trigre, uložené procedúry a iné databázové skripty.

V súčastnosti sú podporované rôzne technológie, akými sú napríklad Oracle, Teradata, Informatica, IBM DB alebo Microsoft SQL. Vyextrahované metadata následne parser uloží do metadatového úložiska, ktoré je ďalším z nástrojov projektu Manta.

1.4.2 Metadatové úložisko

Metadátové úložisko slúži na ukladanie metadát a pre následné poskytovanie analýz nad nimi. Metadata sú v úložisku uložené v grafovej štruktúre, pretože tá najviac odpovedá vzťahom medzi metadátami. Uzol grafu je dátový objekt, napríklad stĺpec tabuľky a hrana grafu reprezentuje vzťah medzi dátovými objektami, napríklad dátový tok.

Samotná grafová štruktúra je implementovaná pomocou technológie Titan [3], ktorá tvorí platformu pre použitie viacerých druhov databáz a ich podporných technológií. Ako samotné dátové úložisko je použitá technológia Persistit a ako podporná technológia je použitá Lucene, ktorá slúži na indexovanie metadát.

Všetky tieto technológie spolu poskytujú podporu pre dotazovací jazyk Gremlin [4]. Gremlin sa využíva pre vyhľadávanie dát a manipuláciu s dátami v grafových databázach. V budúcnosti by mohol byť bez veľkých zmien nasadený do tohto projektu.

1.4.3 Vizualizácia dátových tokov

Projekt Manta umožňuje zobraziť grafickú reprezentáciu metadát pomocou vykreslenia grafu. Uzly grafu sú zobrazené ako dátové objekty (tabuľky, pohľady), ktoré majú v sebe vnorené ďalšie podobjekty (stĺpce). Dátové toky sú medzi nimi zobrazené ako smerové hrany.

Tento zobrzený graf je interaktívny, čiže umožňuje užívateľovi zakliknutie jednotlivých objektov v grafe a následné zobrazenie podrobností alebo vyfiltrovanie iba požadovaných mapovaní na ostatné objekty a podobne. Graf umožňuje aj fulltextové vyhľadávanie pre rýchlu lokáciu objektu vo veľkých grafoch.

Príklad takéhoto vizualizovaného grafu je zobrazený na obrázku 1.3. Červenou farbou je zvýraznený skúmaný atribút. Žltou farbou je zobrazený atribút v transformačnom skripte a je pripojený smerovou hranou. Zdrojový atribút je zobrazený modrou farbou a je pripojený k transformačnému skriptu smerovou hranou.

Všetky zvýraznené objekty sú podobjektami, iných objektov. Napríklad modrý atribút je stĺpec, ktorý je súčasťou tabuľky. Žltý atribút je zase podobjektom skriptu, pretože je jeho parametrom a samotný skript je podobjektom priečinku, v ktorom je uložený.

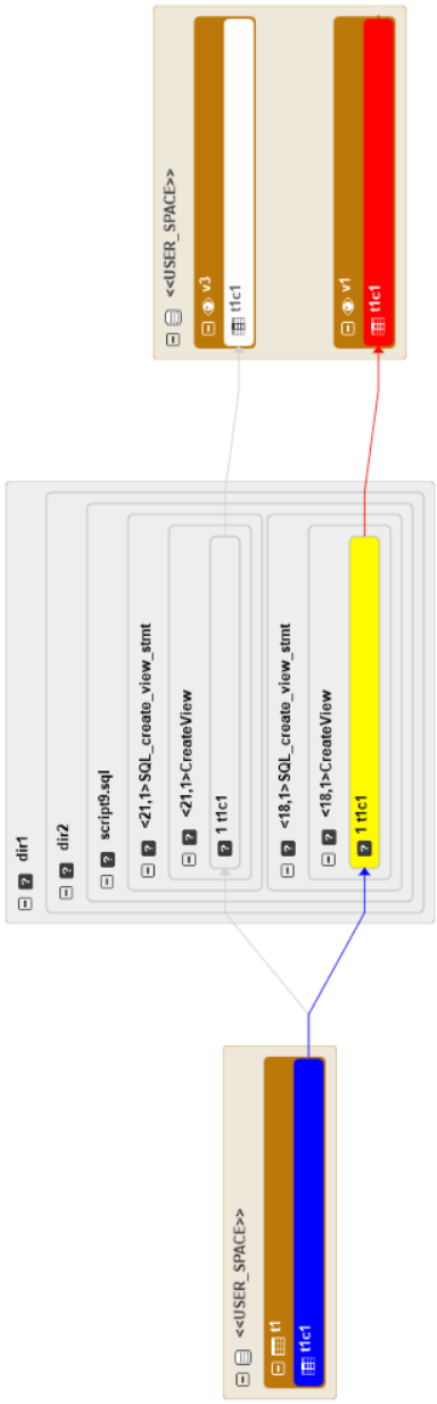
1.4.4 Podpora externých systémov a analýz

Ďalšou požiadavkou na projekt Manta je podporova prepojenia s externými systémami pomocou nejakého rozhrania (API) a možnosť implementovať a spúšťať nad metadátami vlastné analýzy.

Tieto analýzy budú môcť pre prístup k dátam využiť buď poskytnuté rozhranie alebo implementáciu vlastnej analýzy, ktorá sa bude spúšťať priamo na strane úložiska Manta. Takto sa dosiahne lepšia efektivita a výkon. K tomuto účelu bude slúžiť doménovo špecifický jazyk (DSL).

Keďže majú metadata formu grafu, bude potrebné tieto grafy medzi systémami prenášať v serializovanej forme. Projekt Manta bude musieť podporovať aj rôzne spôsoby serializácie grafu.

Všetky tieto funkčnosti sú súčasťou aplikačnej vrstvy, ktorej analýza, návrh a implementácia prototypu bude cieľom tejto práce.



Obr. 1.3: Príklad vizualizácie dátového toku

Kapitola 2

Cieľ práce

Cieľom tejto diplomovej práce je zanalyzovať, navrhnúť a implementovať prototyp aplikačnej vrstvy nad metadatovým úložiskom projektu Manta.

Táto aplikačná vrstva musí umožňovať komunikáciu externých systémov s metadatovým úložiskom a zároveň umožniť týmto systémom spúšťať skripty, ktoré implementujú analýzy nad metadatami na strane Manta úložiska. Dáta, ktoré sú hlavnou zložkou komunikácie majú štruktúru grafu a preto ich musí aplikačná vrstva vhodne serializovať.

2.1 Špecifikácia

Projekt Manta je implementovaný formou webovej aplikácie, ktorá beží na integrovanom serveri Tomcat. Ku komunikácii sa používa protokol HTTP. Preto je nutné, aby bolo komunikačné rozhranie založené na rovnakých technológiách. Komunikačné rohranie musí umožniť komunikáciu pomocou protokolu HTTP a to formou webového REST (API) rozhrania.

Projekt Manta beží na platforme Java (JVM) a preto je nutné, aby bol doménový jazyk (DSL) kompatibilný s touto technológiou, keďže analýzy v ňom implementované budú bežať na serverovej strane. Taktiež je nutné, aby tento jazyk mal ľahko osvojiteľnú a minimalistickú syntax.

Obe tieto súčasti (API aj DSL) musia podporovať atomické operácie nad grafovou databázou s metadatami, import grafu a export grafu.

Pre import a export grafu je nutná podpora serializácie grafu, ako v smere vkladania, čiže parsovanie serializovaného grafu a jeho vytvorenie v databáze a tak aj vygenerovanie serializovanej podoby grafu z podgrafov z databázy a jeho poskytnutie externému systému.

Formát serializácie grafu musí byť vhodne zvolený, aby bol podporovaný externými systémami a pravdepodobne bude nutná podpora viacerých takýchto formátov.

Všetky bussiness a technické požiadavky sú podrobne spracované v nasledujúcej kapitole v sekcii 3.1.

Kapitola 3

Analýza

Táto kapitola rozoberá vlastnosti, ktoré by mala mať požadovaná aplikačná vrstva, porovnaním rôznych technológií a prístupov, ktoré je možné pre tento projekt použiť a ďalej preskúmaním už existujúcich projektov, ktoré by takisto mohli poskytnúť niektoré časti riešenia.

Analýza je pri vývoji softvéru jednou z najdôležitejších častí, avšak veľakrát sa stáva, že je práve tejto časti venovaná malá pozornosť. Cieľom analýzy je zistiť, čo je vlastna potrebné vytvoriť. Ďalšia fáza vývoja, návrh a voľba technológií potom odpovedá na otázku, ako je to možné vytvoriť.

3.1 Požiadavky

Požiadavky obsahujú vlastnosti a funkčnosti, ktoré musí konečné riešenie spĺňať ako z hľadiska businesu, tak aj po technickej stránke.

3.1.1 Bussiness požiadavky

- Aplikačná vrstva musí byť implementovaná platformovo nezávisle, aby nekládla na zákazníka žiadne ďalšie nároky a umožnila nasadenie naprieč všetkými platformami.

- Aplikačná vrstva musí byť navrhnutá formou Maven modulu, aby bolo nasadenie u zákazníka plynulé a nekomplikoval sa tak proces distribúcie projektu Manta.

- Formát serializovaného grafu musí byť textový, pretože projekt využíva na komunikáciu protokol HTTP, ktorý je taktiež textový.

- Formát serializovaného grafu musí mať podporu v externých knižniciach, aby umožnil externým systémom jeho ľahké spracovávanie a generovanie.

- Jazyk DSL musí mať ľahko osvojiteľnú syntax a veľkú robustnosť, aby bolo možné intuitívne a rýchlo implementovať akúkoľvek analýzu nad metadatovým úložiskom.
- Aplikačná vrstva musí poskytovať vysokú úroveň zabezpečenia ako webového rozhrania, tak aj spúšťania jazyka DSL, pretože metadatové úložisko, nad ktorým DSL pracuje, bude často obsahovať citlivé dáta.

3.1.2 Technické požiadavky

- Aplikačná vrstva musí komunikovať pomocou protokolu HTTP, keďže ostatné moduly projektu sú realizované ako webová aplikácia.
- Programové rozhranie musí odtieniť API konzumenta od fyzického modelu úložiska metadát, pretože štruktúra fyzického modelu grafovej databázi sa môže v budúcnosti meniť.
- Implementácia aplikačnej vrstvy musí umožniť zmenu fyzického úložiska v budúcnosti, bez nutnosti upravovať API a to z dôvodu pravdepodobného využitia odlišných grafových databáz v budúcnoti.
- Komunikácia s grafovou databázou musí byť kompatibilná s rozhraním TinkerPop Blueprints. Je to nutné práve preto, aby bola zmena technológie grafovej databáze čo najplynulejšia.
- Formát serializovaného grafu musí byť prúdovo spracovateľný externými systémami, pretože serializované grafy môžu mať obrovské veľkosti a tým by príliš zaťažovali serverové prostriedky.

3.2 Súčasný stav

Súčasný stav analyzuje možnosti a funkčnosti projektu v čase pred začatím návrhu aplikačnej vrstvy. Zistenie súčasného stavu umožňuje začatie analýzy nových častí, ktoré budú využívať funkcie, ktoré už v súčasnosti projekt obsahuje. Poskytuje tak akýsi základ pre návrh novej aplikačnej vrstvy a jej prepojenia na metadatové úložisko.

3.2.1 Architektúra

Projekt Manta sa skladá z viacerých samostatných nástrojov. Nástroj metadatové úložisko má modulárnu architektúru a preto je nutné, aby aj novo vzniknutá aplikačná vrstva mala architektúru modulu, ktorý sa do nástroja vloží.

Moduly sú riešené pomocou utility Maven, ktorá sa používa na správu závislostí Java knižníc. Tento nový modul sa založí ako Maven modul a pridá sa do závislotí samotného projektu. Ten potom pri zostavovaní projektu všetky moduly získa podľa požadovanej verzie.

3.2.2 Dátové úložisko

V súčasnosti je v projekte použitá technológia Titan, čo je platforma pre databázové backendy a podporné technológie. V tomto prípade je použité dátové úložisko Persistit a indexovacia technológia Lucene. Technológia Titan je takzvaná TinkerPop enabled, čo znamená, že je kompatibilná s rozhraním TinkerPop Blueprints

Rozhranie TinkerPop Blueprints [5] je projektom organizácie Apache a umožňuje zjednotiť prístup ku grafovým úložiskám od rôznych distribútorov a rôznym grafovým analytickým nástrojom.

Hlavnou výhodou použitia tohto rozhrania je možnosť použiť rôzne databázové technológie v rôznych situáciach. Niekedy je vhodné využiť in-memory databázu a inokedy zase databázu distribuovanú medzi viaceré stroje. Túto zmenu bez nutnosti upravovať implementáciu softvérovej vrstvy umožňuje práve toto rozhranie.

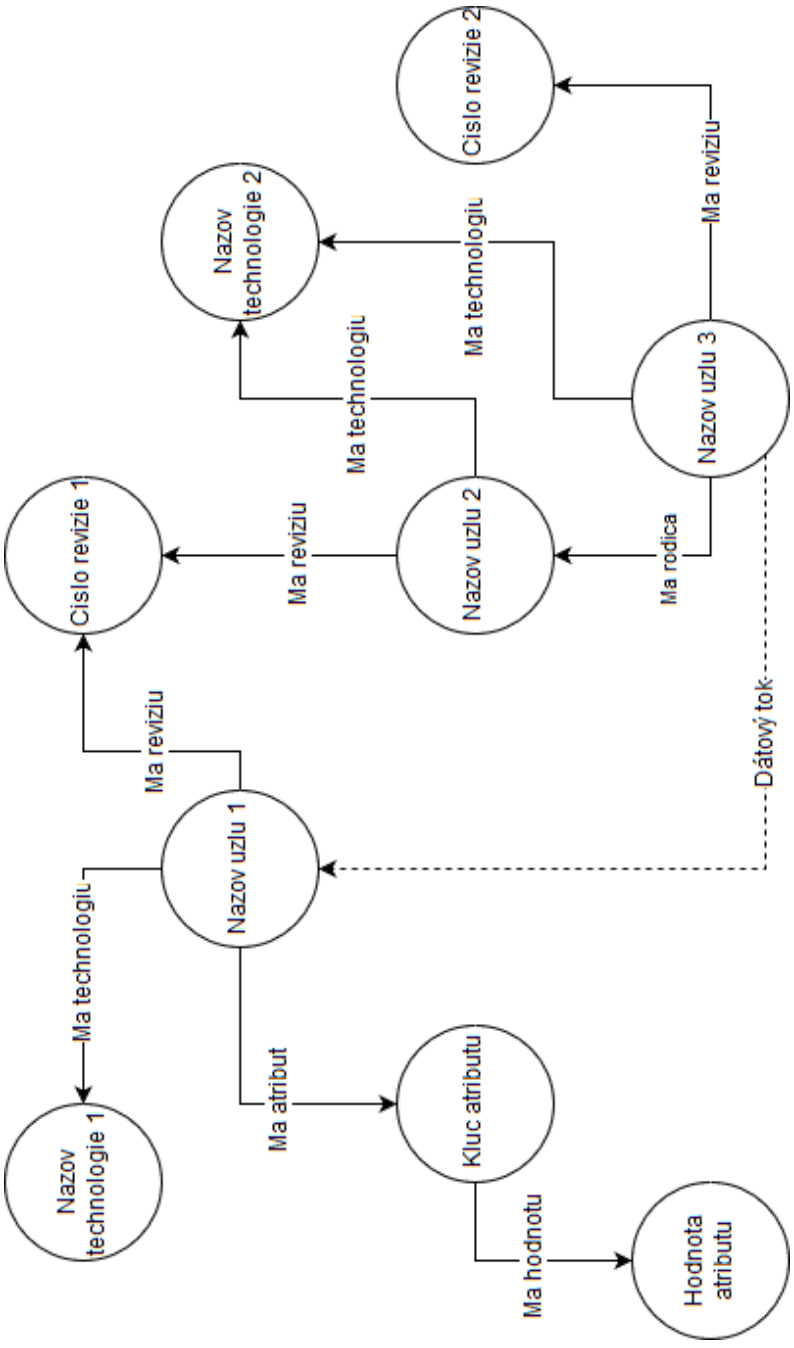
Samotné dátové úložisko je v projekte Manta vstavané, to znamená, že nie je nutné na systém doinštalovávať žiadne ďalšie technológie. Databázový server sa automaticky spustí so spustením serveru Tomcat, ktorý tvorí základ projektu metadatového úložiska Manta a je taktiež v aplikácii vstavaný.

3.2.3 Fyzický model

Fyzický model predstavuje dátovú štruktúru, v akej sú v grafovej databáze uložené všetky dáta. Model je tu v krátkosti popísaný, podrobnejšie informácie sú uvedené v diplomovej práci, ktorá sa týmto modelom zaoberá [6].
Grafová databáza umožňuje pracovať s dvomi typmi objektov a to s vrcholmi a hranami.

- **Vrchol** - reprezentuje dátovú entitu, napríklad osobu, miesto alebo udalosť. Môže obsahovať vlastnosti, ktoré sú realizované ako páry kľúč - hodnota.
  V prípade metadatového úložiska reprezentuje vrchol napríklad tabuľku, stĺpec tabuľky alebo technológiu tabuľky.

- **Hrana** - reprezentuje vzťah medzi dátovými entitami, napríklad osoba pozná inú osobu alebo osoba sa zúčastnila nejakej udalosti na nejakom mieste. Taktiež obsahuje vlastnosti, ktoré sú realizované pomocou páru kľúča a hodnoty.
  V prípade metadatového úložiska reprezentuje vzťahu medzi stĺpcami, čiže dátový tok alebo priradenie tabuľky k technológii.



Obr. 3.1: Príklad fyzického modelu grafu

3.2.4 Entity modelu

Jednotlivé entity modelu sú vo fyzickom modeli realizované kombináciou vrcholov a hrán a ich popiskov. Príklad realizácie takéhoto modelu je zobrazený na obrázku 3.1. Tento model môže byť v rôznych databázach realizovaný rôzne alebo môže dôjsť k jeho zmene a preto musí byť od užívateľa aplikačnej vrstvy odtienený nejakým stálym logickým modelom.

- Koreňový vrchol - tento vrchol musí obsahovať každý databázový model. Všetky ostané vrcholy sú vždy potomkami koreňového vrcholu. Koreňový vrchol tak umožňuje prístup ku všetkým ostatným vrcholom v grafe pomocou prechádzania grafu po hranách.

- Technologický vrchol - tento vrchol slúži na priradenie technológie, z ktorej boli metadata do úložiska získané. Všeobecne platí, že technologické vrcholy sú potomkami koreňového vrcholu a ostatné vrcholy sú potom potomkami technologických vrcholov.
  Pre získanie technológie je potom potrebné prechádzať graf od konkrétneho vrcholu smerom ku koreňovému uzlu a najbližší technologický uzol určí technológiu skúmaného vrcholu.

- Revízny vrchol - tento vrchol obsahuje takzvanú revíziu, čo je forma verzovacieho systému, ktorá je použitá v grafovom modeli. Revízia je číslo, ktoré sa pri každej zmene navyšuje a tak niektoré objekty môžu stratiť platnosť alebo byť naopak pridané do novej revízie. Viac o revízii grafu je popísané v ďalšej časti práce.

- Uzlový vrchol - tento vrchol reprezentuje uzly v data lineage grafe. Teda reprezentuje databázové objekty ako napríklad tabuľky, pohľady alebo transformačné skripty.
  Vlastnosti uzlových vrcholov sú definované pomocou rôznych priradzovacích hrán a iných typov vrcholov.

- Atribútový vrchol - tento vrchol reprezentuje atribút uzlu a to tak, že obsahuje jeho kľúč a ďalšími hranami sú priradené hodnoty atribútu. Je to realizované z dôvodu, že atribút môže obsahovať aj pole hodnôt.

- Hodnotový vrchol - tento vrchol obsahuje samotné hodnoty atribútu uzlu a je s uzlom prepojený priradzovacími hranami cez atribútový vrchol.

- Priradzovacia hrana - (na obrázku 3.1 vyznačená plnou čiarou) táto hrana reprezentuje logické spojenie medzi rôznymi vrcholmi a spolu tak vytvoria entitu logického modelu.
  Typ priradzovacej hrany je určený jej popiskom, napríklad rodičovská hrana prepája vrchol stĺpec s vrcholom tabuľka, do ktorej stĺpec patrí alebo revízna hrana spája vrchol tabuľky s vrcholom revízie a určuje tak revíziu objektu tabuľky.

- Dátová hrana - (na obrázku 3.1 vyznačená prerušovanou čiarou) táto hrana reprezentuje dátový tok z data lineage grafu. Spája vždy dva uzlové vrcholy. Jej typ je taktiež určený popiskom a môže byť priama a filtrovacia.
  Priama hrana zobrazuje priame mapovanie dát a filtrovacia zobrazuje dáta, ktoré sa navzájom ovplyvňujú, ale nie sú na seba priamo mapované (napríklad hodnota podmienky v databázovom skripte).

3.3 Serializácia grafu

Serializácia grafu znamená prenesenie grafu z jeho podoby ako objektu v pamäti do nejakej formy, ktorá zachová stav objektu a pri tom sa dá prenášať medzi procesmi a systémami. Táto časť rozoberá rôzne možnosti formátov pre serializáciu grafu a zaoberá sa aj ich výhodami a nevýhodami.

Serializácia grafov je veľmi obsiahla téma a v tejto časti budú rozobraté len možnosti, ktoré prichádzajú v úvahu pre tento prípad grafu. Napríklad serializácia grafov, ktoré sú obojsmerné je o niečo náročnejšia ako serializácia binárnych stromových grafov, ktoré majú jasne danú štruktúru.

Analýza serializácie grafu počíta grafom, ktorý má jeden koreňový uzol a skladá sa z listových uzlov a jednosmerných hrán.

Serializácia sa podľa kódovania výstupu dá rozdeliť na:

- Binárna - výstupný formát serializácie má binárnu formu. Grafová štruktúra je reprezentovaná pomocou postupnosti bajtov. Výhodou je vysoká kompaktnosť a efektívne strojové spracovanie, avšak tento druh výstupu nie je pre človeka čitateľný.

- Textová - výstupný formát serializácie má textovú formu. Grafová štruktúra je reprezentovaná pomocou znakov, ktoré môžu znamenať hodnotu alebo mať špeciálny význam. Táto forma je pre človeka dobre zrozumiteľná, ale nie je tak kompaktná ako binárna reprezentácia.
  V dnešnej dobe nie je problém s kapacitou úložných zariadení ani rýchlosťou prenosu a preto to, že má textová reprezentácia väčšiu veľkosť už nehrá tak podstatnú úlohu.

Serializácia sa podľa zanorenia elementov [7] dá rozdeliť na:

- Sekvenčná - pri tomto druhu reprezentácie sú jednotlivé elementy grafu serializované po jednom a sú zoradené do zoznamu za sebou. Pre poradie je dôležité len to, aby bol element, ktorý odkazuje na nejaký iný element, serializovaný až po tom odkazovanom elemente. Inak môžu nastať problémy pri deserializácii.
  Pri grafe to znamená to, že sa vždy najprv serializujú všetky uzly a až potom nasledujú atribúty a hrany. Zaručí sa tak, že hrany budú mať vždy na čo odkazovať.

- Vnorená - pri tomto druhu reprezentácie jednotlivé elementy nenasledujú po sebe, ale sú do seba vnorené tak, že vlastne vytvárajú stromovú štruktúru a kopírujú svojím usporiadaním samotný graf.
  Pri serializácii sa vždy musí začať od koreňového elementu a jeho rekurzívnou serializáciu sa serializuje celý graf. Pri deserializácii je nutné načítať všetok obsah formátu do pamäte a tak prúdové spracovanie nie je možné.

Pre serializáciu grafu bude použitá niektorá z externých knižníc, aby bola poskytnutá dobrá podpora externým systémom. Nie je potrebné implementovať niečo znova od začiatku, ale serializáciu postaviť na niektorom z už existujúcich formátov.

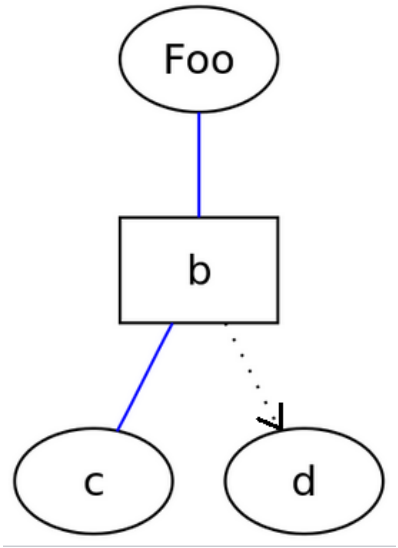
Takisto nebude analýza obsahovať rozbor binárnych formátov pre serializáciu, keďže projekt Manta je realizovaný ako webová aplikácia, ktorá komunikuje cez protokol HTTP. Tento protokol je textový a preto musí byť aj formát serializácie textový a binárne formáty nebudú uvažované.

Textové formáty uvažované pre serializáciu grafu:

- XML - je to jednoduchý a flexibilný formát, ktorý je veľmi rozšírený a používa sa na výmenu obrovského množstva dát, hlavne v rozsiahlych informačných systémoch veľkých korporácií. XML je priemyselný štandard [8] a je preto veľmi podporovaný.
  Pri serializácii grafu by bolo možné využiť sekvenčný prístup aj prístup vnorení. Tento formát pôvodne nebol určený na serializáciu grafu, ale pre svoju všetstrannosť umožňuje serializovať akýkoľvek objekt. V Jave sa formát XML používa často, hlavne pre ukladanie objektov na súborový systém.

- GraphML - tento formát je založený na formáte XML. Nevyužíva žiadnu špeciálnu syntax [9] a preto je jeho podpora bezproblémová.
  Umožňuje serializovať rôzne druhy grafov, napríklad hypergrafy alebo hierarchické grafy a pridávať rôzne atribúty, ktoré sú špecifické pre aplikáciu.
  Pre prípad serializácie grafov v tomto projekte sú možnosti GraphML viac ako dostačujúce.

- DOT - je to jednoduchý textový formát, ktorý sa používa na popis grafov v balíčku softvéru Graphviz [10]. Je podporovaný viacerými nástrojmi, avšak všetky na spracovanie formátu využívajú knižnice z Graphvizu.
  Formát používa na zachytenie štruktúty grafu textové identifikátory uzlov, ktoré sú spájané pomocou pomlčiek a znakov väčšie a menšie, ktoré znázorňujú bezsmerové a smerové hrany. V nasledujúcom kóde 3.3 je znázornená serializácia grafu zobrazeného na obrázku 3.2 v jazyku dot.



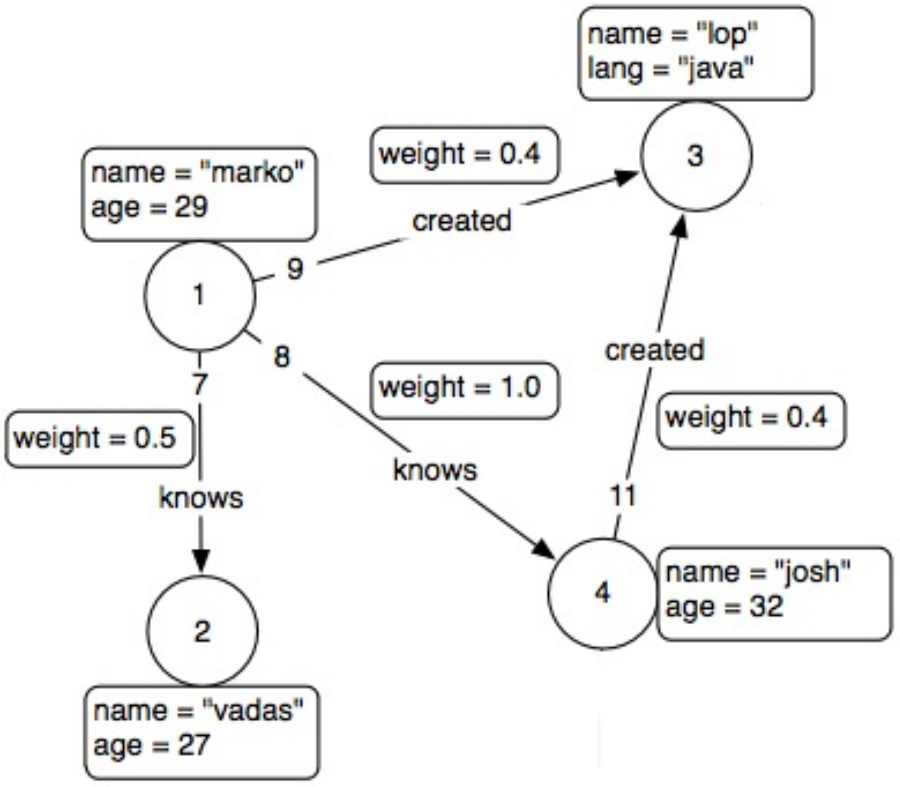
Obr. 3.2: Príklad grafu pre serializáciu v jazyku DOT

```
//samotny graf
graph graphname {
     //atribut grafu
     size="1,1";
     //atribut uzlu
     a [label="Foo"];
     //atribut uzlu
     b [shape=box];
     //struktura grafu pomocou bezsmerovych hran
     a -- b -- c [color=blue]; //popisok hrany
     //struktura grafu pomocou smerovej hrany
     b -> d [style=dotted]; //popisok hrany
}
```

Obr. 3.3: Príklad serializovaného grafu v jazyku DOT

- JSON - JavaScript Object Notation [11] je formát na výmenu dát, ktorý je pre človeka veľmi ľahko čitateľný. Tento formát je priemyselný štandard, ktorý vyvinula organizácia ECMA International. Momentálne patrí k najviac využívaným formátom na výmenu dát po webe.
  JSON rovnako ako XML umožňuje vďaka svojej flexibilite serializovať akýkoľvek objekt a je podporovaný v podstate každou technológiou.
- GraphSON - tento formát je založený na formáte JSON, avšak je prispôsobený na použitie pri serializácii grafov. GraphSON je vyvíjaný organizáciou TinkerPop, ktorej rozhrania sú používané naprieč celým projektom Manta.

GraphSON serializuje graf sekvenčne a to ako zoznam uzlov, ktoré obsahujú aj svoje atribúty a zoznam hrán, ktoré taktiež obsahujú svoje atribúty. Tento formát je tak prúdovo spracovateľný a má podporu v knižniciach od organizácie TinkerPop.



Obr. 3.4: Príklad komplexného grafu pre serializáciu v GraphSON

V nasledujúcom kóde je zobrazená serializácia grafu z obrázku 3.4 do formátu GraphSON.

```
{
    "graph": {
        "mode":"NORMAL",
        "vertices": [
            {
                "name": "lop",
                "lang": "java",
                "_id": "3",
                "_type": "vertex"
            },
```

```
            {
                "name": "vadas",
                "age": 27,
                "_id": "2",
                "_type": "vertex"
            },
            {
                "name": "marko",
                "age": 29,
                "_id": "1",
                "_type": "vertex"
            },
            {
                "name": "josh",
                "age": 32,
                "_id": "4",
                "_type": "vertex"
            } ],"edges": [
            {
                "weight": 0.5,
                "_id": "7",
                "_type": "edge",
                "_outV": "1",
                "_inV": "2",
                "_label": "knows"
            },
            {
                "weight": 0.4000000059604645,
                "_id": "9",
                "_type": "edge",
                "_outV": "1",
                "_inV": "3",
                "_label": "created"
            },
            {
                "weight": 1,
                "_id": "8",
                "_type": "edge",
                "_outV": "1",
                "_inV": "4",
                "_label": "knows"
            },
            {
                "weight": 0.4000000059604645,
                "_id": "11",
                "_type": "edge",
                "_outV": "4",
                "_inV": "3",
                "_label": "created"
            }
        ]
    }
}
```

- CSV - tento formát využíva pre účely ukladania údajov textové hodnoty, ktoré sú oddelené čiarkou (preto comma-separated values). Každý riadok v súbore tvorí samostatný záznam a preto je CSV veľmi dobre spracovateľné prúdovým prístupom. Tento formát patrí k najrozšírenejším a najpodporovanejším formátom vôbec.
  Pre účely serializácie grafu, sa každý riadok v súbore CSV považuje za samostatný grafový objekt. Jeden riadok reprezentuje napríklad jeden uzol grafu alebo atribút uzlu, či dátový tok. Objekty (čiže riadky) musia dodržiavať správne poradie, pretože medzi nimi môžu existovať závislosti, napríklad riadok s atribútom nemůže byť v súbore uvedený skôr ako riadok s jeho priradeným uzlom.
  Tento formát je štandardizovaný [12] a požadovaný užívateľmi aplikačnej vrstvy. Ukážka serializácie grafu do formátu CSV je na obrázku 3.5.

Keďže grafy môžu mať obrovskú veľkosť a kvôli splneniu podmienky na prúdové spracovanie súborov so serializovanými grafmi, budú v návrhu použité textové formáty so sekvenčným uložením grafu. Medzi ne patrí napríklad GraphSON a CSV, ktorých dôvody použitia sú ďalej rozpísané v kapitole o návrhu aplikačnej vrstvy.

3.4 Aplikačné programové rozhranie

Táto časť sa zaoberá analýzou rozhrania, ktoré bude slúžiť pre komunikáciu medzi metadatovým úložiskom Manta a externými systémami. Aplikačné programové rozhranie musí spĺňať sériu požiadaviek ako je napríklad komunikácia cez protokol HTTP alebo odtienenie užívateľa od reálneho fyzického modelu databázy a možnosť výmeny samotnej grafovej databázy.

Najčastejšími prístupmi k návrhu rozhrania po protokole HTTP sú REST a SOAP. Prístup SOAP využíva pre komunkáciu SOAP obálky, čo sú vlastné XML správy, ktoré musia dodržiavať konkrétny predefinovaný formát. Tento prístup sa hodí skôr pre korporátne systémy s middleware zbernicou. Pre riešenie rozhrania tohto projektu bude uvažované len REST rozhranie.

3.4.1 REST

Pojem REST slúži pre označenie spôsobu poskytovania webových služieb, pomocou predefinovaných zdrojov a operácií nad nimi. Tieto operácie sú bezstavové a poskytujú textovú reprezentáciu entity. Typicky používa rozhranie REST pre komunikáciu protokol HTTP. Jednotlivé operácie sú mapované pomocou webovej adresy (URL) a HTTP metódy.

```
resource,0,BTEQ scripts,BTEQ,BTEQ
resource,1,Teradata,Teradata,Teradata
node,18,,<<USER_SPACE>>,Database,1
node,19,18,v4,View,1
node,20,19,t2c2,Column,1
node,25,18,v3,View,1
node,26,25,t1c2,Column,1
node,27,25,t1c1,Column,1
node,28,4,<20,1>SQL_create_view_stmt,BTEQ Statement,0
node,29,28,<20,1>CreateView,BTEQ CreateView,0
node,30,29,2 t2c2,BTEQ ColumnFlow,0
node,31,29,1 t1c1,BTEQ ColumnFlow,0
node,32,18,v2,View,1
node,33,32,t2c2,Column,1
node,34,32,t1c1,Column,1
node,40,18,v1,View,1
node,41,40,t1c3,Column,1
node,42,40,t1c2,Column,1
node,43,40,t1c1,Column,1
node,44,18,t3,Table,1
node,45,44,t3c3,Column,1
node,46,44,t3c2,Column,1
node,47,44,t3c1,Column,1
node_attribute,19,TABLE_TYPE,VIEW
node_attribute,25,TABLE_TYPE,VIEW
node_attribute,32,TABLE_TYPE,VIEW
node_attribute,40,TABLE_TYPE,VIEW
edge,0,39,43,DIRECT,0
edge,1,38,42,DIRECT,0
edge,2,37,41,DIRECT,0
edge,3,31,34,DIRECT,0
edge,4,30,33,DIRECT,0
edge,5,24,27,DIRECT,0
edge,6,23,26,DIRECT,0
```

Obr. 3.5: Príklad serializovaného grafu vo formáte CSV

Najčastejšími formátmi, ktoré REST API pre komunikáciu využíva sú XML, JSON, HTML alebo ďalšie predefinované textové formáty. V tomto prípade je potrebné zvoliť formát, ktorý umožní jednoduchú serializáciu samostatných grafových objektov, napríklad uzlu alebo hrany do textovej podoby.

Zdroje musia byť navrhnuté tak, aby umožnili dotazovanie sa nad grafovou štruktúrou a zároveň boli navrhnuté tak, aby umožnili jednoduché a intuitívne používanie. Operácie nad zdrojmi musia byť atomické a nespolihať sa na stav zadaný užívateľom na vstupe. Rozhranie REST je typicky bezstavové a doplnenie používania stavov je možné pomocou princípu HATEOAS.

3.4.2 HATEOAS

Princíp HATEOAS (Hypermedia As The Engine Of The Application State) znamená využitie hypertextových linkov pre reprezentáciu stavu v RESTových rozhraniach.

V tomto prípade by to znamenalo napríklad doplnenie linkov do objektu pre revíziu, pomocou ktorých by sa volali ďalšie operácie možné so stavom tejto konkrétnej revízie a to napríklad by pre otvorenú revíziu bol doplnený link pre jej uloženie.

3.4.3 Bezpečnosť

Metadatové úložisko častokrát obsahuje citlivé informácie o dátových skladoch užívateľov. Webové rozhranie predstavuje časť projektu Manta, ktorá bude vystavená pre použitie externými systémami a preto je zaistenie bezpečnosti kritickým bodom analýzy. Bezpečnosť webového rozhrania znamená zabezpečiť to, aby sa k dátam dostali len používatelia, ktorý na to majú práva.

Medzi základné požiadavky na bezpečnosť patrí nakonfigurovanie užívateľa, ktorý bude mať k rozhraniu prístup. Prístup nemusí byť rozdelený pre viaceré role podľa zdrojov ani operácií nad nimi. To znamená, že užívateľ, ktorý má mať prístup k webovému rozhraniu, bude mať prístup k všetkým jeho súčastiam a operáciám. To návrh zabezpečenia značne zjednodušuje.

Ďalšou užívateľskou požiadavkou na bezpečnosť je možnosť integrácie s externým autentizačným serverom, napríklad LDAP. Bezstavové získanie prístupu takýmto spôsobom vyžaduje komunikáciu s autorizačným serverom (napríklad open source server Zool, vyvíjaný na FIT ČVUT [13]) pri každej požiadavke API užívateľa, čo by značne znížilo výkon a spomalilo komunikáciu a preto bude potrebné využiť čiastočne stavové riešenie, aby bola požiadavka na bezpečnostný server zasielaná len raz.

3.4.4 Atomické operácie

Atomické operácie znamenajú získavanie konkrétnych údajov o REST zdrojoch alebo ich zmena pomocou HTTP operácií nad týmito zdrojmi.
Atomické operácie budú rozdelené podľa entít na:

1. Operácie s revíziou
2. Operácie s technológiou
3. Operácie s uzlom
4. Operácie s atribútom
5. Operácie s dátovým tokom

3.4.4.1 Revízia

Predpoklady na reprezentáciu revízie sú, že je reprezentovaná číslom a vo fyzickom modeli je reprezentovaná samostaným vrcholom. Toto ale musí API vedieť od užívateľa odtieniť. V logickom modeli má každý objekt grafu platnosť v určitom intervale revízií alebo v jednej konkrétnej revízii.

Operácie musia umožniť získavať číslo poslednej revízie a jej stav. Revízia sa môže nachádzať v otvorenom stave alebo v uloženom stave. Medzi ďalšie požiadavky patrí vytvorenie novej otvorenej revízie, uloženie otvorenej revízie (commit) a vrátenie otvorenej revízie (rollback).

Kvôli zjednodušeniu používania bude API pracovať vždy nad jednou konkrétnou revíziou. Samotné objekty, s ktorými bude pracovať nemusia patriť presne len do tejto revízie, ale bude stačiť, že sa táto revízia bude nachádzať v ich platnom intervale.

Revízia sa bude musieť zadávať pri každej požiadavke na API. Spôsobov ako dostať číslo revízie do rozhrania je viac. Patrí medzi ne napríklad vytvorenie trvalého spojenia, ktoré si bude revíziu pamätať, ale toto riešenie by nespĺňalo princíp REST bezstavovosti. Medzi ďalšie riešenia patrí posielanie čísla revízie pri každej požiadavke a to buď v tele požiadavky (čo nie je možné pri HTTP metóde GET) alebo ako parameter URL adresy zdroja.

3.4.4.2 Technológia

Technológia je vo fyzickom grafe reprezentované pomocou vrcholu. API vrstva pre túto entitu musí vytvoriť samostatný zdroj, na ktorý sa bude možné dotazovať. Operácie nad technológiou budú musieť byť schopné nájsť detailné informácie ako názov a popis konkrétnej technológie a umožniť priradiť technológiu k ostatným entitám grafu.

Technológia môže byť priradzovaná k ostatným entitám grafu pomocou názvu, avšak neexistuje žiadny pevný číselník týchto názvov a detailov k nim priradených, ďalej pomocou linkov (HATEOAS) alebo pomocou nejakého unikátneho identifikátora.

3.4.4.3 Uzol

Uzol je reprezentácia dátabázového objektu a vo fyzickom modeli je realizovaný pomocou uzlového vrcholu. Obsahuje informácie ako sú názov, typ, technológia a interval revízií, v ktorom má objekt platnosť. Vnútorne obsahuje každý vrchol aj svoje interné identifikačné číslo.

Uzol sa musí dať jednoznačne identifikovať, aby sa bolo možné naň cez API dotazovať. Identifikácia uzla je možná dvoma spôsobmi a to podľa interného ID, ktoré obsahuje každý vrchol v databáze alebo pomocou kompletnej cesty v grafe od koreňového uzlu.

Identifikácia podľa interného ID je značne jednoduchšia a pri dotazovaní sa prenáša menej dát. Má to avšak nevýhody v tom, že sa interné ID môže medzi

jednotlivými revíziami meniť a tak po čase stratí platnosť. Navyše, užívateľ API pri prvom dotaze nevie hodnotu vnútorného ID.

Druhý spôsob využíva cestu v grafe pre jednoznačnú identifikáciu uzla. Tento spůsob je spoľahlivý aj naprieč revíziami a typicky bude používateľ API vedieť cestu k uzlu už dopredu. nevýhodou je veľké množstvo prenášaných dát a preto nemožnosť využiť HTTP metódy GET, pretože cestu grafom nie je možné zadať ako parameter URL.

Atomické operácie budú musieť umožniť nájsť konkrétny uzol a získať jeho rodiča, zoznam potomkov, atribúty uzla, technológiu uzla, overiť jeho revíziu a získať zoznam dátových tokov, ktorých je súčasťou. Pri vyhľadávaní uzla podľa jeho názvu bude potrebné umožniť filtráciu aj podľa ďalších vlastností, kvôli zamedzeniu zbytočne veľkých výsledkov.

3.4.4.4 Atribút

Atribút je vo fyzickom úložisku implementovaný pomocou samostatných atribútových vrcholov, ktoré držia kľúč atribútu a jeho hodnoty, avšak v API musí byť tento detail od užívateľa odtienený. Atribút je vždy závislý na svojom vlastníckom uzle.

Atribúty sa dajú reprezentovať ako zoznam vo vnútri konkrétneho objektu uzla, ďalej vytvorením samostatného zdroja pre atribúty a ďalšou možnosťou je vytvoriť podzdroj k zdroju uzla. Prvá možnosť znamená prenášanie informácii o všetkých atribútoch uzla pri každej odpovedi zdroja uzla.

Atribút musí mať minimálne jednu hodnotu, ale môže ich mať aj viacero. Každý uzol má pridelené ľubovoľné množstvo atribútov. Atribúty musí byť možné získavať podľa vlastníckeho uzla a filtrovať podľa kľúča.

3.4.4.5 Dátový tok

Dátový tok je v grafe reprezentovaný pomocou smerových hrán, ktoré majú popisky a rovnako ako uzly, platnosť v určitom intervale revízií. Vo fyzickom modeli můžu byť realizované pomocou viacerých hrán a vrcholov, to ale v logickom modeli nesmie byť vidieť. Dátový tok je vždy závislý na dvoch uzloch.

Dátový tok sa dá v logickom modeli reprezentovať pomocou jedného uzla a smeru, ďalej pomocou dvoch uzlov alebo pomocou dvoch uzlov a smeru. Popisok musia obsahovať všetky možnosti reprezentácie. Posledná možnosť obsahuje redudantné informácie. Dátové toky musí byť možné získavať podľa uzlov a filtrovať podľa smerov a popiskov.

3.4.5 Import grafu

Aplikačná vrstva musí poskytovať možnosť importovať graf zo serializovaného formátu. Súbor s textovou reprezentáciou grafu musí byť možné zadať pomocou webového rozhrania. Pri importe sa musí zohľadniť aj revízia, v ktorej bude nový graf platný.

Import grafu musí kontrolovať chybné formáty súborov alebo nesprávne poradie záznamov a v prípade chyby import neumožniť. Vo výsledku môžu byť do databázy naimportované iba validné grafy.

Samotný algoritmus importu musí byť výkonný a efektívny, pretože serializované grafy môžu mať obrovské rozsahy a preto je nutné používať prúdové spracovávanie importovaných súborov.

3.4.6 Export grafu

Ďalšou požadovanou funkcionalitou na aplikačnú vrstvu je export grafu do serializovanej podoby. Táto funkcionalita je veľmi komplexná a vyžaduje nejaký nástroj, ktorým by sa dala, čo najjednoduchšie vyšpecifikovať požadovaná časť grafu pre export.

Vyexportovaná časť grafu musí spĺňať niektorý z formátov, ktoré sa použijú pre serializáciu grafu a pre import. Je nutné, aby bol export čo najrýchlejší a najefektívnejší. Preto bol ako nástroj pre vyšpecifikovanie presného grafu pre export vybratý doménovo špecifický jazyk. Ten pri dostatočnej komplexnosti zaručí neobmedzené možnosti exportu a analýz nad grafom.

Doménovo špecifický jazyk beží na strane serveru, takže sa predchádza zbytočnej komunikácii medzi používateľom a serverovým systémom počas procesu exportu. Podrobnejšia analýza možností vytvorenia takéhoto jazyka je rozobratá v nasledujúcej sekcii.

3.5 Doménovo špecifický jazyk

Doménovo špecifický jazyk je skriptovací jazyk, ktorý je prispôsobený pre implementáciu logiky nad určitou doménou (produktom, databázou) a umožňuje tak čo najrýchlejšiu a najjednoduchšiu možnosť rozširovania funkčností domény.

V prípade tohto projektu musí byť doménovo špecifický jazyk prispôsobený dotazovaniu sa nad grafovou databázou, avšak zároveň umožniť implementáciu akejkoľvek analýzy nad výsledkami týchto dotazov a následný export týchto výsledkov do serializovanej podoby.

Dôvodom rozšírenia aplikačnej vrstvy o doménovo špecifický jazyk je to, že užívateľ API bude mať možnosť implemetnovať komplexnú analýzu v tomto jazyku a na server poslať len konečný skript, ktorý potom vráti výsledky analýzy späť užívateľovi. Bez tejto funkčnosti by musel užívateľ všetky dáta získavať postupne pomocou API, čo by bolo v niektorých prípadoch veľmi neefektívne.

Doménovo špecifický jazyk môže byť vytvorený viacerými spôsobmi a podľa využitia prostredia sa DSL dajú rozdeliť na externé a interné [14].

3.5.1 Externé DSL

Externé doménovo špecifické jazyky sú nezávislé na prostredí, v ktorom budú používané, avšak vyžadujú vlastné závislosti, ktoré sú za účelom ich implementácie použité. Návrh a implementácia externého DSL je zložitá a náročná úloha a pre tento projekt nie sú možnosti externého DSL potrebné a preto sa bude analýza ďalej zaoberať internými DSL bežiacimi na prostredí JVM.

3.5.2 Interné DSL

Interné doménovo špecifické jazyky využívajú pre svoj beh funkčnosti hostiteľského jazyka. Nevyžadujú preto žiadne dodatočné prostredie ani závislosti, avšak ich rozsah je obmedzený rozsahom hostiteľského prostredia a väčšinou je ním inšpirovaná aj syntax takéhoto jazyka.

V prípade tohto projektu je hostiteľským jazykom Java a hostiteľským prostredím Java Virtual Machine a preto by v prípade použitia interného doménovo špecifického jazyka musel tento jazyk bežať na JVM. Táto platforma je široko podporovaná a preto je táto možnosť veľmi výhodná.

3.5.3 Návrhové vzory DSL

Možností ako vytvoriť DSL je mnoho a v nasledujúcich sekciách budú popísané najčastejšie postupy, tak ako ich uvádza vo svojej knihe [15] expert na doménovo špecifické jazyky Martin Fowler.

3.5.3.1 Zreťazené metódy

Tento postup je založený na vytvorení DSL objektov, ktoré budú umožňovať volať metódy nasledujúce po sebe bez toho, aby bolo nutné na doménové objekty vytvárať referencie. Každý DSL objekt potom dovoľuje volať len tie metódy, ktoré sú v danom momente dostupné podľa stavu doménového objektu. Ukážka takéhoto DSL pre tvorbu jednoduchého grafu je zobrazená v nasledujúcej ukážke.

```
1 Graph()
2     .edge()
3         .from("a")
4         .to("b")
5         .weight(12.3)
6     .edge()
7         .from("b")
8         .to("c")
9         .weight(10.5)
```

3.5.3.2 Vnorené funkcie

Tento postup využíva vnorovanie funkcií pre dosiahnutie plynulého rozhrania jazyka. Výhodou tohto postupu je jeho hierarchická štruktúra a vypustenie nutnosti vytvárania dodatočných DSL objektov, ktoré uchovávajú momentálny stav. Ukážka tohto postupu pre tvorbu jednoduchého grafu je zobrazená v nasledujúcej ukážke.

```
Graph (
    edge(from("a"), to("b"), weight(12.3),
    edge(from("b"), to("c"), weight(10.5)
);
```

3.5.3.3 Lambda výrazy

Tento postup je založený na využívaní Lambda výrazov ako interného jazyka DSL. Lambda výrazy musia byť podporované hostiteľským jazykom. Ukážka tohto postupu pre vytvorenie jednoduchého grafu je zobrazená v nasledujúcej ukážke.

```
Graph(g -> {
    g.edge( e -> {
        e.from("a");
        e.to("b");
        e.weight(12.3);
    });
    g.edge( e -> {
        e.from("b");
        e.to("c");
        e.weight(10.5);
    });
})
```

3.5.4 Bezpečnosť

Bezpečnosť pri doménovo špecifickom jazyku znamená to, že skripty, ktoré budú implementované pomocou tohto jazyka a spúšťané na strane serveru, nebudú môcť spôsobiť žiadne ohrozenie behu systému ani kompromitáciu citlivých dát. Zabezpečiť tieto požiadavky pri návrhu doménového jazyka je značne zložité.

Prvým dôležitým požiadavkom je zamedzenie zahltenia serverových zdrojov spustením DSL skriptu. Môže sa tak stať napríklad spustením nekonečného cyklu alebo množením procesov (takzvaná fork bomba). Čas a pamäťové zdroje skriptu musia byť v konečnom riešení obmedzené na nejakú rozumnú úroveň.

Druhou požiadavkou je zabezpečiť to, aby sa pomocou DSL skriptov nedalo dostať k iným dátam serveru než tým, pre ktoré je doménovo špecifický jazyk určený. Znamená to, že DSL musí bežať v nejakom oddelenom prostredí, odkiaľ bude mať prístup len k určenej grafovej databáze a nikam inam.

Kapitola 4

Návrh

Táto kapitola sa zaoberá výberom vhodných technológií na základe rešerší spracovaných v predchádzajúcej kapitole. V tejto fáze vývoja aplikácie sú dôležité technologické znalosti a ich správne využitie na dosiahnutie všetkých funkčností, ktoré boli vytýčené v predchádzajúcej kapitole.

4.1 Logický model

Za účelom odtienenia používateľa rozhrania od detailov fyzickej implementácie databázového modelu bude použitý logický model (obrázok 4.1). Aplikačná vrstva bude poskytovať všetky služby tak, akoby sa prevádzali na tomto logickom modeli.

Logický model sa bude skladať z viacero entít, ktoré sú síce vo fyzickom modeli implementované ako viacero vrcholov a hrán, ale navonok sa budú chovať ako jeden atomický prvok. Popis jednotlivých entít je uvedený v ďalšej sekcii.

4.1.1 Entity logického modelu

- Technológia - bude vytvorená ako samostatná entita, ktorá bude obsahovať názov, typ a popis. Každý uzol musí mať priradenú technológiu a priradenie bude využívať ID technológie, ktoré bude súčasťou údajov o uzle. Na technológiu sa preto bude dať dotazovať pomocou ID. Vo fyzickom modeli je technológia tvorená vrcholom, ktorý je potomkom koreňového vrcholu a predkom všetkých uzlov, ktoré majú túto technológiu priradenú.

- Revízia - bude vytvorená ako samostatná entita, ktorá bude obsahovať číslo revízie a stav. Revízia sa bude môcť nachádzať v otvorenom alebo uloženom (commit) stave. Z otvoreného stavu sa bude dať vrátiť (rollback) alebo uložiť (commit).

Každý uzol bude mať platnosť v konkrétnom intervale revízií, ktorý sa bude overovať pri každej atomickej operácii. Bude sa dať dotazovať len na poslednú revíziu, ktorú bude možné uložiť alebo vrátiť. Bude možné vytvoriť novú otvorenú revíziu, ktorá sa stane poslednou.

- Uzol - bude vytvorený ako samostatná entita a bude tvoriť základ pre dotazovanie sa nad grafom v logickom modeli. Bude obsahovať interné ID, názov, typ, technológiu a svojho predka. Technológia aj uzol predka budú k tomuto uzlu priradené pomocou interných ID. Tak sa dosiahne minimálne množstvo prenášanej informácie a umožní sa ďalšie dotazovanie na podrobnosti o technológii alebo rodičovi uzla.

- Atribút - bude vytvorený ako entita, ktorá je závislá na entite uzla. Nebude možné sa dotazovať priamo na atribút, ale vždy bude dotaz relatívny k uzlu. Možné bude získať všetky atribúty uzla alebo jeden atribút podľa kľúča. Atribút sa bude skladať z kľúča a poľa hodnôt. Vo fyzickom modeli je atribút realizovaný pomocou viacerých vrcholov a hrán.

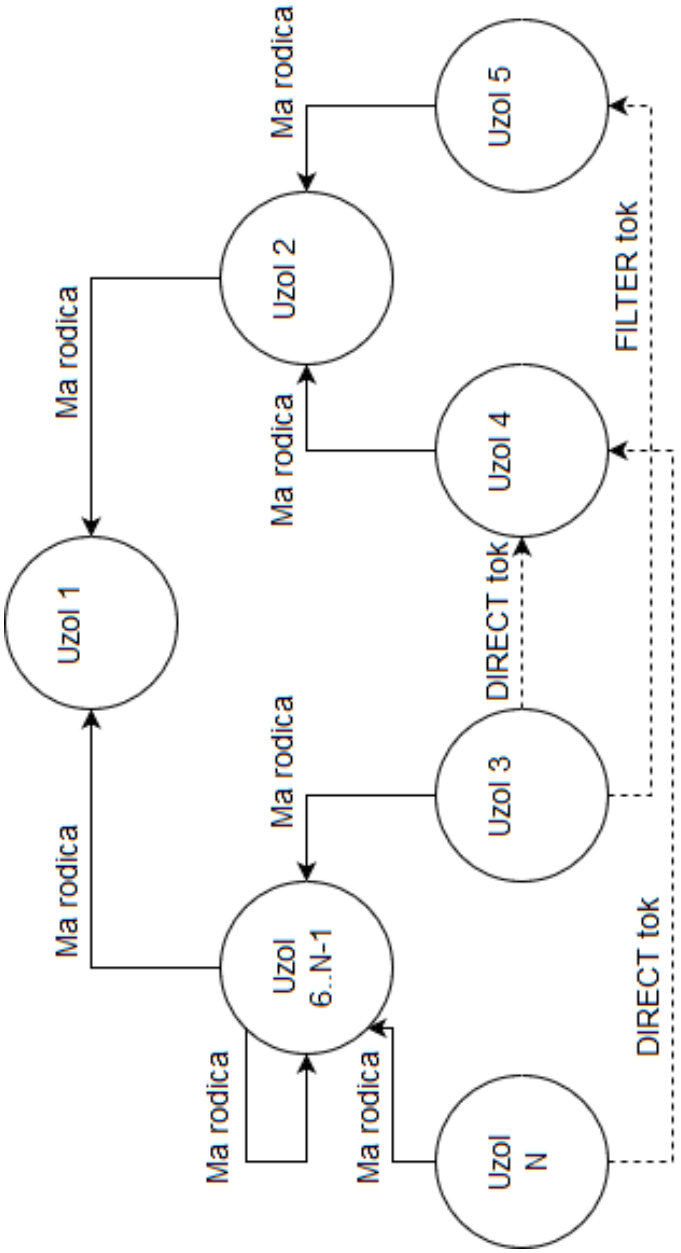
- Dátový tok - bude realizovaný ako entita, ktorá je závislá na entite uzla. Nebude možné sa dotazovať priamo na dátové toky. Dotazy budú riešené relatívne vzhľadom k štartovému alebo koncovému uzlu. Dátový tok bude obsahovať popisok, štartový uzol, koncový uzol a technológiu. Technológia, štartový a koncový uzol budú k dátovému toku priradené pomocou vnútorných ID. Bude tak možné sa na ne jednoducho ďalej dotazovať.

4.2 Serializácia grafu

Z rešerší uvedených v predchádzajúcej kapitole vyplynulo, že najvhodnejším spôsobom ako serializovať graf v prípade tohto projektu je kombinácia zaužívaného textového formátu CSV a textového formátu GraphSON.

Komunikácia medzi aplikačnou vrstvou a používateľom prebieha pomocou protokolu HTTP, čo je textový protokol a keďže sú oba tieto formáty textové, sú pre tento prípad vyhovujúce.

Kombinácia dvoch formátov je nutná kvoli rôznym požiadavkám zákazníkov, z ktorých niektorí preferujú formát CSV a niektorí vyžadujú formát založených na formáte JSON, napríklad GraphSON.



Obr. 4.1: Príklad logického modelu grafu

4.2.1 CSV

Hlavné dôvody pre použitie formátu CSV:

- Je to textový formát, čo je pre použitie s protokolom HTTP vhodné.
- Umožňuje prúdové spracovávanie, čo umožní efektívne spracovanie obrovských grafov.
- Podpora tohto formátu je veľmi široká a tak nebudú mať externé systémy s integráciou žiaden problém.
- Je ľahko čitateľný aj pre človeka a preto sú možné aj manuálne úpravy.

4.2.2 GraphSON

Hlavné dôvody pre použitie formátu GraphSON:

- Je to taktiež textový formát, čo je pre použitie s protokolom HTTP vhodné.
- Elementy grafu sú v ňom radené sekvenčne a preto je možné použiť prúdové spracovávanie.
- Je založený na formáte JSON a preto je podporovaný väčšinou systémov.
- Exitujú pre neho externé knižnice, ktoré umožňujú jednoduché parsovanie tohto formátu.

4.3 API

Z analýzy vyplynulo, že najvhodnejším prístupom k návrhu webového komunikačného rozhrania pre tento projekt bude jeho realizácia ako bezstavové webové rozhranie založené na princípe RESTful.
Stavovosť v tomto prípade nebude skoro vôbec potrebná, keďže sa jedná o získavanie konkrétnych metadát, ktoré nie sú súčasťou žiadneho procesu. Princíp HATEOAS preto nie je potrebný a kvôli zamedzeniu nadbytočnosti nebude použitý.

4.3.1 Rozdelenie URL

REST API bude používať URL, ktorá sa bude skladať z modulu projektu Manta (v tomto prípade "public"), z verzie API (v súčasnosti "v1"), ďalej z API zdroja a podzdroja, ktoré budú popísané v nasledujúcej sekcii a v niektorých prípadoch aj z parametrov. Pri niektorých zdrojoch bude súčasťou URL aj názov akcie, ktorá sa má nad zdrojom vykonať. Tento prístup bude použitý v prípadoch, kde nestačí jednoduché rozlíšenie akcií pomocou HTTP metód.

API zdroje a podzdroje:

- Uzol
- Technológia
- Atribút
- Hrana
- Revízia
- Import
- Export

4.3.1.1 Uzol

Identifikácia konkrétneho uzla bude riešená kombináciu dvoch uvažovaných riešení a to kompletnou cestou v grafe a interným ID. Takéto riešenie umožní užívateľovi pri prvej požiadavke na rozhranie zadať kompletnú cestu v grafe pre vyhľadanie uzla a v odpovedi bude spolu s údajmi o uzle aj ID uzla. Toto ID má nejakú dobu platnosť a tak ho môže užívateľ používať pre ďalšie dotazy v rámci tejto komunikácie.

4.3.1.2 Technológia

Identifikácia technológie bude riešená použitím interného ID, ktoré bude zadané ako atribút uzla a v prípade, že bude užívateľ chcieť o technológii podrobné informácie, bude možné sa pomocou tohto ID dotazovať nad osobitným API zdrojom. Takéto riešenie zabráni redudantnému prenášaniu dát v podobe podrobných informácií o technológii, ktoré väčšinou nebudú potrebné.

4.3.1.3 Revízia

Zdroj revízia bude umožňovať len obmedzené možnosti na dotazovanie. To znamená, že dotazovať sa bude možné len na poslednú revíziu a to nezávisle od toho, či je otvorená alebo zatvorená. Ďalej bude možné nad týmto zdrojom prevádzať operácie ako vytvorenie novej otvorenej revízie, zatvorenie poslednej revízie (commit) alebo vrátenie zmien poslednej revízie (rollback). Tieto operácie budú oddelené pomocou URL.

4.3.1.4 Import

Zdroj import umožní vkladanie grafu z jeho serializovanej podoby. Import bude podporovať dva formáty a to CSV a GraphSON. Rozlíšenie formátu bude riešené pomocou rozdielnej URL.

4.3.1.5 Export

Zdroj export bude umožňovať vkladanie súborov s DSL skriptami, ktoré budú následne spúšťane na strane serveru. Formát exportu a aj všetky ďalšie parametre budú súčasťou skriptu a nebudú sa vkladať pomocou API.

4.3.1.6 Ostatné zdroje

Ostatné zdroje závisia na existencii konkrétneho uzla. Napríklad atribút uzlu nemôže existovať bez toho, aby existoval jeho vlastnícky uzol. Taktiež nemôže existovať dátový tok medzi dvoma uzlami, ak jeden z týchto uzlov neexistuje.

Práve z týchto dôvodov budú atribúty a dátové toky skryté za zdroj uzla a vytvoria sa z nich tak podzdroje. Tieto podzdroje budú priamo závisieť na svojom nadradenom zdroji a vyrieši to priradzovanie atribútov a dátových tokov k uzlom.

Atomické operácie umožnia získať zoznam všetkých atribútov konkrétneho uzla a vyfiltrovať atribút uzla pomocou kľúča atribútu. Dátové toky bude taktiež možné získať podľa uzla alebo vyfiltrovať podľa popisku a smeru toku.

4.3.2 Formát správ

Správy, ktoré sa budú používať na výmenu dát prostredníctvom API budú využívať reprezentáciu vo formáte JSON. Je to hlavne z dôvodu vysokej rozšíriteľnosti a podpory. Formát JSON je na serializáciu jednotlivých objektov ako je uzol alebo atribút ideálnym riešením bez akejkoľvek redudantnej funkčnosti.

Správy nebudú používať žiadnu obálku, ktorá by obsahovala ďalšie dodatočné metadata, pretože to nie je potrebné.

4.3.3 Zabezpečenie

Zabezpečenie aplikačného programového rozhrania bude riešené stavovo pomocou tokenu. Tento prístup čiastočne porušuje princípy REST, avšak bezstavové riešenie autentifikácie v prípade tohto projektu nie je možné, pretože medzi požiadavky na riešenie patrí aj integrácia s externými autentifikačnými servermi, napríklad s LDAP.

Autentifikácia pomocou externého serveru vyžaduje vygenerovanie a zaslanie požiadavky a následné čakanie na odpoveď. Pri bezstavovom riešení by tento proces musel nastať pri každej požiadavke na API, čo by zvýšilo čas odpovede nad akceptovateľnú mieru.

Server pri prvej požiadavke vygeneruje token, ktorý bude mať časovo obmedzenú platnosť a užívateľ ho bude prikladať ku každej ďalšej požiadavke na API.

4.4 DSL

Návrh doménovo špecifického jazyka je najzložitejšou a najkomplexnejšou časťou aplikačnej vrstvy a musí spĺňať viaceré požiadavky, aby mohol byť v konečnom riešení použitý.

Hlavnou požiadavkou, ktorá vyplynula z analýzy bola nezávislosť na platforme a čo najmenšie množstvo inštalovaných súčastí. Práve kvôli splneniu týchto požiadaviek bol ako základ jazyka DSL navrhnutý skriptovací jazyk Groovy.

Dôvody použitia jazyka Groovy:

- Tento skriptovací jazyk je založený na syntaxe jazyka Java a je veľmi rozšírený. Preto je jeho využitie dobrou voľbou aj z hľadiska intuitívnosti, keďže budú užívatelia používať už známu technológiu.
- Jazyk Groovy je perfektne integrovateľný do Java projektov a nevyžaduje žiadne ďalšie podporné technológie. Na efektívny beh mu stačí Java Virtual Machine.

Jedná sa teda o interný doménovo špecifický jazyk, ktorého syntax je dosť podobná jazyku Java, avšak v tomto prípade to neznamená obmedzenie, ale výhodu, pretože si užívateľ rozhrania na jazyk DSL rýchlejšie zvykne, keďže je táto syntax veľmi známa.

4.4.1 Návrhový vzor

Ako návrhový vzor bude použité reťazenie metód. Tento spôsob bol vybratý preto, že výsledné DSL má oveľa jednoduchšiu štruktúru a na formovanie dotazov sa toto reťazenie výborne hodí. Zložitejšou časťou bude implementácia DSL objektov, ktoré budú musieť udržiavať momentálny stav doménových objektov a podľa toho poskytovať metódy, ktoré je v tomto stave možné volať.

4.4.2 Zabezpečenie

Zabezpečenie doménovo špecifického jazyka bude dosiahnuté pomocou vytvorenia osobitného uzavretého prostredia, v ktorom budú DSL skripty bežať. Toto prostredie musí mať vymedzené množstvo zdrojov, ktoré môže používať, ako z hľadiska hardvérovej kapacity, tak z hľadiska funkčnosti.

Obmedzenie bude realizované aj meraním času behu skriptu a jeho zastavením v prípade povolenia akceptovanej hranice.

Kapitola 5

Realizácia

Táto kapitola sa zaoberá použitím zvolených technológií za účelom implementovania aplikačnej vrstvy.

V tejto fáze vývoja aplikácie sú dôležité technologické znalosti a ich správne využitie na dosiahnutie všetkých funkčností, ktoré boli vytýčené v predchádzajúcej kapitole.
Realizácia je rozdelená do troch logických celkov:

1. Implementácia serializácie grafu
2. Implementácia programového webového rozhrania
3. Implementácia doménovo špecifického jazyka
4. Implementácia komplexnej analýzy

Všetky tieto dielčie časti spolu tvoria kompletné riešenie. Výstupom je funkčný softvér, pripravený na testovanie a následné opravy prípadných chýb.

5.1 Implementácia serializácie grafu

Pri implementácii serializácie grafu bolo potrebné vziať do úvahy rôzne možnosti použitia, napríklad pre použitie v rámci webového rozhrania ako serializátor objektov do HTTP odpovedí, ale zároveň aj pre použitie ako súčasť doménovo špecifického jazyka, v ktorom musí byť možné graf exportovať v jeho serializovanej forme.

Serializácia jednoduchých objektov (jeden konkrétny uzol) do formátu JSON je implementovaná pomocou knižnice Jackson, ktorá umožňuje priamy prevod Java objektov na odpovedajúce objekty JSON. Tieto Java objekty sú dopredu vymodelované pre použitie v odpovediach API a preto obsahujú len presne definované a požadované atribúty. Sú realizované formou POJO objektov.

Serializácia komplexného grafu do formátu CSV je implementovaná s využitím knižnice OpenCSV. Ako už názov napovedá, táto knižnica je open source a preto v použití nijako neobmedzuje. Java serializátor je implementovaný ako builder návrhový vzor a má preťažené metódy na pridávanie objektov priamo z grafového modelu.

Serializácia komplexného grafu do formátu GraphSON je realizovaná podobne ako serializácia jednoduchých objektov, čiže používa knižnicu Jackson, ktorá serializuje POJO objekty, avšak tieto objekty sú špeciálne vymodelované tak, aby spĺňali presný formát GraphSON. Serializácia a deserializácia je možná aj pomocou knižníc od TinkerPop, avšak v tomto prípade nebolo potrebné ich využiť, pretože väčšina funkčností je už v projekte implementovaná.

5.2 Implementácia webového API

Implementácia webového rozhrania bola v tomto prípade prispôsobená konvenciám Java frameworku Spring. Je využitý návrhový vzor MVC (Model-View-Controller), ktorý je implementovaný pomocou tried so Spring anotáciami.

Triedy, ktoré tvoria modelovú vrstvu priamo pristupujú do databázy pomocou rozhrania TinkerPop Blueprints a zapuzdrujú tak samotnú logiku dotazovania. Poskytujú verejné metódy, ktoré vo vnútri ošetrujú vstupy a v prípade chyby vracajú príslušnú výnimku pre kontrolerovú vrstvu.

Kontrolerová vrstva implementuje mapovanie URL a dotazov na API. Za účelom získavania a zápisu dát využíva výlučne metódy modelovej vrstvy. V prípade výnimky vyplní odpovedajúcu HTTP chybovú odpoveď. Pre reprezentáciu získaných dát využíva pohľadovú vrstvu.

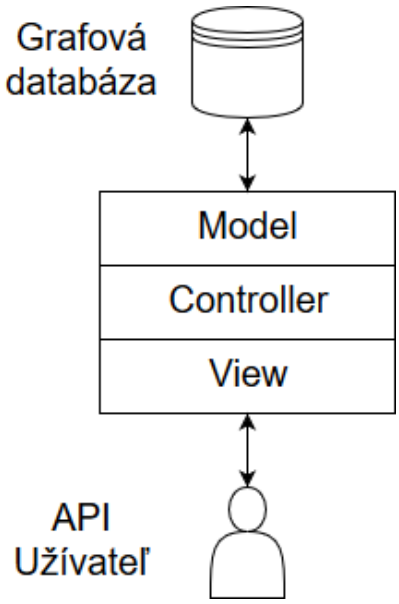
Pohľadová vrstva je v tomto prípade realizovaná ako skupina POJO objektov, ktoré naplní kontrolerová vrstva dátami a následne ich knižnica Jackson prevedie do serializovanej podoby (v prípade CSV je to knižnica OpenCSV).

Na obrázku 5.1 je znázornená celá schéma vrstiev a ako medzi sebou komunikujú.

5.3 Implementácia DSL

Doménovo špecifický jazyk je realizovaný ako vnorený Groovy shell, ktorý umožňuje spúšťať DSL skripty priamo na JVM servera. Toto prostredie je oddelené a má k dispozícii základné fukcionality jazyka Groovy, čo umožňuje neobmedzené možnosti pri implementácii komplexných analýz.

Pre rozšírenie možností jazyka Groovy pre prácu s grafovou databázou (a teda jeho povýšenie na DSL) sú pridané funkcionality, ktoré umožňujú rovnaké možnosti ako API (teda komunikújú s modelovou vrstvou), avšak sú implementované ako chainable návrhový vzor pre tvorbu DSL.



Obr. 5.1: Schéma MVC

Takýto prístup umožňuje vytvárať dotazy reťazením rôznych funkcionalít a približuje sa tak dotazovaciemu jazyku SQL, avšak je prispôsobený logickému modelu grafovej databázy.

DSL skripty umožňujú oveľa väčšie možnosti pre prácu s databázou, pretože bežia na serverovej strane a tým umožňujú udržiavanie stavu, čo REST API neumožňuje. Na obrázku 5.2 je zobraný stavový diagram, podľa ktorého je možné pracovať s revíznym systémom v databáze.

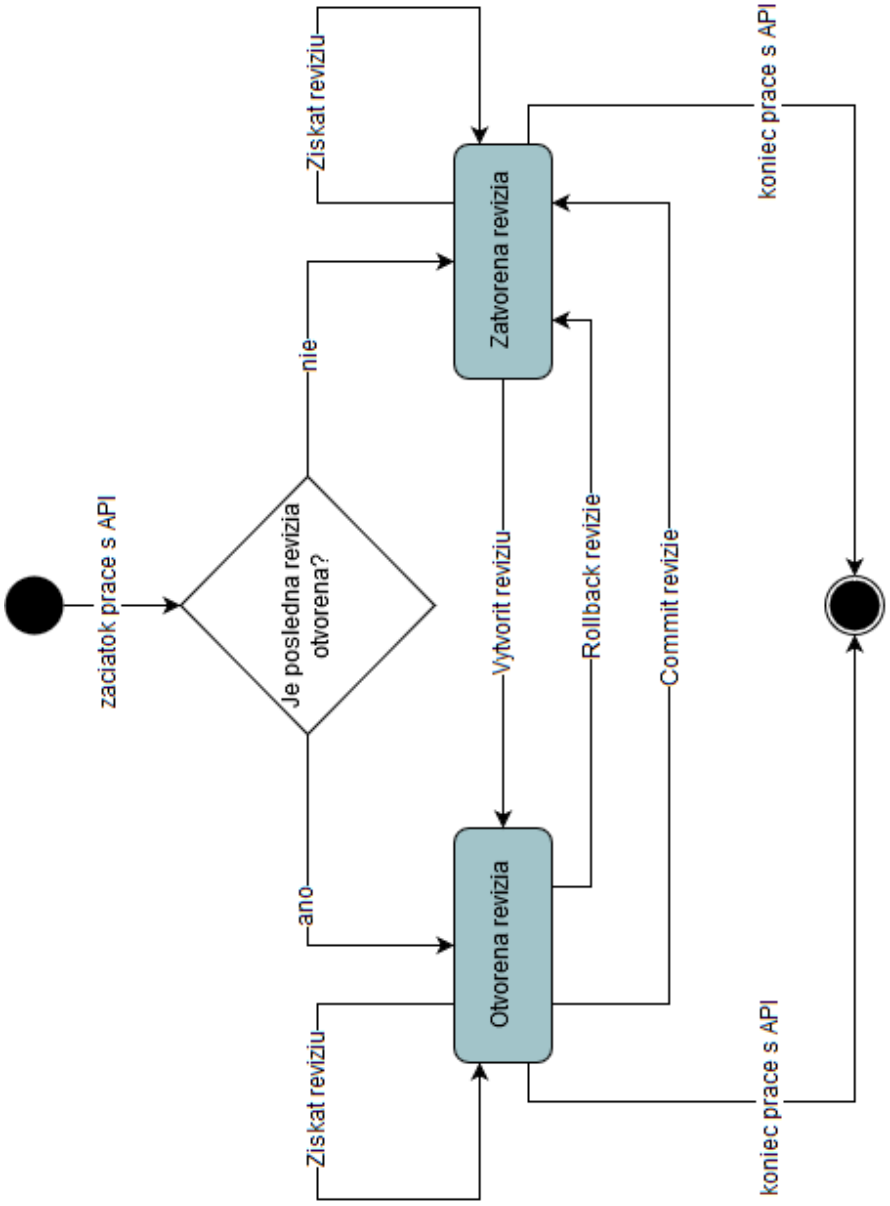
5.4 Komplexná analýza

Komplexná analýza je implementovaná pomocou jazyka DSL a jej cieľom je previesť analýzu dopadov (impact analysis) a nad výsledným grafom previesť filtráciu tak, aby vo výsledku ostali len koncové uzly v strome.

Táto analýza je zobrazená v nasledujúcom zdrojovom kóde, ktorý je okomentovaný tak, aby bola funkčnosť ľahko zrozumiteľná.

```
1 //ziskam cestu k startovemu uzlu
2 def startPath =
3   READ.revision 1 nodes 't2c1' filter 't2c1','Column' get 0 path()
4
5 //definujem zoznam ciest k startovym uzlom
6 def paths = [startPath]
7 //definujem zoznam koncovych uzlov
8 endNodes = []
9
```

```
10 //traverzovanie grafom smerom spat po priamych aj filter hranach
11 def graph = ANALYZE.traverse paths,'backward',true,null,null,10,1
12
13 //ziskam traverzer z grafu
14 traverser = graph.traverser()
15 //ziskam startovny uzol
16 def startNode = READ.revision 1 node startPath fetch()
17
18 //definujem funkciu pre rekurzivne vyhodnotenie koncoveho uzlu
19 def evalEnd(def node) {
20     //ziskam susedov uzlu
21     def neighbours =
22         traverser.current (node.getId()) neighbours 'in',null
23     //ak nema ziadnych dalsich susedov
24     if (neighbours.size() == 0) {
25         //pridam do zoznamu koncov
26         endNodes.add node
27     //inak
28     } else {
29         //pre vsetkych susedov
30         for (def neighbour in neighbours)
31             //zavolam funkciu pre vyhodnotenie konca
32             evalEnd(neighbour)
33     //koniec podmienky
34     }
35 //koniec funkcie
36 }
37
38
39 //zavolam funkciu pre vyhodnotenie konca pre startovy uzol
40 evalEnd(startNode)
41
42 //odstranim duplicitne hodnoty
43 endNodes.unique()
44
45 //vsetky koncove uzly
46 for (def node in endNodes)
47     //vyexportujem do formatu CSV
48     CSV.add node
49
50 //vypisem CSV
51 print CSV
```



Obr. 5.2: Stavový diagram revízie

Kapitola 6

Testovanie

Ďalšou fázou vývoja aplikácie je testovanie. Cieľom testovania je overiť či aplikácia spĺňa vlastnosti, ktoré boli požadované a či osahuje všetky nato potrebné súčasti. Pri testovaní sa okrem overenia funkčnosti zisťuje aj kvalita aplikácie. Jedným z najväčších prínosov tejto fázy je zistenie potencionálnych chýb v aplikácii, ktoré mohli vzniknúť pri vývoji.

Existuje veľa druhov a spôsobov testovania softvéru. Pri testovaní tejto aplikačnej vrstvy boli vybrané tie z nich, ktoré sú na testovanie tohto typu systému najvhodnejšie.

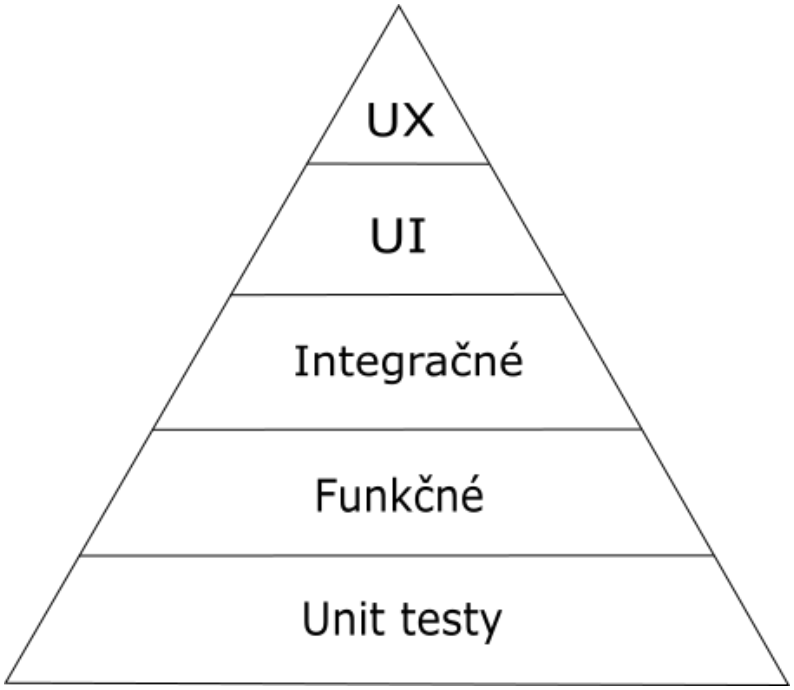
Testovanie prebiehalo už počas vývoja, napríklad prebehnutím Unit testov, no niektoré testy bolo možné vykonať až po skončení vývoja. Usporiadanie a náväznosť jednotlivých typov testov je znázornená na obrázku 6.1.

6.1 Unit testy

Tento typ testov sa zameriava na tie najmenšie časti aplikácie, akými sú napríklad jednotlivé funkcie, triedy či metódy. Pred začatím testu sa určia rôzne vstupy a k nim správne výstupy. Po prevedení takýchto testov a zistení chybného výstupu je hneď jasné, v ktorej časti aplikácie je chyba. Týmto sa dosahuje jednoduchá a rýchla oprava chýb. Na návrh jednotlivých testov je treba myslieť už pri samotnom vývoji.

Rozdelenie na jednotlivé funkčné bloky, ktoré je použité pri vývoji tejto aplikačnej vrstvy, rozdelí celú štruktúru na tri časti: atomické operácie, import a export grafu a doménovo špecifický jazyk. Takéto rozdelenie významne uľahčuje testovanie jednotlivých častí. Spúšťanie Unit testov je veľmi rýchle a preto je možné ich spustiť po každej väčšej zmene kódu a overiť tak zachovanie správnej funkčnosti.

Inak je to s časťami aplikácie, ktorých testovanie vyžaduje modifikáciu dát v databázi. Pred každým opätovným spustením je potrebné vrátiť príslušné dáta v databázi do pôvodného stavu a preto je náročnosť spustenia takýchto testov značne vyššia.



Obr. 6.1: Hierarchia testovania

6.1.1 JUnit

Konkrétna implementácia Unit testov v tejto aplikácii je dosiahnutá pomocou knižnice JUnit. JUnit je testovací framework určený pre projekty implementované v Jave. Je to doležitá súčasť každého test-driven projektu a podľa Githubu je to najpoužívanejšia knižnica v open source projektoch [16]. Každý test case je v kóde označený pomocou anotácií. JUnit je podporovaný väčšinou IDE programov a preto je možné po každej zmene kódu tieto testy jednoducho spustiť a po skončení zobraziť prehľadný výsledok.

6.2 Funkčné testy

Testovanie jednotlivých častí v Spring MVC architektúre je síce jednoduché a rýchle, avšak na komplexné testovanie aplikácie nepostačuje. Preto ďalším typom použitých testov su funkčné testy. Tento typ testov testuje aplikáciu ako celok a zameriava sa na overenie funkčnosti a priebeh procesu, ktorý siaha cez viacero jednotlivých častí aplikácie. Obvykle sa simulujú kroky v aplikácii, ktoré môžu nastať pri používaní. Toto testovanie prebieha vo viacerých kolách. Po nájdení chyby sa táto chyba opraví a nastáva ďalšie kolo, v ktorom sa krok, kde bola zistená chyba, otestuje znova.

6.2.1 Progresné testy

Progresné testy sú tie, ktoré sa spúšťajú po pridaní nových častí softvéru. Ich cieľom je overiť, či tieto nové časti spĺňajú požadované funkčnosti. Pri testovaní tejto aplikácie boli progresné testy spúšťané vždy po pridaní novej atomickej operácie a jej previazaní na webovú vrstvu alebo pri overovaní funkčnosti doménovo špecifického jazyka pre novú analýzu nad grafom.

6.2.2 Regresné testy

Regresné testy sa spúšťajú opakovane nad pôvodnými časťami aplikácie, aj v prípade, že pri pridaní nových súčastí neboli zmenené. Ich zmyslom je overiť, či funkčnosť všetkých súčastí ostala správna, teda či novo pridaná časť nezmenila chovanie niektorej z pôvodných súčastí.

Tieto testy sú pri projektoch veľkého rozsahu vhodné na automatizáciu, pretože sú spúšťané mnohokrát. Pri testovaní tejto aplikácie boli regresné testy využívané pravidelne, vždy po nejakej zmene funkčnosti.

6.2.2.1 Atomické operácie

Zoznam regresných testov atomických operácií je uvedený v tabuľkách 6.1 a 6.2.

Získaj	Čas [s]	Výsledok	Test
uzol podľa ID	0,233	uzol s daným ID	úspech
uzol podľa cesty	0,786	uzol na danej ceste	úspech
cestu k uzlu	0,718	zoznam uzlov tvoriaci cestu	úspech
uzly podľa mena	0,783	zoznam uzlov s daným menom	úspech
uzly podľa typu	0,918	zoznam uzlov daného typu	úspech
predka uzla	0,824	uzol tvoriaci predka	úspech
všetkých potomkov uzla	0,725	zoznam potomkov	úspech
potomkov uzla podľa mena	0,866	zoznam uzlov s daným menom	úspech
všetky atribúty uzla	0,810	zoznam všetkých atribútov	úspech
jeden atribút podľa kľúča	0,759	atribút s daným kľúčom	úspech
všetky hrany uzla	0,826	zoznam všetkých hrán	úspech
hranu podľa typu	0,945	hrana typu direct	úspech
hranu podľa typu	1,073	hrana typu filter	úspech
resource uzla	0,799	resource uzla	úspech
resource podľa ID	0,325	resource s daným ID	úspech

Tabuľka 6.1: Pozitívne regresné testy atomických operácií

Získaj	Čas [s]	Výsledok	Test
uzol s nevalidným ID	0,421	chyba	úspech
uzol s nevalidnou revíziou	0,953	chyba	úspech
uzol na nevalidnej ceste	0,784	chyba	úspech
uzol s nevalidným menom	0,779	chyba	úspech
uzol s nevalidným typom	0,760	chyba	úspech
neexitujúceho predka uzla	0,807	prázdny	úspech
neexistujúcich potomkov uzla	0,811	prázdny	úspech
atribút s neexistujúcim kľúčom	0,726	prázdny	úspech
neexistujúce atribúty uzla	0,759	prázdny	úspech
neexistujúce hrany uzla	0,972	prázdny	úspech
hranu s nevalidným smerom	0,797	chyba	úspech
hranu s nevalidným typom	0,791	chyba	úspech
resource s nevalidným ID	0,824	chyba	úspech
neexistujúci resource uzla	0,955	prázdny	úspech

Tabuľka 6.2: Negatívne regresné testy atomických operácií

Výsledky týchto testov overujú splnenie požiadavkov na atomické operácie, ktoré vyplývajú zo zadania projektu. Pozitívne testy overujú možnosť dotazovať sa na požadované entity grafu a rozne ich filtrovať a negatívne testy zase overujú ošetrenie chybných vstupov a dotazovanie sa na neexistujúce entity.

6.2.2.2 Operácie pre import grafu

Zoznam regresných testov operácií určených pre import grafu zo serializovanej reprezentácie je uvedený v tabuľkách 6.3 a 6.4.

Získaj	Čas [s]	Výsledok	Test
vráť poslednú revíziu	0,863	posledná revízia	úspech
vytvor novú revíziu	0,522	otvorená revízia	úspech
rollback poslednej revízie	0,986	rollbacknutá revízia	úspech
commit poslednej revízie	1,025	commitnutá revízia	úspech
importuj graf z CSV formátu	3,056	importovaný graf	úspech

Tabuľka 6.3: Pozitívne regresné testy operácií pre import

Získaj	Čas [s]	Výsledok	Test
vráť neexistujúcu revíziu	0,789	prázdny	úspech
vytvor revíziu bez commitu poslednej	0,786	chyba	úspech
rollback commitnutej revízie	0,818	chyba	úspech
rollback neexitujúcej revízie	0,793	chyba	úspech
commit commitnutej revízie	0,918	chyba	úspech
commit neexitujúcej revízie	0,889	chyba	úspech
importuj nevalidné CSV	1,823	chyba	úspech
importuj do commitnutej revízie	1,058	chyba	úspech

Tabuľka 6.4: Negatívne regresné testy operácií pre import

Výsledky týchto testov overujú funkčnosť aplikačnej vrstvy, ktorá sa týka importu grafu zo serializovanej podoby a operácie s revíziami, ktoré s importom nutne súvisia. Pozitívne prípady testujú vytváranie a modifikáciu poslednej revízie a samotný import serializovaného grafu z CSV súboru. Negatívne prípady overujú ošetrenie chybných vstupov a nevalidných stavov v procese importu.

6.2.2.3 Operácie pre export grafu a DSL

Zoznam regresných testov DSL operácií a operácií určených pre export grafu do jeho serializovanej podoby je uvedený v tabuľkách 6.5 a 6.6.

Získaj	Čas [s]	Výsledok	Test
podgraf podľa hrán smerom von	1,502	správny podgraf	úspech
podgraf podľa hrán smerom dnu	1,224	správny podgraf	úspech
podgraf podľa všetkých hrán	0,987	správny podgraf	úspech

Tabuľka 6.5: Pozitívne regresné testy operácií pre export

Získaj	Čas [s]	Výsledok	Test
uzol podľa ID	0,925	CSV s daným uzlom	úspech
uzly s atrinútmi podľa filtra	1,586	GraphSON s danými uzlami	úspech
uzly a hrany podľa filtra	1,718	GraphSON s danými uzlami	úspech
komplexná analýza	1,983	CSV s podgrafom	úspech

Tabuľka 6.6: Pozitívne regresné testy DSL operácií

Výsledky týchto testov overujú funkčnosť aplikačnej vrstvy, ktorá umožňuje vyexportovanie častí grafu podľa rôznych kritérií a filtrov. Export je umožnený do formátu CSV alebo GraphSON. Testovanie operácií pre export sa zaoberá samotným otestovaním správneho výstupného formátu grafu a testovanie DSL operácií overuje funkčnosť a možnosti doménovo špecifického jazyka a implementuje v ňom komplexnú analýzu.

6.2.2.4 Komplexná analýza pomocou DSL

V nasledujúcom kóde je uvedený príklad komplexnej analýzy pomocou DSL jazyka. Táto analýza nájde koncové uzly v podgrafe dátového toku.

```
//ziskam cestu k startovemu uzlu
def startPath =
  READ.revision 1 nodes 't2c1' filter 't2c1','Column' get 0 path()

//definujem zoznam ciest k startovym uzlom
def paths = [startPath]
//definujem zoznam koncovych uzlov
endNodes = []

//traverzovanie grafom smerom spat po priamych aj filter hranach
def graph = ANALYZE.traverse paths,'backward',true,null,null,10,1

//ziskam traverzer z grafu
traverser = graph.traverser()
//ziskam startovny uzol
def startNode = READ.revision 1 node startPath fetch()

//definujem funkciu pre rekurzivne vyhodnotenie koncoveho uzlu
def evalEnd(def node) {
    //ziskam susedov uzlu
    def neighbours =
        traverser.current (node.getId()) neighbours 'in',null
    //ak nema ziadnych dalsich susedov
    if (neighbours.size() == 0) {
        //pridam do zoznamu koncov
        endNodes.add node
    //inak
    } else {
        //pre vsetkych susedov
        for (def neighbour in neighbours)
            //zavolam funkciu pre vyhodnotenie konca
            evalEnd(neighbour)
    //koniec podmienky
    }
//koniec funkcie
}

```

```
38
39 //zavolam funkciu pre vyhodnotenie konca pre startovy uzol
40 evalEnd(startNode)
41
42 //odstranim duplicitne hodnoty
43 endNodes.unique()
44
45 //vsetky koncove uzly
46 for (def node in endNodes)
47     //vyexportujem do formatu CSV
48     CSV.add node
49
50 //vypisem CSV
51 print CSV
```

6.3 Integračné testy

Tento druh testov overuje aplikáciu z pohľadu iného systému. Spúšťajú sa až na konci vývoja, v poslednej fáze testovania. Pokrývajú celé procesy, ktoré môžu nastať behom používania aplikácie.

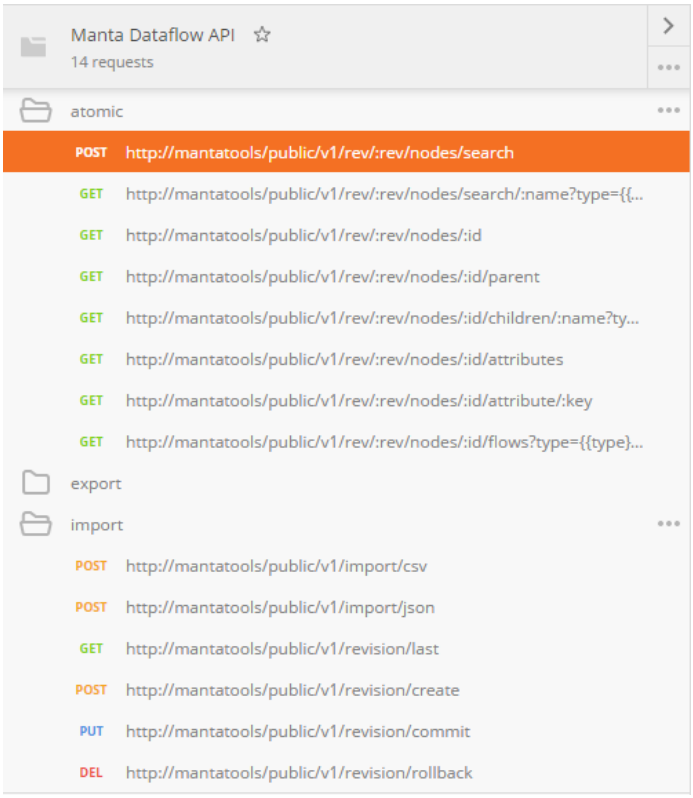
Keďže hlavným cieľom tejto práce bolo vytvoriť programové rozhranie a doménovo špecifický jazyk, aby boli iné systémy schopné s aplikáciu komunikovať, je tento typ testov úplne nevyhnutný. Systém, ktorý s aplikáciu komunikuje sa väčšinou simuluje pomocou rôznych mock-up nástrojov.

Pri integračnom testovaní tejto aplikácie boli použité nástroje Postman a SoapUI. Obe slúžia na simuláciu protistrany, s ktorou potom aplikačná vrstva komunikuje pomocou protokolu HTTP. Viac je o nich rozpísané v nasledujúcich sekciách.

6.3.1 Postman

Postman je moderný nástroj, ktorý umožňuje generovanie komplexných HTTP dotazov a analýzu HTTP odpovedí. Podporované sú všetky možnosti ako napríklad generovanie HTTP hlavičiek pre autentizáciu alebo vkladanie súborov. Nahradzuje prácu s knižnicou ako je napríklad cURL, ktorá má síce všetky možnosti ako Postman, ale práca s ňou je oveľa náročnejšia a zdĺhavejšia.

Tento nástroj taktiež umožňuje vytváranie kolekcií testov podľa súčasti, na ktorú sú testy zamerané (obrázok 6.3.1) a následnú automatizáciu celého procesu testovania. Postman sa podľa servera Stackshare umiestnil na druhom mieste v kategórii pomocný nástroj roka 2016 [17].



Obr. 6.2: Príklad kolekcie API testov

6.3.2 SoapUI

SoapUI je momentálne najpoužívanejší nástroj na testovanie SOAP a REST webových rozhraní [18]. Tento nástroj ponúka funkčné testovanie webových služieb, pokrýva WSDL popis rozhrania (ktoré od verzie 2.0 umožňuje aj popis REST služieb [19]) a je vyvýjaný pomocou open source prístupu. SoapUI je použitý ako komplement k nástroju Postman.

6.4 UI testy

Tieto testy sa používajú na testovanie grafického používateľského rozhrania. Ich cieľom je zistiť, či výsledné GUI spĺňa požiadavky na funkčnosť s ohľadom na zariadenie, na ktorom softvér beží. Aplikačná vrstva, ktorej sa táto práca venuje, však samotná neposkytuje žiadne grafické rohranie a preto tento druh testov nebol pri testovaní použitý.

6.5 UX testy

UX testy sa zameriavajú na overenie použiteľnosti softvéru. Cieľom je zistiť neintuitívne miesta v aplikácii a upraviť ich tak, aby bolo používanie softvéru čo najjednoduchšie. Väčšinou sa používajú v kombinácii s UI testami na otestovanie interakcie medzi človekom a softvérom.

V prípade tejto aplikačnej vrstvy, keďže neposkytuje grafické rozhranie, boli tieto testy použité na overenie intuitívnosti použitia webového API a implementácie analýzy v doménovo špecifickom jazyku. V úlohe užívateľa bol použitý človek s pokročilejšími technickými znalosťami a programovacími schopnosťami, pretože to bude aj typický užívateľ webového programového rozhrania a doménového jazyka.

6.5.1 Výsledok testovania UX

Scenár testovania UX sa skladá z testovania použiteľnosti aplikačného programového rozhrania pre účel dotazovania sa nad grafovou databázou a importu nového grafu.

Užívateľ bol pri používaní API veľmi efektívny už na začiatku a nemal problém s orientáciou medzi rôznymi API zdrojmi a operáciami nad nimi. Na problém narazil pri použití API pre import grafu, keďže sa nevedel zorientovať v stavoch aktuánych revízií a prechodoch medzi nimi. Tento problém bol vyriešený vytvorením stavového diagramu revízií. Celkovo užívateľ zhodnotil použitie API ako intuitívne.

6.6 Automatizácia testovania

Testovanie softvéru a hlavne jeho základných funkcionalít je nutné opakovať po každej väčšej zmene a to môže byť pracná a časovo náročná činnosť. Automatizácia testovania umožňuje vykonať túto činnosť efektívne a bez nároku na dodatočný čas. V prípade tohto projektu je nutné zaistiť automatizáciu unit testov, hlavne preto, že ich je väčší počet. Ako nástroj bol zvolený server Jenkins spolu s Maven rozšíreniami.

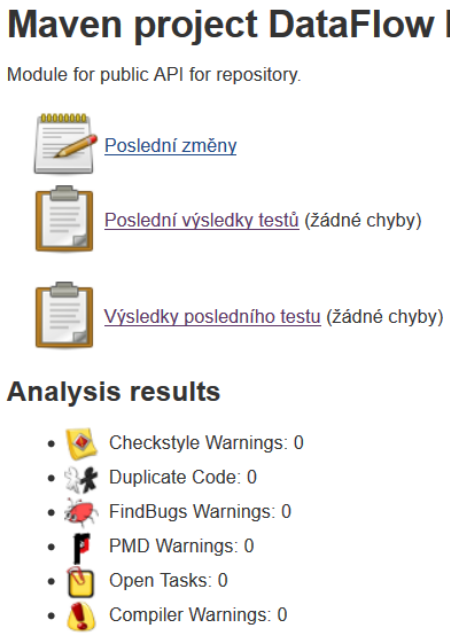
6.6.1 Jenkins

Jenkins je open source nástroj, ktorý beží ako samostatný server v prostredí Java a umožňuje automatizovať celý rad rôznych úloh ako je napríklad kompilácia projektov, testovanie a nasadzovanie. Nástroj je jednoducho rozšíriteľný pomocou pluginov. Spustenie kompilácie a testovania je riešené pomocou plánovača alebo nastavené na konkrétnu udalosť, napríklad zmenu vo verzovacom systéme, čo je pre tento projekt ideálne riešenie. Príklad výstupu Jenkins serveru je na obrázku 6.3.

6.7 Statická analýza kódu

Statická analýza kódu znamená analyzovať zdrojový kód bez toho, aby bol spustený. Všeobecne sa využíva k nájdeniu potencionálnych chýb a kontroluje korektnosť kódu podľa predefinovaných kódovacích pravidiel. Často sa statická analýza kódu integruje do procesu kompilovania softvéru, ako je to aj v prípade tohto projektu.

Na integrovanie statickej analýzy kódu do softvérového procesu sú v tomto projekte využité pluginy pre server Jenkins a to Checkstyle, FindBugs a PMD, ktoré pri každej zmene kódu spustia analýzu a zobrazia výsledky, príklad je na obrázku 6.3.



Obr. 6.3: Ukážka výsledkov z Jenkins serveru

6.7.1 Checkstyle

Checkstyle je nástroj pre statickú analýzu, ktorý podľa zadaných pravidiel kontroluje zdrojový kód. Tieto pravidlá sú zamerané skôr na prezentačnú stránku kódu a nedokážu kontrolovať správnosť logiky alebo kompletnosť programu.

Medzi typické chyby, ktoré Checkstyle kontroluje patria napríklad vynechanie komentára pri metódach, konvencie pre pomenovanie premenných, príliš veľká dĺžka riadku alebo prekročenie nastaveného limitu na počet argumentov funkcií.

6.7.2 FindBugs

FindBugs je taktiež nástroj pre statickú analýzu zdrojového kódu, avšak namiesto textu používa ako vstup samotný bytekód programu. Vyhľadáva časti bytekódu podľa vzorov, ktoré sú väčšinou zdrojom chýb.

Nástroj vráti upozornenia na podozrivé časti kódu, ktoré nemusia byť naisto chybné. V skutočnosti je šanca menej ako 50%, že bude kód správny [20]. Je potom na vývojárovi, aby celú situáciu overil.

6.7.3 PMD

PMD je nástroj na statickú analýzu kódu, ktorý je ale na rozdiel od dvoch predchádzajúcich zameraný na často sa vyskytujúce chyby v logike programu. Medzi takéto chyby patria napríklad prázdne sekcie na odchytávanie výnimiek, nepoužívané premenné alebo nepotrebné vytváranie objektov.

Niektoré funkcie PMD sú integrované priamo v mnohých IDE, takže na chybu upozorní už priamo pri písaní kódu, ostatné chyby sú potom analyzované pri kompilácii na serveri Jenkins.

Kapitola 7

Nasadenie

Táto kapitola sa zaoberá nasadením aplikácie v prostredí zákazníka (užívateľa API a DSL), ktoré bude umožňovať jej bezproblémový chod.

Projekt Manta sa skladá z viacerých samostatných častí, ktoré je možné nasadiť jednotlivo a na rôzne fyzické stroje. Podmienkou je, aby tieto stroje medzi sebou vedeli komunikovať pomocou protokolu HTTP, pretože ten sa využíva v celom projekte.

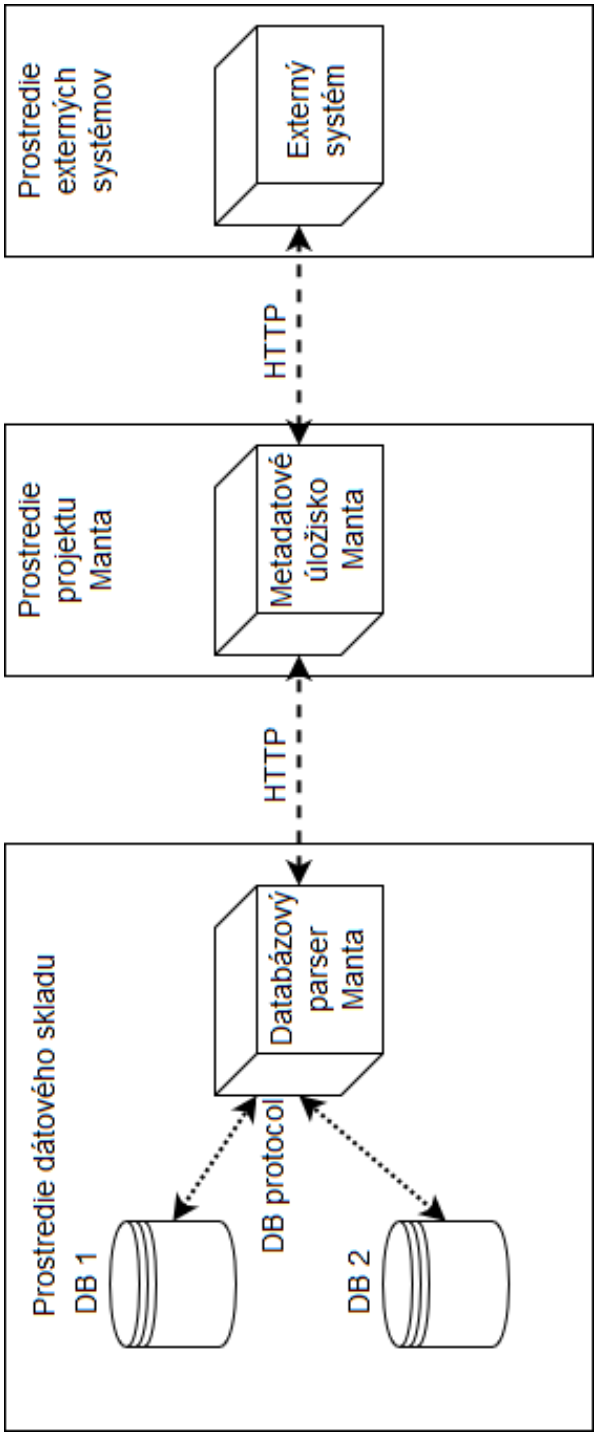
Samostatne nasaditeľné časti:

1. Databázový parser Manta
2. Metadatové úložisko Manta

Obe tieto časti sú zabalené do archívov JAR, čo umožňuje ich jednoduché nasadenie a aktualizáciu v prípade nových verzií. Jedinou podmienkou pre nasadenie samostatnej časti je nainštalované prostredie JVM, ktoré podporuje Javu verzie 7 a vyššiu.

Všetky súčasti projektu (napríklad webový server, databázový server a podobne) sú už zabalené v týchto JAR archívoch a preto je proces nasadenia veľmi jednoduchý. Stačí nakopírovať JAR archív požadovanej časti projektu na pre neho určený fyzický stroj s kompatibilným JVM prostredím.

Na obrázku 7.1 je znázornená schéma nasadenia projektu v prostredí dátového skladu.



Obr. 7.1: Schéma nasadenia

Kapitola 8

Dokumentácia

Táto kapitola zahŕňa všeobecný popis dokumentácie a takisto aj konkrétny spôsob dokumentovania častí aplikačnej vrstvy. Dokumentácia softvéru je častokrát zanedbávaná súčasť vývoja, avšak je veľmi dôležitá. Veľakrát nastáva prípad, kedy je potrebné upraviť či zmeniť nejakú časť a vtedy sa dokumentácia môže veľmi hodiť.

Bez dokumentácie by vývojár strávil viac času prechádzaním a študovaním zdrojového kódu ako jeho samotnou úpravou. Práve tento problém jednoducho a rýchlo vyrieši dobre komentovaný zdrojový kód. Vývojár tak dokáže rýchlo pochopiť všetky funkcionality a v zdrojovom kóde sa ihneď orientovať.

Pre dodržiavanie kvality komentárov je v projekte použitý Jenkins plugin, ktorý na prípadné nedostatky (chýbajúci komentár, nesprávny formát komentára) upozorní hneď po kompilácii projektu. Dokumentácia sa skladá z dvoch častí a to dokumentácie API a DSL.

8.1 Aplikačné programové rozhranie

Dokumentácia webového rozhrania je vytvorená v dvoch najpoužívanejších nástrojoch pre dokumentáciu REST služieb.

Prvá technológia je Open API Specification [21], ktorá bola vytvorená konzorciom priemyslových expertov a stal sa z nej otvorený a nezávislý štandard. Táto technológia je veľmi rozšírená a pôvodne bola založená na technológii Swagger.

Druhou technológiou použitou pre dokumentáciu REST API je API Blueprint [22], ktorý je veľmi jednoduchý a expresívny, zameraný hlavne na dizajn a REST API tohto projektu bolo v tejto technológii pôvodne navrhované.

Obe tieto dokumentácie sú k dispozícii spolu s touto prácou, v priložených súboroch.

8.2 Doménovo špecifický jazyk

Dokumentácia doménovo špecifického jazyka je vytvorená pomocou technológie JavaDoc a to z dôvodu jednoduchej prepoužiteľnosti komentárov a faktu, že je táto technológia priemyselným štandardom pre dokumentovanie projektov implementovaných v Jave.

Výsledok je prehľadná dokumentácia, ktorá s pomocou hypertextových odkazov umožňuje prehliadanie funkcionalít všetkých častí DSL. Táto dokumentácia je k dispozícii vygenerovaná ako súbory vo formáte HTML v priložených súboroch.

Záver

Zadaním tejto diplomovej práce bolo spracovať analýzu, návrh a vytvoriť prototypovú implementáciu aplikačnej vrstvy nad metadatovým úložiskom projektu Manta. Táto aplikačná vrstva umožňuje komunikáciu s dátovým úložiskom pomocou API, prácu so serializovanými grafmi a podporu pre DSL jazyk.

Prvým krokom bolo zanalyzovať existujúcu situáciu a vytýčiť podrobné požiadavky (funkčného aj technického charakteru) na konečné riešenie. Druhým krokom bolo navrhnúť riešenie tak, aby pokrývalo všetky požiadavky zadania práce a zároveň detailné požiadavky zistené počas analýzy. Ďalším krokom bola samotná implementácia riešenia a jej nasadenie na skúšobný server. Posledným krokom bolo testovanie (táto fáza sa v značnej miere prekrývala s implementačnou fázou použitím unit testov) a vyhodnotenie splnenia vytýčených cieľov.

Testovanie funkcionalít dopadlo úspešne (tabuľky v kapitole 6) a taktiež boli splnené všetky technické požiadavky na výsledné riešenie (aplikačná vrstva pracuje nad rozhraním TinkerPop Blueprints, komunikuje textovo cez protokol HTTP a umožňuje prúdové spracovanie grafov). Požiadavok na platformovú nezávislosť a jednoduchú inštaláciu bol splnený využitím JVM a Maven modulov. Serializácia grafu taktiež spĺňa všetky predpoklady z analýzy a to textovú podobu a dobrú podporu. Programové rozhranie je ľahko použiteľné a odtieňuje konzumenta od prebytočných detailov metadatového úložiska a zároveň podporuje jazyk DSL založený na syntaxe Groovy a tak spĺňa požiadavok na intuitívnosť a robustnosť. Z vyhodnotenia vyplynulo, že zadanie a ciele tejto práce sa podarilo splniť.

V budúcnosti je možné ďalšie rozširovanie tohto riešenia o ďalšie zaujímavé časti, ako je napríklad práca s vrstvami grafu, umožnenie zápisu pomocou atomických operácií, zapisovanie do ľubovoľnej revízie alebo rozšírenie importu a exportu grafu o ďalšie formáty serializácie.

Literatúra

[1] Cisco: The Zettabyte Era - Trends and Analysis. *White Papers*, Jún 2016. Dostupné z: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[2] Ralph Kimball, J. C.: *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2004, ISBN 978-0-7645-6757-5.

[3] DataStax: *Titan Storage Backend*. Dostupné z: http://s3.thinkaurelius.com/docs/titan/1.0.0/storage-backends.html

[4] Think Aurelius: *Gremlin Query Language*. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html

[5] Apache Software Foundation: *TinkerPop*. Dostupné z: http://tinkerpop.apache.org/docs/current/

[6] Peroutka, M.: *Optimální struktura a indexy modelu metadatového úložiště v grafové databázi*. ČVUT, 2017.

[7] Bloch, J.: *Effective Java (2nd Edition)*. Sun Microsystems, 2008.

[8] W3C: *Extensible Markup Language*. Dostupné z: https://www.w3.org/XML/Core/#Publications

[9] GraphML Team: *The GraphML File Format*. Dostupné z: http://graphml.graphdrawing.org/

[10] Emden Gansner, Eleftherios Koutsofios and Stephen North: *The DOT Language*. Dostupné z: http://www.graphviz.org/Documentation/dotguide.pdf

[11] ECMA International: *The JSON Data Interchange Format*. Dostupné z: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[12] Internet Engineering Task Force: *MIME Type for Comma-Separated Values (CSV) Files*. Dostupné z: https://www.ietf.org/rfc/rfc4180.txt

[13] FIT ČVUT: *Zuul OAAS*. Dostupné z: https://rozvoj.fit.cvut.cz/Main/oauth2

[14] Subramaniam, V.: Creating DSLs in Java. *JavaWorld*, August 2008. Dostupné z: http://www.javaworld.com/article/2077891/scripting-jvm-languages/scripting-jvm-languages-creating-dsls-in-java-part-3-internal-and-external-dsls.html

[15] Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Professional, 2010, ISBN 978-0321712943.

[16] Weiss, T.: The Top 100 Libraries in Java. *OverOps*, November 2013. Dostupné z: http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[17] StackShare, I.: Top developer tools of 2016. *Stacks*, December 2016. Dostupné z: https://stackshare.io/posts/top-developer-tools-2016

[18] Bear, S.: Soap UI. *Overview*, Jún 2016. Dostupné z: https://www.soapui.org/open-source.html

[19] W3C: *Web Services Description Language 2.0*. Dostupné z: https://www.w3.org/TR/wsdl20/

[20] Pugh, B.: FindBugs Fact Sheet. *FindBugs*, Marec 2015. Dostupné z: http://findbugs.sourceforge.net/factSheet.html

[21] Open API Initiative: *Open API Specification*. Dostupné z: https://www.openapis.org/

[22] AB Foundation: *API Blueprint*. Dostupné z: https://apiblueprint.org/documentation/

Dodatok A

Zoznam použitých skratiek

API Application programming interface

CSV Comma separated values

DBMS Database management system

DSL Domain specific language

ETL Extract transform load

GUI Graphic user interface

HATEOAS Hypermedia as the engine of application state

HTTP Hypertext transfer protocol

JSON Javascript object notation

LDAP Lightweight Directory Access Protocol

POJO Plain old Java object

RAML REST API markup language

REST Representational state transfer

SOAP Simple object access protocol

UI User interface

URL Uniform resource locator

UX User experience

XML Extensible markup language

Dodatok B

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF
  - `thesis.ps` ..... text práce vo formáte PS