VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÝCH TECHNOLÓGIÍ
FACULTY OF INFORMATION TECHNOLOGY

POKROČILÉ DATABÁZOVÉ SYSTÉMY
PROJEKT DO PREDMETU PDB

AUTOR PRÁCE **BC. ŠIMON POKORNÝ, BC. HENRICH ZRNČÍK**
AUTHOR

BRNO 2020

Kapitola 1

Specifikace zadání

Predpokladáme že zákazník nam zadal projekt pre bankovný inforačná systém. Zadávateľ projektu má sériu bánk, a bankomatov rozmiestnených po celom kontinente. Preto okrem bežných finančných transakcií medzi účtami potrebuje poskytovať aj možnosti platenia kartou či výber priamo v automate, alebo manipuláciu v banke. O každom účte predpokladáme že musí mať práve jedného majiteľa, ktorým je fyzická osoba a takisto niekoľko ručiteľov. Okrem toho systém eviduje lokality v ktorých sa jeho aktíva nachádzajú, takisto si udržuje informácie o adresách objektov (obchody, reštaurácie) v ktorých zákazníci platili. Zákazníkovi okrem tejto funkcionality treba poskytnúť aj trendy, v podobe rôznych agregácií ako mesačné útraty, či účty na ktoré smerujú jeho financie. Takisto je dôležité mať prehľad o aktivite užívateľa, čo znamená sledovať ako často sa prihlasuje a či úspešne. Okrem týchto bežných činností banka potrebuje mať schopnosť sledovať aktivitu medzi účtami, pokiaľ by napríklad označila nejaký z účtov za podozrivý (z legislatívneho hľadiska), Potrebuje byť schopná sledovať toky peňazí.

1.1 Rozkazy a dotazy

Na základe požiadavkov klienta sme identifikovali a rozdelili následujúce rozkazy a dotazy[1]

1.1.1 Commands

Commands alebo rozkazy predstavujú operácie ktoré majú nejakým spôsobom meniť dáta[2].

- **updateUserInfo** - Upraví základné užívateľské informácie o vlastníkovi účtu.
- **createDisponent** - Vytvorí disponenta pre daný účet.
- **deleteDisponent** - Zruší disponenta pre daný účet.
- **manipulateAccount** - Séria rozkazov ktoré nejak ovplyvňujú výslednu čiastku na účte.
  - **withdrawlCard** - platenie kartou v nejakom objekte (reštaurácia, bar, obchod atď...).

[1] Terminológia prevzatá z Command Query Responsibility Segregation

[2] V prípade kombinácie CQRS s Event sourcing, by tieto operácie mohli jedine zapisovať, náš projekt však bude predstavovať mierne volnejšiu verziu kde budeme umožňovať aj úpravu dát

  - **depositBank** - vloženie peňazí dispondentom alebo majiteľom účtu, priamo z banky.
  - **withdrawlBank** - výber peňazí dispondentom alebo majiteľom účtu, priamo z banky.
  - **makeTransaction** - prevod peňazí z účtu na účet.
  - **withdrawlATM** - prevod peňazí z účtu na účet.

Okrem týchto operácií bude existovať ešte jendoduchšie operácie typu create či update, tie tu ale pre jednoduchosť nebudeme vypisovať, kedže pre doménu entity ako krajina či mesto nemajú väčší význam.

1.1.2 Queries

- Primitivní dotazy
  - Základní informace o účtu (majitel, zůstatek)
  - Seznam posledních transakcí
- Pokročilé dotazy (grafový přístup)
  - Existuje spojení mezi dvěma účty?
  - Jaká je vzdálenost mezi dvěma účty?
  - Do kolika účtů vidí jeden uživatel?
  - Kolik spojení má daný účet?
  - Hodnota transakcí mezi dvěma účty?
- Pokročilé dotazy a agregáty (destrukturalizovaný přístup)
  - Ve kterých městech se nejvíce utrácí?
  - Průměrná/maximální/minimální hodnota výběru pro daný účet?
  - Z jakých států se daný uživatel kolikrát hlásil?
  - Obrat na účtu za měsíc?
  - Která města mají nejvyšší obraty?

1.2 Principy fungování

Níže jsou popsané některé reprezentační principy fungování jednotlivých příkazů a dotazů. Ostatní dotazy a příkazy jsou principielně velmi podobné.

1.2.1 Fungování commands

Veškeré operace, které mají charakter vytváření noých entit v systému budou vždy zasahovat do relačního modelu a následně pomocí projekce zaneseny také do nerelačních datových úložišť.

Příkladem může být command **logIn**, na jehož základě bude upravena relační databáze a až následně dojde k projekci dané akce do nerelačního databázového serveru. V případě, že nedojde k přihlášení uživatele (například z důvodu zadání špatného hesla), data se budou

propagovat do nerelační databáze také, výsledek daného příkazu bude jiný, aby bylo možné odlišit a analyzovat například pokusy o zcizení účtu.

Dalším příkladem je command **manipulateAccount - makeTransaction**, kde opět dojde k zpracování ve stejném pořadní, nicméně v případě, že na zdrojovém účtu není dostatečný obnos, data již nebudou projektována do nerelační databáze. Tento příkaz tedy bude zrušen a stornován. Obnobně bude tento přístup zvolen i u ostatních příkazů z této kategorie.

Posledním příkladem je command **createDisponent**, který by měl na úrovni relační databáze proběhnout téměř vždy naprosto bez problému (vyjímaje případu, kdy už daný disponent existuje), a tedy by i téměř vždy mělo do jít k propagaci dat do nerelačního datového úložiště, konkrétně do grafové databáze.

1.2.2 Fungování queries

Jakmile jsou data vkládána do systému, začnou se plnit i databáze, které jsou určené především pro čtení. Výsledky dotazů, které jsou prováděny nad nerelačními daty, odpovídají ekvivalentním výsledkům dotazů, které jsou tvořeny nad relační databází[3]. Pro zjednodušení je zde opět popsán vždy jeden zástupce z kategorie výše.

Pro kategorii **primitivní dotazy** není nutné vybírat konkrétního zástupce. Nacházejí se zde zpravidla dotazy, které jsou uloženy v rámci jedné relační tabulky, případně výstupy, které jsou sjednocením dvou tabulek dohromady. Tyto dotazy se provádí přímo v rámci relační databáze.

Z kategorie **grafový přístup** je vybrán zástupce *Jaká je vzdálenost mezi dvěma účty?* Jedná se totiž o ideální dotaz, který je položen nad některou z grafových databází. Při použití klasické relační databáze je možný pouze iterační přístup, který bude postupně hledat cestu z A do B (na úrovni relačního databázového stroje se jedná o přidávání propojení JOIN). Pomocí grafové databáze jsme schopni tuto informaci dostat jedním jednoduchým dotazem. Z výše uvedeného vyplývá, že se jedná při nejmenším o lepší přístup, než kdybychom data dolovali z klasické relační databáze. Pro zajištění správnosti výsledku je potřeba zajistit především správné vkládání hodnot při vykonávání příkazů.

Představitelem kategorie **pokročilé dotazy a agregáty** je dotaz *Která města mají nejvyšší obraty?* Pro tento typ dotazů je zvolena dokumentová databáze, která umožňuje jednoduše pracovat s agregáty a generovat agregáty nové. Opět je potřeba databázi aktualizovat s každým novým příkazem. Z dokumentové databáze je pak díky zavedení redundance a sjednocení některých entit velmi snadné získat požadovaná data.

1.3 Testování a ověření funkčnosti

Pro ověření funkčnosti je vždy v každém projektu vyžadována fáze testování, která ověří, zda navržený/implementovaný model vyhovuje zadaným podmínkám.

1.3.1 Testování příkazů

Jednotlivé příkazy budou vždy testovány následujícím způsobem:

- Zmapování aktuálního stavu databází.
- Provedení příkazu.

[3]Tento fakt je splněn pouze za podmínky, že máme plně synchronizovanné relační a nerelační databáze

- Porovnání aktuálního stavu databáze s předchozím uloženým.

Všechny tyto kontroly budou prováděny ručně přímým přístupem k jednotlivým databázovým strojům.

1.3.2 Testování dotazů

Abychom zajistili konzistentní a správný výstup dotazů, bude se postupovat dle následujícího scénáře:

- Modelování očekávaného výstupu ručně pomocí dat získaných pomocí přímého přístupu k databázovému stoji.
- Provedení dotazu pomocí webového aplikačního rozhraní (REST API).
- Porovnání očekávaného a skutečného výsledku.

V případě, že budou výsledky příkazů, respektive dotazů, odpovídat očekávaným změnám, repektive výstupům, bude test považován za úspěšný. V opačném případě bude upravena část aplikace tak, aby bylo možné test považovat za úspěšný a zároveň nedošlo k ovlivnění fungování ostatních dotazů a příkazů.

Kapitola 2

Návrh řešení

2.1 CQRS

Jedná sa o aplikáciu ktorá bude rozhodne obsahovať množstvo logiky na business vrstve a v budúcnosti vyžadovať škálovanie. Pre oba tieto prípady sa hodí použitie vzoru CQRS teda command query responsibility separation podľa ktorej následne popíšeme architektúru systému. Už v kapitole 1 sme popísali už aj konkrétne qurries a commands pre daný systém.

2.2 Zdroj pravdy

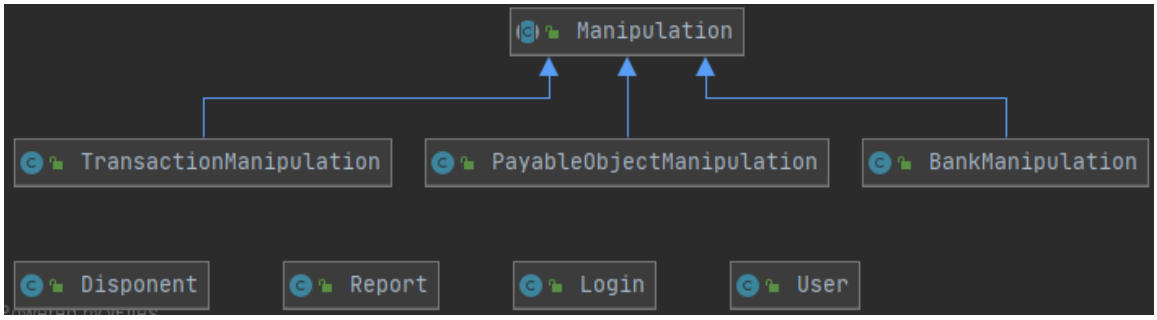
CQRS sa dá vhodne skombinovať s event sourcingom, v našom prípade sme však pre jednoduchosť zvolili ako zdroj pravdy relačný model (detail možeme vidieť na obrázku 2.2). Vďaka tomu že finančné transakcie budeme ukladať spôsobom ako pri event sourcingu budeme môcť aj pri relačnom prevedení niektoré jeho výhody zužitkovať (primárna nulovú stratu dát v tejto časti).

2.2.1 Synchronizácia

V ďalších kapitolách si podrobnejšie rozoberime zvolene NOSQL databázi pre model na čítanie, obidva tieto modely budeme synchrozovať zároveň so zápismi do relačného modelu. Dôvod je ten že vzhľadom na dve rôzne databázy, budeme môcť predpokladať čiastočnú funkčnosť aj pri absolutnom výpadku všetkých replík jednej databázy, Na čo sa teda snažíme zameriať je dostatok týchto replík a pomerne vysoká konzistencia a odozva.

2.3 Doménový model pro zápis

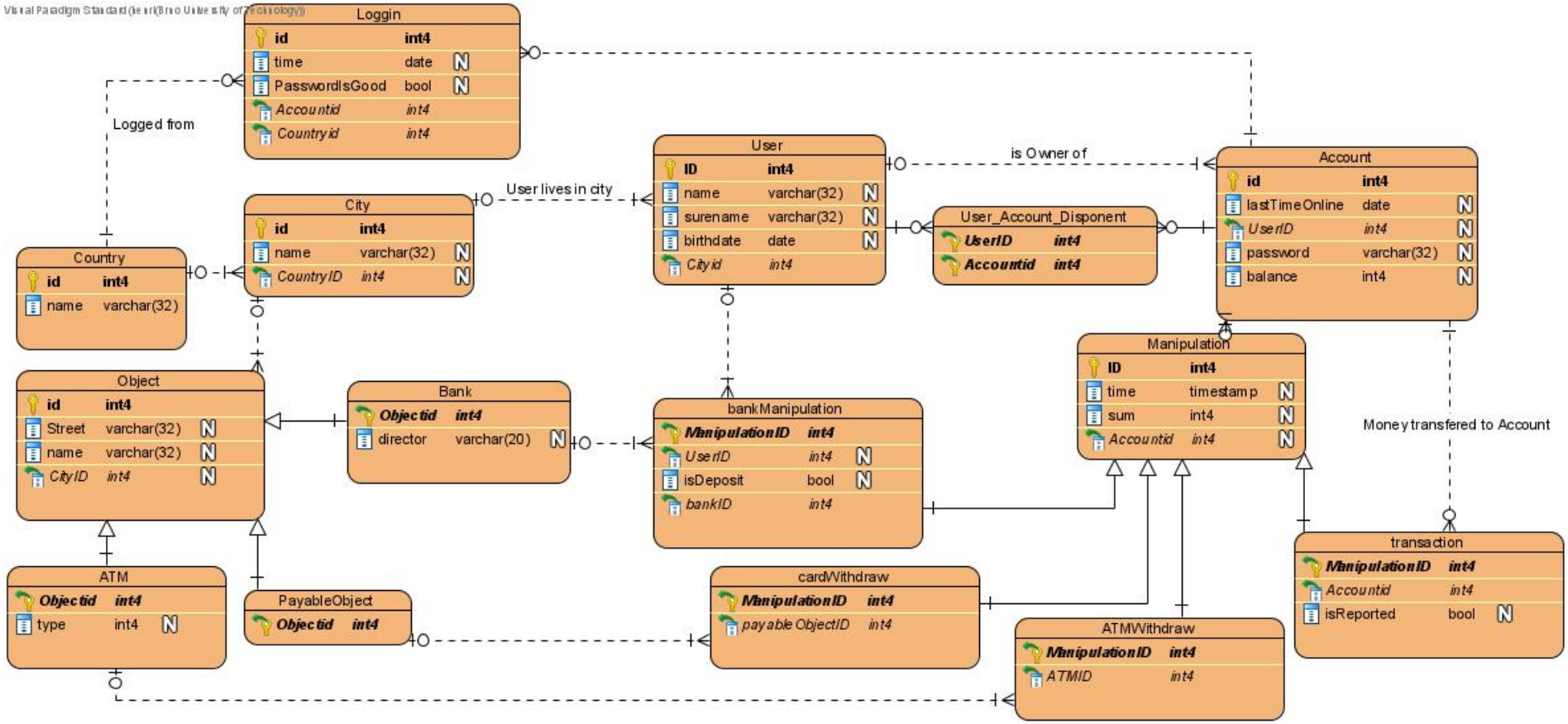
Čo sa doménovéhoho modelu a architektúry pre zapisovanie týka, identifikovali sme následné objekty (viď obrázok 2.1), ktoré v aplikácií reprezentujú objekty slúžiace pre zápis. Umiestnené budú v balíčku `domain`.



Obr. 2.1: Sťruktúra balíčku `domain`

Čo sa architektúry časti zápisov týka, požiadaky budú smerované na triedu `CommandHandler` ktorá následne rozhodne ktorú z tried z balíčku `commands` invokovať. Táto trieda už bude obsahovať odkaz na inštanciu triedy z balíčku `repository`, zodpoenú za vytvoreniu objektu podľa doménovej špecifikácie a následne vykonanie zmien v databáze.

2.4 Návrh schématu relační databáze



Obr. 2.2: Návrh schématu relační databáze

2.5 Neo4J

Jedná sa o NOSQL databázu grafového typu. Vzhľadom na sledovanie toku medzi jednotlivými transakciami, je pre túto časť uĺohy dobrou možnosťou, v inakšom prípade by sme logiku vyhľadávania museli takmer určite vniesť na aplikačnú úroveň. Namiesto toho sa nám touto cestou umožní poskytovať pomerne širokú škálu dotazov ohľadne transakcií.

2.5.1 Dotazy

- Existuje spojení mezi dvěma účty?
- Jaká je vzdálenost mezi dvěma účty?
- Do kolika účtů vidí jeden uživatel?
- Kolik spojení má daný účet?

2.5.2 Štruktúra

Všetky tieto dotazy by pre relačný model predstavovali veľký problém. Namiesto toho budeme účty, a užívateľov ukladať ako uzly a tranzakcie zas ako hrany. Pri aplikácií sa ráta aj s ďalšími možnými rošíreniami v tom prípade by za uzly mohli byť považované aj krajiny, vďaka už spomenutému čiastočnému ukladaniu na spôsob event sourcing budeme ukladať množstvo dát ktoré by sa pre takéto rozšírenia dali zreprodukovať.

2.5.3 Struktura Neo4J objektů

Níže je popsána struktura jednotlivých entit a vazeb, které budou v rámci projektu implementovány.

Entita Účet

Entita pro účet (*Account*) bude obsahovat následující údaje:

- **Id** - Identifikátor objektu
- **OwnerId** - Identifikátor majitele účtu
- **OwnerName** - Celé jméno majitele účtu

Vazba Transakce

Mezi jednotlivými dvěma účty bude provázáno za pomocí následující vazby:

- **Id** - Identifikátor transakce
- **Sum** - Finanční hodnota transakce
- **IsReported** - Označení, zda je transakce neočekávaná
- **Timestamp** - Časové razítko, kdy byla transakce provedena

2.6 Mongo

Na zvyšné projekcie sa skvelo hodí dokumentová databáza, ktorá by vďaka možnosti porušenia atomickostí umožnila naozaj jednoduché a rýchle čítanie pre agregáty a to riešenie požiadaviek na zisk trendov, a takisto vyhnutie sa nákladným operáciám (JOIN, group by) pre výpisi transakcií, v čom táto voľba teda naozaj vynikne je odstránenie **impedance mismatch**[1]. Vzhľadom na to že mongo je z pohľadu CAP teorému CP, teda consistence a partition tolerance, je veľmi schopné škálovatelnosti ale citlivé na výpadok. Z už spomenutej synchronizácia a rýchlosti prepočítavania sa vieme baviť o vysokej vierohodnosti dát v týhcto dokumentoch.

2.6.1 Dotazy

Dotazy nad mongom budú dve hlavné kategórie, primitívne dotazy a agregácie.

- Základní informace o účtu (majitel, zůstatek)
- Seznam posledních transakcí
- Obrat na účtu za měsíc
- Hodnota transakcí mezi dvěma účty

2.6.2 Štruktúra

Štruktúra bude celkovo odpovedať dotazom nad týmito dokuemntami. Vytvárane budú nad entitami **account** a zahňovať nasledujúce data.

- jednoduché informácie o majiteľovi
- jednoduché o informácia o učte
- zoznamov pokusov o prihlásenie
- zoznamy transakcií podľa jendotlivých typov
- zoznam objektov súvisacich s tranzakciami.

Nad týmito datmi budeme následne dopočítavať jednotlivé cieľové informácie.

2.6.3 Přesná struktura MongoDB objektů

Detailněji je pro snadné pochopení struktura dat rozepsána v podsekcích níže.

Zobrazení dat pro uživatele

Pro rychlé zobrazení následujícího přehledu bude uživateli zobrazován jeho záznam z následujícího typu dokumentu.

[1] Odstánenie odporu medzi relačným a objektovým "svetom".

```
[{
    AccountID: int,
    UserID: int,
    UserName: string,
    TransactionCount: int,
    Balance: int
}]
```

Manipulace s účtem

Jedná se o typ dokumentu, který uchovává pole všech provedených transakcí v rámci systému. Nad tímto polem se budou agregovat data pro jednotlivé účty a statistické soubory.

```
[{
    ManipulationID: int,
    AccountID: int
    Timestamp: dateTime,
    Sum: int,
    Type: [Transaction/Bank/CardWithdraw/ATMWithdraw]
    isDeposit: boolean,
    ObjectId: int,
    Street: string,
    ObjectName: string,
    ObjectCity: string,
    ObjectCountry: string
}]
```

Kapitola 3

Implementace

3.1 Zvolené technologie

Jako implementační prostředí jsme si zvolili jazyk *Java* využívající *Maven*, který se staral o všechny potřebné závislosti. Jednoduše jsme tak mohli pouze přidat požadavek na knihovnu například pro připojení k NoSQL databázi a Maven se o vše již postaral sám. Celá jeho definice se nachází ve standardním souboru `pom.xml`.

Celý projekt je napsán pro webový server *Tomcat*, který naší aplikaci spouští. Pro grafické uživatelské rozhraní jsme použili *Java Servlet*, který nám tvoří komunikační vrstvu mezi uživatelem a aplikací. Zároveň také odděluje jednotlivé obrazovky co se HTML stránky týče a logiky, která jí řídí.

Na úrovni kódu se aplikace skládá s tříd DAO (z anglického Data access object), které fungují jako fasáda mezi servlet třídami a databází. Doménový model pak tvoří jednotlivé objekty, se kterými se pracuje napříč celou aplikací. Speciální třídy jsme pak tvořili přímo pro grafovou databázi. Ty reprezentují přímo objekty, se kterými databáze pracuje.

Samotnou vrstvu komunikace s databázovými stroji v tomto projektu zajišťuje modul `utilities`, který obsahuje jednotlivé konfigurace pro připojení k různým databázovým strojům.

3.1.1 Zdroj pravdy

Jako zdroj pravdy byla zvolena databáze *MySQL*, která udržuje všechny informace v reálném čase. Jejím úkolem je také zjišťovat případné nesrovnalosti, které jsou zde definovány pomocí integritních omezení, jako jsou například primární a cizí klíče.

Ke komunikaci s databází bylo použito frameworku Hibernate. Byl zvolen, protože umí automaticky vygenerovat databázové tabulky z předepsaných tříd v kódu a zároveň zajišťuje mapování jetnotlivých řádků tabulek na již připravené třídy.

3.1.2 Grafová databáze

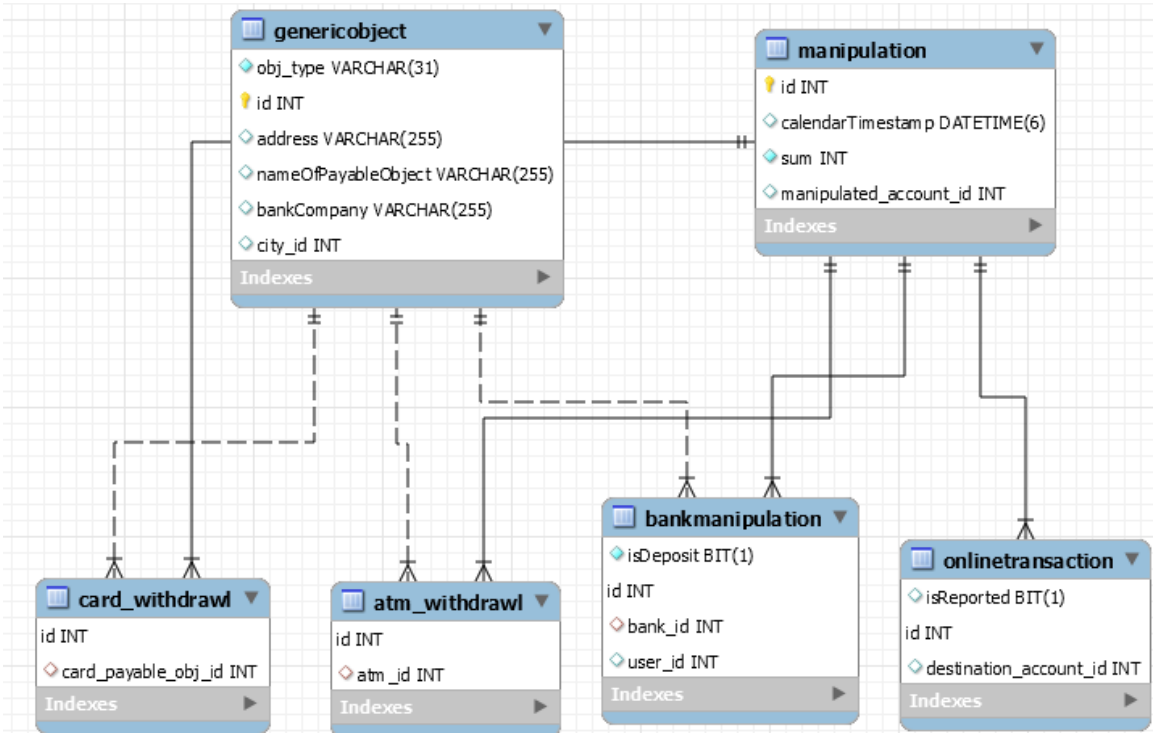
Grafovou databázi jsme zvolili *Neo4j*, která nám přináší především rozsáhlou dokumentaci, která umožnila její jednoduché začlenění do projektu. Veškeré potřebné příklady jsou k dispozici a mohli jsme se tak jendoduše inspirovat.

3.1.3 Dokumentová databáze

Dokumentovou databázi nám zde představuje *MongoDB*. Obět byla zvolena z důvodu jednoduchého zapojení do projektu a to jak z hlediska zadávání nových hodnot do databáze, tak jednoduchému vyčítání hodnot.

3.2 Relační databáze

Zjednodušený model databáze, který používáme je vidět na obrázku 3.1. Z brázku je vidět, že většina databázových tabulek je tvořena jedním společný generickým objektem. Díky tomu jsme měli jednoduše zajištěnou propagaci společných vlastností.



Obr. 3.1: Zjednodušená struktura databáze

Pro zajištění správnosti dat, je vždy před jakoukoliv manipulací proveden test, že je možné data vložit. Příkladem může být odeslání peněz z jednoho účtu na druhý, přičemž na prvním účtu musí být dostatečný obnos. Tento test se prování vždy nejdříe na úrovni relační databáze a až v případě, že se vše povedle bez problému, jsou data zanesena také do NoSql databází, které se používají k analýze dat.

Obrázek výše je pouze demonstrace zjednodušení tvorby některých databázových tabulek. Schéma na obrázku 2.2 odpovídá reálném stavu databáze.

3.3 Neo4J

Grafová databáze Neo4J využíváme především ke kontrole a zjišťování rozsáhlédo propojení zvolených subjektů. V praxi se provádí dotaz na databázový stroj, který vyhodnocuje, zda

jsou spolu nějaké 2 účty propojeny, případě skrze kolik dalších objektů. Lze tak určovat jednoduše jejich vzdálenost.

V databázi se nacházejí 2 typy entit.

- Uzel typu `Account`, který obsahuje refereci na ID v relační databázi a celé jméno uživatele.
- Propojení `Transaction`, které naopak reprezentuje jednotlivé hrany grafu. Hodnoty, které pak drží jsou zdrojový objekt, cílový objekt, identifikátor transakce, který je uložený v relační databázi a hodnotu, která byla předána.

3.3.1 Dotazy grafové databáze

Grafová databáze je využívána především pro zjištění existence spojení a nalezení nejkratší cesty mezi dvěmy uzly. Využíváme proto několika následujících dotazů. Pro vložení nových uzlů se využívá příkazu `CREATE`, pokud chceme některý z účtů vrátit, využíváme příkazu `MATCH`. Klíčové slovo `RETURN` pak definuje, jaké hodnoty daný dotaz bude vracet.

Vložení nového uzlu

```
CREATE (a: Account {ownerFirstName: $ownerFirstName,
    ownerLastName: $ownerLastName, accountId: $accountId, ownerIsFrom: $ownerIsFrom})
    RETURN a.ownerFirstName + ', from node ' + id(a)
```

Vložení nového spojení

```
MATCH (a1 :Account {accountId: $sourceId})
    MATCH (a2 : Account {accountId:$destinationId })
    CREATE (a1) - [t :TRANSACTION {id: $transactionId , sum:$sum}]
        -> (a2) return a1, t, a2
```

Zjištění spojení mezi dvěma účty

```
MATCH (n: Account {accountId : $sourceId})
    -[*]-(m: Account {accountId: $destinationId})
    RETURN n.accountId
```

Zjištění nejkratší cesty mezi uzly

```
MATCH (a1:Account { accountId: $sourceId }),
    (a2:Account { accountId: $destinationId }),
    p = shortestPath((a1)-[*..15]-(a2))
    RETURN length(p)
```

3.4 MongoDB

Dokumentová databáze MongoDB nám umožňuje ukládat data s volnější strukturou. Hlavní předností pro nás pak je především možnost vyhledávání v dokumentech jako takových s možností vyhnutí se komplikovaným konstrukcím typu `Join`, nebo například `Group by` a to díky zavedení lehké duplicity v datech. Protože využíváme databáze právě za účelem rychlosti, může pro nás být více databázových strojů pozitivních vzhledem k možnosti distribuce části dotazů.

3.4.1 Dotazy dokumenové databáze

Zatímco dotazy v grafové databázi se zaměřují především na řešení vztahů mezi účty, dokumentová databáze řeší především zjednodušení agregací a manipulaci se zdroji v rámci jedno uživatele.

Vložení nové transakce mezi účty

Abychom mohli správně nad daty agregovat, musíme vložit do databáze, při převodu z účtu na účet, rovnou 2 záznamy. První definuje odchozí platbu, druhý pak platbu příchozí na cílovém účtu.

```
Document transactionOut = new Document("transactionId", transactionId)
    .append("isIN", false)
    .append("isTransaction", true)
    .append("sum", sum)
    .append("date", dateString)
    .append("destinationAccountId", destinationAccountId);
Document transactionIn = new Document("transactionId", transactionId)
    .append("isIN", true)
    .append("isTransaction", true)
    .append("sum", sum)
    .append("date", dateString)
    .append("sourceAccountId", sourceAccountId);
```

Vložení nových zdrojů k účtu

K vložení nových zdrojů k účtu využíváme velmi podobného příkazu, jako v předchozím případě. Jednoduše tedy vyrobíme nový dokument, který pak předáme přímo databázovému stroji.

```
new Document("transactionId", transactionId)
    .append("isIN", isDeposit)
    .append("isTransaction", false)
    .append("sum", sum)
    .append("date", dateString)
    .append("bankName", "New York Bank");
```

Získání agregace jednotlivých manipulací

Abychom mohli dobře manipulovat s daty, musíme si nejdříve vybrat všechny transakce, které náleží danému účtu, destrukturalizovat jednotlivé manipulace v účtu (pomocí příkazu `unwind`) a poté je možné je sjednotit do jednoho výstupu, který je pak předán uživateli nazpět do uživatelského rozhraní.

```
match(Filters.eq("accountId", id)),
    project(Projections.include("accountId", "accountManipulations")),
    unwind("$accountManipulations"),
    group("$accountManipulations.dateAggregateFormat",
        Accumulators.sum("total", "$accountManipulations.sum"))
```

3.5 Grafické uživatelské rozhraní

V rámci aplikace jsme implementovali jednoduché webové grafické rozhraní, které umožňuje uživateli jednoduše zadávat nová data. K dispozici má uživatel rutiny jako:

- vložit peníze na účet,
- vybrat peníze z účtu,
- převést peníze na jiný účet,
- zkontrolovat stav svého účtu.

Všechny tyto metody jsou přístupné až po přihlášení uživatele. Pokud uživatel není přihlášen, je k dispozici možnost analýzy. Uživatel tak jednoduše může zjistit například propojení mezi účty.

3.6 REST API

Abychom umožnili přístup také přímo jednotlivým aplikacím, vytvořili jsme také webové aplikační rozhraní. Následující podkapitoly popisují způsob použití a typy výsledků, které se vracejí. Rozhraní může být také použito pro jednodušší automatizované testování.

Vzhledem k jednoduchosti naší aplikace jsme veškerá volání API implementovali skrze požadavky typu HTTP GET. Veškeré výstupy z aplikačního rozhraní jsou ve jednoduše čitelném formátu JSON.

Veškeré chybové výstupy jsou reprezentované jednoduchou hodnotou zobrazenou níže.

```
{"result":"wrong"}
```

Vložení nové transakce

Endpoint `/accAccSumTransaction?from=1&to=2&sum=5`

```
{"result":"good"}
```

Informace o účtu

Endpoint `/acc?id=5`

```
{
    ownerName: "Tony",
    ownerLastName: "Fat",
    balance: 104990
}
```

Jednotlivé transakce na účtu

Endpoint `/accTransaction?id=1`

```
[
    {
        id: 25,
        sum: 5000,
        destinationAccId: 2,
        date: "2020-12-06",
        isIn: false,
        sourceAccId: 0
    },
    {
        id: 2,
        sum: 10,
        destinationAccId: 3,
        date: "2020-12-09",
        isIn: false,
        sourceAccId: 0
    }
]
```

Zobrazení celkového průtoku zdrojů na účtu

Endpoint `/accTrend?id=1`

```
[
    {
        id: 202012,
        totalManipulation: 125030
    },
    {
        id: 202011,
        totalManipulation: 130500
    }
]
```

Zjištění propojení mezi účty

Endpoint `/accConnected?from=1&to=2`

```
{
    Connected: true,
    Path: 1
}
```

Založení nového účtu

Endpoint `/accCreate?balance=10&first=Karel&last=Karlovic`

```
{"result":"good"}
```

Propojení mezi účty

Endpoint `/accConnections?id=1`

```
{
    Connections: 10,
    Neighbor: 3
}
```

Kapitola 4

Testování

Testování celého systému probíhalo především pomocí předem definovaného scriptu, který je k dispozici v odevzdaném archivu v souboru `test.sh`.

Testovány jsou všechny vytvořené enpointy, pomocí jednoduchých metod. Příklad takové metody je ve výpisu níže.

```
def accConnections(id, connections = 0, neighbor = 0):
        url = "http://localhost:8080/mProj_war/accConnections?id=" + str(id)
        print(url)
        r = requests.get(url=url)
        j = json.loads(r.text)
        print(j)
        if not j["Connections"] == connections:
                raise Exception("Connections is not eq")
        if not j["Neighbor"] == neighbor:
                raise Exception("Neighbor is not eq")
```

Tato metoda se stará o kontrolu počtu v rámci propojení účtů. Jako vstupní parametry slouží identifikátor účtu, který chceme sledovat a očekávaný počet propojení a sousedů. V případě, že nebude jedna z očekávaných hodnot souhlasit, bude vyhozena chyba a script nebude dále pokračovat.

Samotný test obsahuje jednoduchý scénář, kdy jsou nejdříve vytvořeny účty, vytvořeny transakce a porovnávání kontrolních hodnot. Honoty jsou porovnávány v závislosti s tím, co se právě díky transakcím děje. Většina částí scénáře tak obsahuje vstupní a výstupní podmínky. Celkový scénář je vidět v dalším výpisu.

```
if __name__ == '__main__':
        accA = testCreate(5) #77
        accB = testCreate(0) #78
        accC = testCreate(0) #79

        accBalance(accA, 5)
        accBalance(accB, 0)
        doTran(accA, accB)
        accBalance(accA, 0)
        accBalance(accB, 5)
        doTranErr(accA, accB)
```

```
accTrend(accA, 5)
accTrend(accB, 5)

accConnected(accA, accB, True, 1)
accConnected(accB, accC, False, 0)
doTran(accB, accC, 3)
accConnected(accA, accC, True, 2)

accConnections(accA, 1, 1)
accConnections(accB, 2, 2)
doTran(accB, accA, 2)
accConnections(accB, 3, 2)
```

Je velmi důležité podotknout, že testování nám velmi pomohlo odhalit několik nesrovnalostí v rámci kódu. Příkladem může být počítání sousedů vybraného účtu. Díky chybné implementaci Neo4J dotazu byla počítána pouze jednosměrná závislost, nikoliv libovolná, což zapříčinilo špatné výsledné hodnoty.

Kapitola 5

Spuštění projektu

Pro spuštění projektu je třeba určitého programového vybavení:

- Java vývojové prostředí (například IntelliJ IDEA od firmy JetBrains),
- Apache Tomcat, který spouští celý projekt a představuje webový server,
- MySQL databázový stroj (konfigurace v souboru `hibernate.cfg.xml`),
- MongoDB databázový stroj (konfigurace v souboru `MongoUtil.java`),
- Neo4j databázový stroj (konfigurace v souboru `Neo4JUtility`),
- funkční rozšíření Maven, který zajistí stažení veškerých požadovaných závislostí.

V případě, že jsou všechny výše uvedené nástroje k dipozici v počítači a správě vyplněny veškeré konfigurační soubory pro připojení do databází, je možné projekt spustit pomocí oblíbeného vývojového prostředí.