Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Webová aplikace poskytující data o interakci aplikací v integrační platformě

Bakalářská práce

Autor: Lukáš Zoubek
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové červen 2016

Prohlášení:

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

V Hradci Králové dne 26.4.2017 Lukáš Zoubek

Poděkování:

Rád bych poděkoval doc. Ing. Filipu Malému, Ph.D. za jeho cenné rady a připomínky. Dále bych rád poděkoval Ing. Martinu Souškovi za umožnění vytvoření této práce a vstřícné konzultace ze strany firmy. Poděkování patří také mé rodině za podporu při tvorbě této práce.

Anotace

Bakalářská práce se zaměřuje na postupný vývoj webové aplikace, která automaticky provádí převod dat z relační databáze do databáze grafové. Cílem aplikace je poskytnout data o interakci aplikací v integrační platformě pomocí webové služby. Jednotlivé kapitoly práce jsou zaměřeny na webové technologie, databáze a metodiky, s kterými se kolem vývoje webových aplikací setkáváme. Tyto poznatky jsou dále použity v praktické části k návrhu grafové databáze a implementaci aplikace. Praktická část je zaměřena především na serverovou část aplikace. Konec praktické části se zabývá front-end částí aplikace, konkrétně vizualizací grafu.

Klíčová slova: webová aplikace, grafové databáze, webové služby, graf

Annotation

Title: Web application providing data about the interaction of applications in integration platform

The Bachelor Thesis focuses on the gradual development of the web application, which is automatically converting data from relational database into graph database. The aim of the application, is to provide the data about the interaction of applications within the integration platform, using web service. Individual chapters focus on web technologies, databases and methodologies, which we are around web application development meeting with. These findings are then used in a practical part, for designing graph database and implementation of the application. The practial part is focused mostly on the server side of the application. The end of the practical part deals with application‘s front-end, namely graph visualisation.

Keywords: web applications, graph databases, web services, graph

Seznam zkratek

HTML..... Hypertext Markup Language (značkovací jazyk)

CSS..... Cascading Style Sheets (styly definující vzhled HTML elementů)

MVC..... Model, View Controller (komponenty architektury aplikace)

IoC..... Inversion Of Control (návrhový vzor)

REST ..... Representational State Transfer (architektura rozhraní)

API..... Application Programming Interface (rozhraní pro programování aplikace)

HTTP ..... HyperText Transfer Protocol (síťový protokol pro přenos dat)

URI..... Uniform Resource Identifier (jednotný identifikátor zdroje)

SQL ..... Structured Query Language (dotazovací jazyk relační databáze)

JSON..... JavaScript Object Notation (způsob zápisu dat nezávislého na platformě)

XML..... eXtensible Markup Language (značkovací jazyk k ukládání a přenosu dat)

JDK ..... Java Development Kit (vývojářská sada pro vývoj v jazyce Java)

JSP..... Java Server Pages (technologie pro podporu dynamického obsahu stránek)

POM ..... Project Object Model (soubor popisující strukuru projektu v technologii Maven)

WAR..... Web application ARchiv (archiv webové aplikace v jazyce Java)

OGM ..... Object Graph Mapping (technologie pro mapování POJO na entity grafu)

D3 ..... Data-Driven Document (JavaScript knihovna pro vizualizaci dat v HTML)

Obsah

1 Úvod

Tato bakalářská práce se věnuje představení jednotlivých technologií a metodik, souvisejících s vývojem webových aplikací, postupné analýze, návrhu a implementaci aplikace pro významnou firmu na poli pojišťovnictví. Tato firma si v relační databázi vede data o interakci jednotlivých aplikací, v rámci integrační platformy firmy. Data v této databázi však nejsou velice přívětivá formátem, jelikož některé aplikace jsou v záznamu o interakci vedeny jen pod číslem jejich systému. Také se může stát, že se v datech některé údaje nebudou vyskytovat, jako například právě číslo systému, či název jedné z aplikací v interakci. Především jde však o to, že tento způsob vedení dat je z hlediska získávání informací o vztazích mezi aplikacemi pro firmu neefektivní, kvůli způsobu, jakým relační databáze ukládá data, a faktu, že těchto interakcí je velké množství a s narůstajícím počtem dat, narůstá také doba provedení dotazu.

Za tímto účelem vznikla tato práce, jejímž cílem je vytvořit webovou aplikaci, která data z relační databáze podle určité logiky převede do databáze grafové. Grafová databáze je založena na vztazích a díky tomu se stala ideální volbou pro tento typ problému. Způsob, jakým grafová databáze pracuje s daty za nás také řeší problém s rychlostí dotazování na data. Zmíněný převod, je žádoucí provádět jednou za měsíc a jelikož se jedná o opakující se činnost, v implementaci se pokusíme o automatizaci tohoto procesu. Logika převodu by měla zajistit, aby měl výsledný vztah informaci o celkovém počtu využití. Jakmile se budou data již nacházet v grafové databázi, je potřeba tyto data poskytnout a to způsobem, který mohou využít jiné aplikace, nehledě na jazyk, ve kterém jsou napsány, a proto by měla aplikace vystavovat rozhraní, které data poskytne. Nakonec se v práci pokusíme o využití dat z grafové databáze k vizualizaci grafu v prohlížeči, pro přehled struktury integrační platformy.

Výsledná aplikace pomůže firmě poskytnout přehledná data o interakci aplikací a celkové struktuře integrační platformy. Díky celkovému počtu využití u jednotlivých vztahů bude také vidět, na jakých uzlech je největší zátěž a díky tomu lze předejít výpadkům.

1.1 Cíl práce a metodika práce

V bakalářské práci jsou postupně představeny technologie a pojmy související s vývojem webových aplikací. Jednotlivé pojmy by měli být srozumitelné i mírně pokročilým uživatelům, kteří se informačními technologiemi nezabývají každý den. Cílem této bakalářské práce je implementace webové aplikace, která převede data z relační databáze do databáze grafové a nad daty z této databáze vystaví webové rozhraní, které data poskytne v univerzálním formátu nezávislém na jazyce aplikací, jež budou rozhraní využívat.

1.2 Teoretická východiska

V této části bakalářské práce se seznámíme s teorií kolem webových aplikací, jejich účelem a architekturou. Představíme si zde jednotlivé knihovny, frameworky a techniky, které se v souvislosti s webovými aplikacemi používají. Velkou část zde věnujeme popisu relačních a grafových databázích. Vysvětlíme si, jakým způsobem jsou vedeny údaje v relační databázi, jak jsou definovány vztahy v této databázi a jaké existují typy vztahů. V kapitole o grafové databázi si řekneme, jaké jsou výhody oproti relačním databázím, jak se dotazujeme na data, představíme si dotazovací jazyk Cypher a vysvětlíme si, jakým způsobem funguje. Nakonec se seznámíme s REST API a fungování HTTP protokolu.

2 Webová aplikace

Na internetu je stále více a více informací a postupně stoupají nároky na reprezentaci a práci s těmito daty. Pro jednoduchou informační činnost postačí obyčejné HTML stránky s pěkným stylováním, ovšem pokud bychom chtěli například kontaktní formulář, nebo pracovat s databází a případně dynamicky generovat obsah na základě dat z této databáze, statický dokument již nebude stačit a je potřeba serverová logika. Webová aplikace se zjednodušeně řečeno dělí na front-end neboli to, co je vidět (výsledná dynamicky vygenerovaná HTML stránka se styly, skripty apod.) a back-end (to, co nevidíme, serverová část, která provádí výpočty, logiku s daty a vrací výsledky na front-end).

V dnešní době se pro webové vývojáře stal velmi populární front-end aplikace, kde kraluje Javascript se svými frameworky a knihovny, které nabízejí řešení nejrůznějších problémů pomocí svých architektur a přístupů k programování front-end části. Je to natolik rozšířené a vyvinuté, že co se týče generování dynamického obsahu, ale například i validace vstupů, hodně práce se přesunulo sem. Serverová část má ovšem stále své opodstatnění, slouží pro složité výpočty, práce s databázemi, komunikaci s jinými webovými službami a mnoho dalšího.

2.1 Architektura MVC

Jedná se o vzor návrhu architektury aplikace, který rozděluje aplikaci na tři logické celky (komponenty), kterými jsou: **m**odel, pohled (**v**iew) a kontrolér (**c**ontroller), odtud tedy MVC. Každá z těchto komponent řeší určité specifické úlohy a snaží se tak oddělit logiku aplikace od uživatelského rozhraní. Díky tomu lze aplikaci dále snadněji rozšiřovat. Aktuálně je jedna z nejčastěji používaných architektur, především u webových aplikací. (Tutorials Point: Basic MVC Architecture. c2016)

2.1.1 Model

Jedná se o největší a nejdůležitější komponentu z MVC. Představuje logiku související s daty, s nimiž uživatel pracuje. Model je tedy zodpovědný za veškerou správu dat a chování aplikace. Často je kvůli své mohutnosti v aplikaci dále dělen na vrstvy, jako například doménová a servisní vrstva. Většina webových aplikací potřebuje pro svou práci databázi. V takovém případě pro nás model zapouzdřuje data z databáze do objektů, se kterými dále pracujeme.

2.1.2 Pohled (View)

Má na starost reprezentaci dat, představuje uživatelské rozhraní aplikace, se kterým uživatel interaguje. V případě webových aplikací, představuje pohled dynamickou webovou stránku. Jde tedy o stránku v HTML se styly v CSS. Dynamika stránky pak vzniká pomocí Javascriptu a využitím šablonovacích systémů. Šablonovací systém umožňuje vložit do HTML proměnné, které se mění na základě dat z modelu. Těchto systémů je spousta a každý z nich nabízí jiné možnosti. Příkladem takovýchto systémů mohou být např.: Thymeleaf, Twig, Mustache a další, nebo, v případě jazyku Java, technologie JSP.

2.1.3 Kontrolér

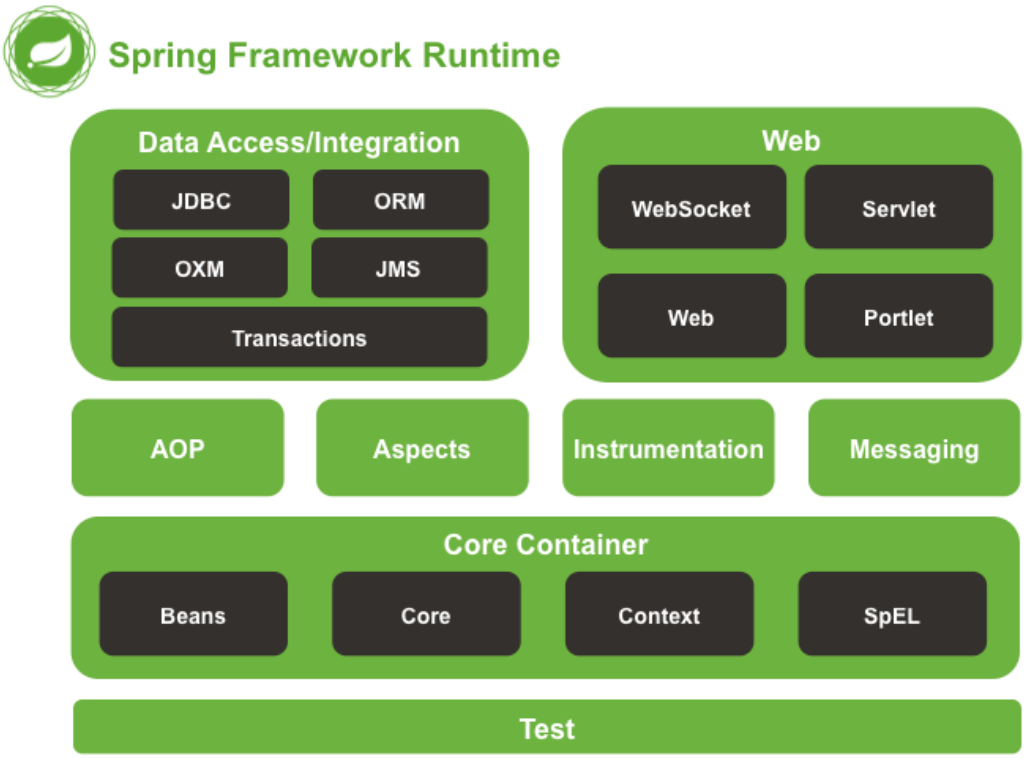
Jak napovídá název této komponenty, kontrolér se stará o řízení. Reaguje na události z pohledu vyvolaných uživatelem, na základě kterých manipuluje s modelem.
Celý cyklus zpracování požadavku v architektuře může vypadat následovně.
Příklad editace dat osoby v architektuře MVC:

1. Uživatel vytvoří událost: edituj osobu s ID 1, pomocí uživatelského rozhraní
2. Tuto událost zachytí kontrolér. Většinou k provedení požadavku potřebuje určité informace z pohledu, v našem případě se jedná o nové údaje o osobě a ID osoby.
3. Poté vyvolá na modelu funkci editace osoby s dostupnými informacemi a model provede vše potřebné pro editaci údajů, například změnu údajů v databázi osob.
4. Kontrolér převezme nová data od modelu a předá je pohledu, který je zobrazí uživateli.

2.2 Spring

Jedná se o framework, který poskytuje rozsáhlou podporu pro vývoj Java aplikací, zejména z hlediska infrastruktury. Java jako platforma postrádá prostředky pro sestavení základních částí programu do jednoho funkčního celku a tento nedostatek doplňuje právě Spring. (Spring: Introduction to Spring. c2010-2016)

Celý Spring Framework je velmi obsáhlý a je rozdělen do několika modulů.



Obrázek 1 Přehled nad Spring Frameworkem [zdroj: Dokumentace Spring]

Podrobné popsání jednotlivých modulů není pro tuto práci podstatné, budeme se tedy věnovat popisu pouze pro nás zajímavých modulů. V této práci využijeme moduly Core, Web, Data Access a Test. Specifické využití těchto modulů bude postupně popsáno v práci tak, jak na ně budeme narážet. První, čím začneme, bude konfigurace Spring IoC kontejneru. K tomu je potřeba seznámit se s Dependency Injection.

2.2.1 Dependency Injection a Inversion of Control

Dependency Injection obecně, je konkrétní technikou návrhového vzoru Inversion Of Control, jejím hlavním cílem je oprostit komponenty od vytváření či získávání svých vlastních závislostí. Namísto toho se závislosti vytvoří na jiném místě a komponentám se dodají. Pomocí tohoto návrhu vznikne kód, který je znovupoužitelný (komponentě můžeme podsunout vždy jiný objekt neboli závislost), čitelný (vidíme všechny závislosti komponent) a je také dobře testovatelný.

2.2.2 Spring IoC kontejner

Tímto se dostáváme ke kontejneru Spring Inversion Of Control. Tento kontejner je právě tím místem, na kterém se nacházejí závislosti jednotlivých

komponent. Tyto závislosti jsou objektům předávány pouze jako atribut konstruktoru, anebo pomocí setteru. Kontejner pak tyto závislosti objektům podsune při vytváření bean. Proces je tedy obrácený, proto Inversion Of Control.

Základními balíčky jsou pro Spring IoC kontejner context a beans, jenž jsou součástí Core modulu. Rozhraní, která nás z těchto balíků zajímají jsou BeanFactory a ApplicationContext. BeanFactory poskytuje konfigurační framework a základní funkcionalitu. V podstatě poskytuje inicializaci a napojování bean. ApplicationContext obsahuje veškerou funkcionalitu BeanFactory a navíc za nás obstarává ještě další funkce. Obecně je doporučeno pro konfiguraci používat ApplicationContext. (Spring: The IoC Container. c2010-2016)

2.2.3 Bean

V textu jsme již párkrát zmínili tento název a nyní si přiblížíme, co pro Spring znamená bean. Jak je uvedeno v dokumentaci Spring (c2004-2016): *„Když vytváříte definici beany, to, co doopravdy vytváříte je návod, jakým vytvořit aktuální instance třídy definované v této definici beany. Myšlenka, že definice beany je návod je důležitá, jelikož to znamená, že stejně jako u třídy můžete mít z jednoho návodu vytvořeno potencionálně více objektových instancí.“* Pozn.: překlad autora.

V definici beany je možné nejen konfigurovat závislosti a hodnoty při konfiguraci instance objektu, ale také scope objektů vytvořených z této definice bean. V základu může být beana vytvořena jedním ze dvou typů scope (celkem jich existuje 7 ale 5 z nich je možné použít jen při použití web-aware ApplicationContextu). Tyto dva základní jsou singleton a prototype. (Spring Framework Documentation. c2004-2016)

Singleton

V případě, že má bean tento scope, říkáme tím Spring kontejneru, že při každém odkázání se na tuto beanu, si přejeme vrátit vždy jednu a tu samou sdílenou instanci beany. Spring kontejner tedy vytvoří přesně jednu instanci objektu, definovaného v bean definici. Tato instance je cachována a při každém dalším dotazu na tuto beanu vrátí kontejner tuto instanci. Tento scope není potřeba v definici beany specifikovat, jelikož je defaultní. (Spring Framework Documentation. c2004-2016)

Prototype

Jedná se o opak singleton, a to v tom smyslu, že při každém dotazu na vytvoření instance beany, je vrácena nová instance této beany. Pravidlem, podle kterého je doporučeno se rozhodovat při volbě scope, je používat prototype scope pro stavové beany a singleton pro bez-stavové. S veškerou konfigurací kontextu a vytvoření definice bean, se podrobněji seznámíme v praktické části při implementaci. (Spring Framework Documentation. c2004-2016)

2.2.4 Aplikační kontext

Aplikační kontext obsahuje definice jednotlivých bean, které jsou využity v aplikaci. Spring framework umožňuje dva typy konfigurace. Pomocí XML souborů a pomocí Java konfigurace. Je možné tyto dva způsoby zkombinovat a tím získat třetí způsob ale ten je dobré použít, jen pokud nelze jinak. Je však dobré vědět, že tu ta možnost je. Oba způsoby mají jisté specifikace, a oba mají své zastánce, a proto nemusí být ani jeden z nich špatná volba, a i dnes se v aplikacích používají oba způsoby.

Java konfigurace

V aplikaci, kterou se tato práce zabývá, budeme provádět konfiguraci frameworku pomocí Java konfigurace. Důvodu jsou pro nás zřejmé:

- Java je typová a množství chyb může odhalit již kompilátor.
- Možnost ladění (debugging) kódu.
- Java kód je přehlednější, vyhledávání a refactoring jsou mnohem snazší.

Samotná Java konfigurace je postavená na anotacích a dříve jsme si řekli, že pro aplikační kontext se doporučuje použít ApplicationContext ze Springu. To ovšem platí pouze pokud je použita konfigurace pomocí XML. Pro případ použití Java konfigurace nabízí Spring třídu AnnotationConfigApplicationContext a pro webové aplikace AnnotationConfigWebApplicationContext. Nejčastěji používané anotace pro nás budou @Configuration a @Bean.

Pokud použijeme anotaci @Configuration na třídě, říkáme tím, že Spring IoC kontejner může třídu použít jako zdroj definice bean. Třídy anotované jako konfigurační pak budou registrovány do aplikačního kontextu. Anotace @Bean se používá na úrovní metody a značí, že metoda vytváří a konfiguruje instanci objektu,

která bude spravována Spring IoC kontejnerem. Je také potřeba zmínit anotaci @Profile, která naznačuje, že komponenty se budou registrovat pouze v případě, že je aktivní jeden z uvedených profilů. Této anotace využijeme pro odlišné sestavení testovacího a produkčního kontextu aplikace. (Spring Framework Documentation. c2004-2016)

2.2.5 Spring MVC

Výše v práci jsme se seznámili s architekturou MVC. Spring jako jednu z mnoha svých možností nabízí realizaci této architektury pomocí Spring Web MVC Frameworku. Tento framework je navržen kolem DispatcherServletu, který deleguje požadavky na kontroléry. Celkový průběh zpracování požadavku je následující.

Dispečer přijme požadavek a deleguje jej na kontrolér, který má správné mapování požadavku, ten jej zpracuje a vrátí dispečerovi vygenerovaný model. Dispečer předá model šabloně, která za použití dat z modelu „vykreslí“ odpověď pro prohlížeč a navrátí kontrolu dispečerovi. Ten poté pouze vrátí odpověď.

Tento DispatcherServlet je potomkem třídy HttpServlet a je třeba mu nadefinovat URL mapování. Toto mapování říká, že pokud na tuto URL přijde požadavek, je práce tohoto dispečeru, se o něj postarat. Toto nastavení je deklarováno ve web.xml. Dispečer dále deleguje požadavek na kontroléry. Ve Spring MVC se, v případě konfigurace annotation driven, kontrolér definuje pomocí anotace na levelu třídy a tou je @Controller. Ta pro Spring značí, že daná třída je v roli kontroléru. Dispečer skenuje třídy s touto anotací a detekuje jejich @RequestMapping anotace. Tato anotace se používá na úrovni metody i třídy a mapuje URL adresy na třídu (kontrolér), či metodu. Nejčastěji je použito tak, že tato anotace na úrovni třídy odděluje požadavky podle vyšší úrovně URL a jednotlivé metody mají pak mapování konkrétních HTTP požadavků se specifikovanými HTTP metody. Tyto konkrétní požadavky často obsahují parametry, identifikující například ID položky, a ty lze z požadavku dostat pomocí anotací jako @RequestParam, @PathVariable a @ModelAttribute. Například kontrolér v e-shopu pro práci s produktem, s metodou pro odstranění produktu, by mohl vypadat následovně.

```
@Controller
@RequestMapping("/product")
public class ProductController {
```

```
        @RequestMapping("/remove/{id}")
        public String remove(@PathVariable Long productId) {
            //odstranění produktu z databáze
            return "productView";
        }
}
```

Dispečer by na tento kontrolér delegoval požadavek, pokud by byl požadavek směrován na URL končící /product/remove/ a navíc obsahoval nějaké ID. Např.: www.eshop.cz/product/remove/10.

Ve výše uvedeném příkladu je návratovou hodnotou řetězec „productView“. Spring MVC kontroléry musí vrátit hodnotu typu String, View nebo ModelAndView. V našem případě tato hodnota označuje název pohledu, který bude vybrán pro zobrazení v odpovědi. K vyhodnocení toho, který pohled bude vrácen, využívá Spring ViewResolver, který nabízí několik implementací. Ve výše uvedeném příkladu by byl použit UrlBasedViewResolver pro JSP, který má definovaný URL prefix a sufix pohledu. Například pro konfiguraci, kde prefix je „/WEB-INF/jsp/“ a sufix „.jsp“, by ViewResolver vyhodnotil URL pohledu na „/WEB-INF/jsp/productView.jsp“ a předal by tento pohled dispečerovi pro zobrazení. V této práci použijeme view resolver specifický pro Thymeleaf a s jeho konfigurací se setkáme v praktické části, nicméně bude fungovat na stejném principu skládání URL. (Spring: Web MVC framework, c2004-2016)

2.3 Databáze

V dnešní době je téměř nemožné narazit na aplikaci, jenž by nepotřebovala pracovat s dlouhodobými daty. Tuto persistenci dat zaručují databáze a díky tomu se stávají nedílnou součástí takovýchto aplikací.

Podle Lacka (2003, s. 25) *“Databázi chápeme jako uložiště údajů, které jsou uloženy a zpracovány nezávisle na aplikačních programech. Databáze zapouzdřuje jednak vlastní údaje, ale i relační vztahy mezi jednotlivými prvky a objekty v databázi, schémata popisující struktury údajů a integritní omezení.”*

Důležitým pojmem kolem databází je SQL (Structured Query Language). Jde o dotazovací jazyk, který slouží k manipulaci s daty v relační databázi. Relační databáze však není jediný typ databáze, jenž existuje, a z toho důvodu se databáze dělí na dva hlavní typy, a to SQL a NoSql (Not only SQL). Rozdíl je především ve způsobu, jakým jsou data v databázi vedena. Zatímco v relačních databázích jsou data uložena v tabulkách, v NoSQL instanci se může jednat o dokument, graf či mapu hodnot. Mezi ty nejznámější však patří databáze relační. (Databáze: datové sklady, OLAP a dolování dat, c2003)

2.3.1 Relační databáze

V této práci se budeme věnovat nejvíce databázím grafovým, z toho důvodu si představíme pouze základní vlastnosti, které je potřeba kolem relačních databází, pro porozumění této práci, znát.

SQL

Je složen z klauzulí, ve kterých se specifikují parametry dotazu. Tyto klauzule jsou děleny do kategorií podle jejich využití.

Pro manipulaci s daty:

SELECT, **DELETE**, **UPDATE** a **INSERT**

Pro definici struktury:

CREATE, **DROP**, **ALTER**.

Za každým z těchto následuje název tabulky, nad kterou se má dotaz provést.

Tabulka

Data v relační databázi jsou založena na databázovém relačním modelu, který je tvořen tabulky. Tabulka je základní dvourozměrná struktura, tvořena sloupci a řádky, kde řádek představuje samotný záznam a sloupce pak definují, o jaký typ dat se jedná.

Tabulka 1 Ukázka tabulky relační databáze
[zdroj: vlastní]

Jméno	Příjmení	PSČ
Lukáš	Zoubek	286 01

Je ihned jasné, že sloupce Jméno a Příjmení by zde byli definovány jako řetězec znaků (nvarchar, text či jiný, dle typu databáze), avšak sloupec PSČ může být jak řetězec znaků, tak při odstranění mezery jej lze ukládat jako čistý číselný údaj o délce 5 znaků. Momentálně se může zdát, že na tom příliš nezáleží, ale v momentě, kdy definujeme sloupec jako řetězec, umožňujeme do sloupce vložit i jiné znaky než čísla a do tabulky by tedy bylo možné uložit PSČ, které neobsahuje číslice vůbec žádné, či dokonce vložit prázdný údaj NULL, a to nás přivádí k integritním omezením.

Integritní omezení

Tato omezení jsou pravidla, která zajišťují konzistenci databáze. Tato omezení říkají, jaká data lze do tabulky vložit na základě jejich datového typu a maximální definované délky. Mohou také definovat chování propojených záznamů v případě smazání jednoho z nich.

Vztah v relační databázi

Vztah v relační databázi je definován pomocí primárního a cizího klíče. Jak uvádí Lacko ve své knize Databáze: datové sklady, OLAP a dolování dat (2003, s27) *„Primární klíč je jednoznačný identifikátor každého záznamu. Může to být sloupec, případně kombinace více sloupců, které slouží pro jednoznačnou identifikaci každého řádku tabulky. Hodnota pole primárního klíče musí být v rámci tabulky jedinečná. Pole primárního klíče musí obsahovat konkrétní hodnotu, tedy nesmí nikdy nabýt hodnoty NULL.“* Na druhé straně vztahu je pak cizí klíč. Opět definice podle Lacka *„Cizí klíč je sloupec, případně kombinace více sloupců, které jsou propojeny na primární klíč v jiné tabulce.“* V relační databázi existuje více typů vztahů. (Databáze: datové sklady, OLAP a dolování dat, c2003)

Vztah jedna k jedné

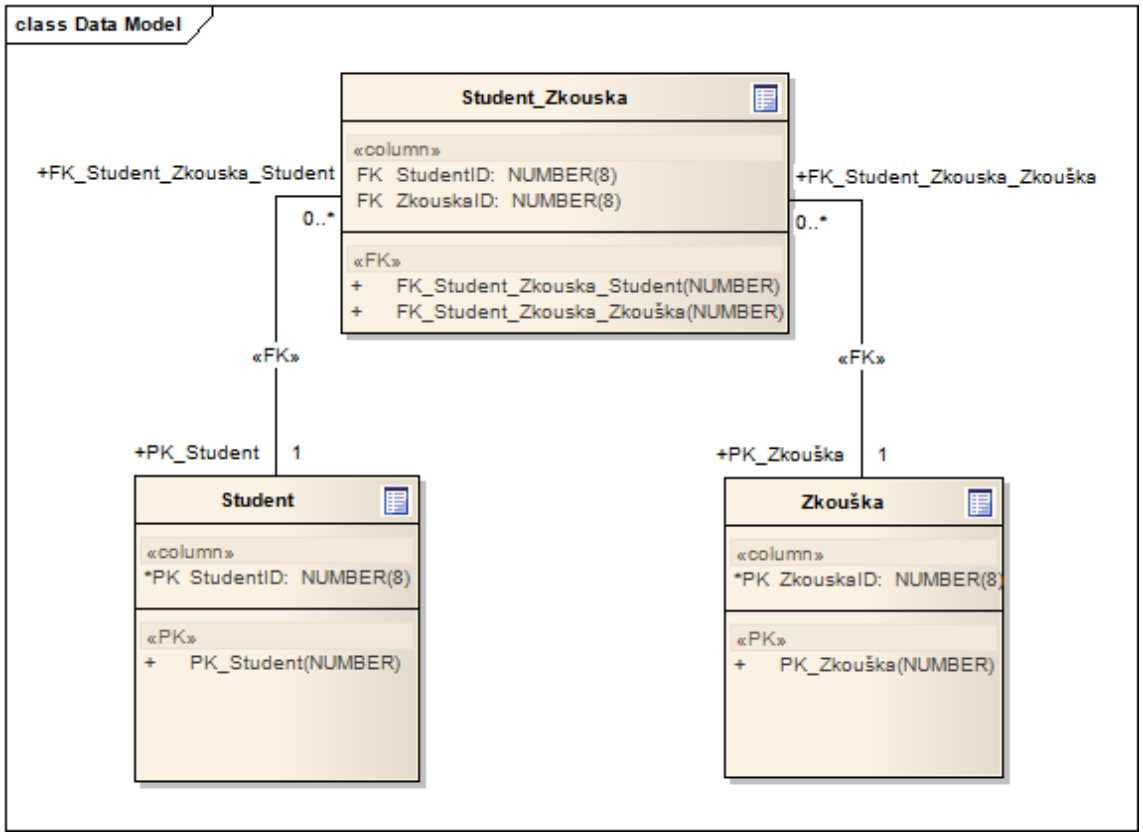
Jedná se o vztah, kdy pro záznam v první tabulce, existuje maximálně jeden záznam v tabulce druhé. Pro každý záznam s privátním klíčem v první tabulce tedy může existovat maximálně jeden záznam v tabulce druhé.

Vztah jedna k mnoha

Tento vztah se vyskytuje nejčastěji a znamená, že jeden záznam v tabulce může být spojen s jedním až několika záznamy v tabulce druhé. Například vztah Osoby a Automobilu. Jedna osoba může vlastnit více automobilů.

Vztah mnoho k mnoha

Více záznamů v první tabulce, může být spojeno s více záznamy druhé tabulky. Tomuto vztahu se však v databázi snažíme vyhnout a pokud se v modelu databáze vyskytne, provádíme dekompozici na dva vztahy jedna k mnoha vytvořením třetí tabulky. Na následujícím obrázku vidíme dekompozici vztahu Studenta a Zkoušky, kde Student může mít více zkoušek a na zkoušce může být více studentů. (Databáze: datové sklady, OLAP a dolování dat, c2003)



Obrázek 2 Dekompozice M:N [zdroj: vlastní]

2.3.2 Grafové databáze

Pro práci s grafovou databází není potřeba podrobná teorie grafu. Pro základní modely a využití potenciálu databáze, nám stačí obecné znalosti o grafech. Pro osvěžení paměti si však připomeňme, jak si obecně definujeme graf.

Graf je tvořen z vrcholů a hran, kde vrcholy představují nějaké entity a hrany přestavují vztahy, jenž tyto entity mezi sebou mají. Velikou výhodou toho je, že entitou může být cokoliv, a to nám umožňuje modelovat různé struktury a typy grafů. Je možné namodelovat součásti strojů, dopravní infrastrukturu, velké využití se objevuje v sociálních sítích, medicíně.

Tím však výhody databáze nekončí. Další z nich je, že grafy vyhovují způsobu, kterým abstrahujeme detaily určité domény. Výsledný náčrt abstrakce, často představuje kroužky, či čtverečky a mezi nimi čáry pro vyznačení vztahu, což je blízké tomu, co modelujeme uvnitř databáze.

Pro grafové databáze jsou nejpřirozenější vztahy mezi entity. U ostatních databází tomu tak není, jak je vidět na přechozím příkladu s relační databází na obrázku 2, kde se vztah mezi entity musel modelovat pomocí primárního a cizího klíče. (Graph Databases, c2015)

Výkon dotazu

Tento fakt se projevuje na výkonu při dotazu na propojená data. Oproti relačním databázím, kde dotaz založený na join se zhoršuje s tím, jak roste objem dat, má grafová databáze tendenci držet si konstantní výkon nehledě na objem dat. Toto dokáže díky tomu, že dotaz je proveden pouze nad částí grafu, která splňuje požadavky dotazu. Nikoliv nad celým grafem. (Graph Databases, c2015)

Flexibilita

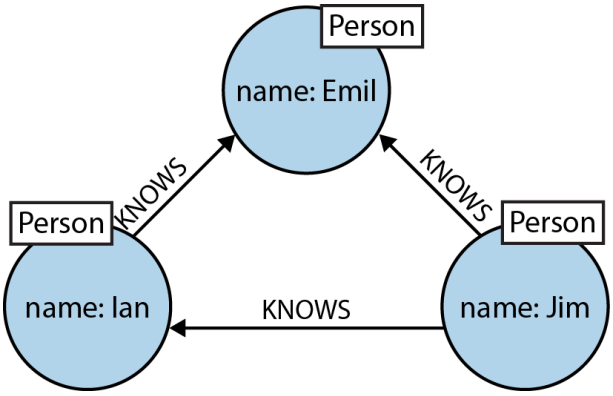
Další z výhod je flexibilita. V grafu je možné promítnout abstrakci naší domény, jak jsme již zmínili, což je pohodlnější způsob pro navrhování modelu. Mnohem flexibilnější vlastností je však přirozená aditivita grafu, což znamená, že je možné na již existující model přidat nové uzly, vztahy a popisy, a přitom se tím nezničí funkcionalita dotazů, což nám umožňuje méně správy a údržby systému v průběhu vývoje. (Graph Databases, c2015)

Cypher

Jak uvedli autoři knihy Graph Databases Robinson Ian et al. (2015, s. 27) *"Cypher je expresivní (přesto kompaktní) dotazovací jazyk grafové databáze."* Je to nejsnazší dotazovací jazyk pro grafové databáze. Tvoří tak skvělý základ pro seznámení se základy grafů. "*Jeho jednoduchost použití spočívá ve skutečnosti, že je v souladu se způsobem, kdy intuitivně popisujeme grafy za použití diagramů."* Pozn.: překlad autora.

Co si pod tímto tvrzením představit. Řekněme, že chceme nalézt data odpovídající určitému vzoru. Nejjednodušší reprezentace tohoto vzoru pro nás bude nakreslení diagramu. Cypher je založena na tom, vyhledat data na základě určitého vzoru. Pro názornou ukázku použijeme příklad z knihy Graph Databases.

Obrázek 3 Ukázka vzoru grafu [zdroj: Graph Databases (2015, s. 28)]



Dotaz v Cypher na tento vzor pak vypadá takto:

(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)

V dotazu jsou identifikátory vrcholů **emil**, **jim** a **ian**. Mezi těmito vrcholy je jejich vztah, jehož směr je vyznačen znakem ostré závorky **>** nebo **<**. Mezi hranaté závorky, hned za dvojtečku píšeme název vztahu, zde se jedná o typ vztahu **KNOWS**. Díky tomu, že jazyk je zpracováván z leva, je možné vyjádřit kružnici zopakováním vrcholu emil. (Graph Databases, c2015)

Podobně jako SQL, je Cypher sestaven z klauzulí. Tou nejčastější je MATCH následovaná klauzulí RETURN.

Příklad MATCH pro nalezení vzájemných přátel lidí s Jimem vypadá takto:

MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),

(a)-[:KNOWS]->(c)

RETURN b, c

V klauzuli MATCH je vrchol typu Person, který má vlastnost name s hodnotou „Jim“ uložen do identifikátoru „**a“**, přes který se dále v dotazu na tento vrchol odkazuje.

RETURN pak obsahuje identifikátory, které se mají vrátit jako odpověď. Může se jednat jak o vrcholy, relace tak i vlastnosti. V tomto případě se tedy vrátí přítel Jima **b**, a jejich společný přítel **c**.

Cypher má ještě další klauzule, v této práci se však nebudeme zabývat všemi a představíme si pouze ty potřebné. První takovou je WHERE, kterou nejspíše není nutno nějak dopodrobna představovat. Stejně jak je tomu v SQL se v této klauzuli definují kritéria dotazu. Je však důležité zmínit, že v Cypher se údaje ve WHERE používají jako upřesnění vzoru v MATCH a ne jako filtr na předem dotazovaný vzor.

MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)

WHERE a.name = 'Jim'

RETURN b, c. (Graph Databases, c2015)

2.4 REST API

REST (Representational State Transfer) je architektura rozhraní. Je použit pro přístup ke zdrojům, který umožňuje pomocí HTTP metod, kde zdroje jsou identifikovány pomocí URI.

Webové aplikace pracují se spoustou dat a často mohou ke své funkci vyžadovat také data z jiných aplikací. Tyto aplikace však mohou být napsané v jiném jazyce a fyzicky nasazené na jiném serveru. Je tedy potřeba nalézt způsob, kterým data mezi aplikacemi předat. Takovým řešením jsou webové služby a rozhraní.

Webové služby umožňují předat informace přes univerzální notaci, nezávislou na jazyce aplikací. Nejčastěji se jedná o XML a JSON i když v poslední době se volí spíše JSON, jelikož není tak robustní, je přehlednější pro člověka a dobře zpracovatelný a generovaný stroji. (RESTful web services, c2007)

2.4.1 JSON

Jde o zkratku znamenající JavaScript Object Notation. Struktura je založena na hodnotě a klíči a stylem odpovídá tedy například mapám v Javě. Například JSON pro objekt osoba s atributy jmeno a prijmeni by vypadal takto.

```
{"osoba": {
 "jmeno": "Lukáš",
 "prijmeni": "Zoubek"
 }}
```

2.4.2 HTTP

HTTP (Hypertext Transfer Protocol) je internetový protokol, který byl vytvořen pro přenos HTML dokumentů mezi server a prohlížečem. Nyní je však používán i pro přenos jiných typů souborů, a to díky rozšíření MIME pro specifikaci typu dokumentu. Protokol funguje na principu dotaz - odpověď. Celý dotaz sestaví obálku, která se pošle serveru. Obálka má hlavičku, HTTP metodu, cestu (URI) a tělo. (RESTful web services, c2007)

Ukážeme si, jak vypadá hlavička dotazu, jenž obsahuje metadata, například jeden z dotazů na stránku: www.uhk.cz.

RequestHeaders
GET /cs-CZ/UHK HTTP/1.1
Host: www.uhk.cz
User-Agent: Mozzila/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Encoding: gzip, deflate, sdch, br

V hlavičce se nachází několik informací, na výše uvedeném příkladu jsme si ukázali jen ty důležité. Těmi nejhlavnějšími pro nás jsou:

Metoda - v tomto dotazu se jedná o metodu GET. Protokol má hned několik metod, kde název metody většinou představuje akci, kterou od serveru požadujeme. Nejčastěji používanými metody jsou GET, POST, DELETE, PUT.

Dále je zde URI (Uniform Resource Identifier, neboli jednotný identifikátor zdroje), ten je uveden hned za metodou, v tomto případě se jedná o /cs-CZ/UHK. Vzhledem k uvedenému serveru www.uhk.cz, je tedy dotaz GET na adresu www.uhk.cz/cs-CZ/UHK. Za URI je definována verze protokolu - HTTP/1.1.

User-agent, nese informaci o prohlížeči, z něhož byl dotaz proveden. Ve vlastnosti Accept je výčet MIME typů, které od serveru příjmem a v Accept-encoding je kódování obsahu. Tělo obálky je v tomto případě prázdné, jelikož se jedná o GET požadavek. Odpověď, kterou dostaneme vrací HTML dokument nazvaný UHK a parametry hlavičky jsou následující.

Response Headers
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/8.5
Set-Cookie: VisitorStatus=21060214238;
Date: Sun, 23 Oct 2016 19:18:30 GMT
Content-Length: 86593

Důležitý je HTTP status, ten je v tomto případě 200, tedy OK. Existuje celý číselník HTTP stavů. Každý má svůj kód a popis. Mezi neradostně známé patří například chybový stav 404 Not Found. Content-Type označuje typ souboru, který byl vrácen a tím je text/html s kódováním utf-8. Dále vidíme název serveru, čas odpovědi a délku obsahu. V těle obálky se nachází samotná data představující dokument UHK.html. Informace z hlavičky jsou opět abstrahované pro zjednodušení ukázky. (RESTful web services, c2007)

3 Analýza

V první části této kapitoly provedeme analýzu požadavků, které jsou kladeny na aplikaci. V druhé části se seznámíme s integrační platformou, vysvětlíme, co to integrační platforma je a připomeneme si jakou roli hraje v naší aplikaci. Bude zde blíže představena relační databáze, jenž bude představovat zdroj dat pro naší aplikaci. Nakonec provedeme analýzu mapování těchto dat na data grafu.

3.1 Analýza požadavků

Aplikace je tvořena pro firmu a je třeba, aby splňovala firmou zadané požadavky. Tyto požadavky jsou rozděleny dle analýzy požadavků na funkční a nefunkční. Funkční požadavky udávají, jak by se měla aplikace chovat, jaké funkce splňovat, zatímco nefunkční, udávají požadavky na výkon, spolehlivost, technologie a podobně. Na základě schůzí s konzultantem za firmu vyplynuly následující požadavky.

3.1.1 Funkční požadavky

- Převod dat z tabulky relační databáze do grafové databáze
- Provedení převodu automaticky za určitou periodu, pomocí plánovače.
- Převod provádět měsíčně
- Dynamické chování v případě neznámé aplikace či metody.
- Poskytnutí dat z grafové databáze přes REST službu
    - Poskytnutí seznamu metod poskytovaných aplikací v IP
    - Poskytnutí seznamu metod konzumovaných aplikací v IP
    - Získání počtu volání metody provedené konzumující aplikací.
    - Poskytnutí seznamu aplikací v grafové databázi

3.1.2 Nefunkční požadavky

- JAVA 7
- Architektura MVC
- Grafová databáze Neo4j
- REST rozhraní
- Apache Tomcat 7, 8

3.2 Analýza integrační platformy

Jak plyne z požadavků, aplikace by měla shromažďovat data o interakci aplikací v rámci integrační platformy firmy a převádět je do grafové databáze. V této sekci se budu věnovat představení struktury integrační platformy, aplikací se kterými budeme pracovat a analýze dat, která budou tvořit zdroj informací pro grafovou databázi.

3.2.1 Integrační platforma

Integrační platforma je software tvořený integrací různorodých aplikací a služeb. Tyto aplikace mezi sebou komunikují a navzájem využívají svých funkcionalit, avšak na venek se tváří jako jeden celistvý software. V této práci bude integrační platformu představovat 25 existujících systémů.

Tabulka 2 Seznam systému a služeb integrační platformy [zdroj: dokumentace firmy]

ID Systému	Název systému	ID Systému	Název systému
0	IBM MB	14	czgcarpricing
1	czgusrmng	15	czgprodcatalog
2	czgucinfo	16	czgagentinfo
3	czgpolinfo	17	czgearnix
4	czgpolicing	18	czgsearch
5	czgmessaging	19	czgcomm
6	czgcalcbm	20	czgleadmng
7	czgdelivery	21	czgblacklisting
8	czgprints	22	czgcontractimportpmv
9	Czgda	23	czgprocesssupport
11	czgagentinfo	24	czctia
12	czgclaims	99	monitoring
13	czgcodelist		

Interakce systémů

Tyto systémy, jak jsme již naznačili, spolu navzájem komunikují. Většinou se jedná o využití služeb integrační platformy, tedy jedné z 25 aplikací, systémem ze zbylých externích. Dochází k tomu tak, že aplikace v integrační platformě poskytují webové služby, které umožňují využít jejich chování, a tyto služby jsou ostatními aplikacemi konzumovány. Pro lepší pochopení, této interakce si představme příklad: Aplikace czgprints může sloužit pro vygenerování PDF dokumentu, poskytovala by tedy metodu, kterou lze pro tuto akci využít a ta se jmenuje createDoc. Aplikace, kterou si pro příklad nazveme Konzument_A, má určitou funkcionalitu, ve které shromažďuje data a v určité fázi svého výkonu, potřebuje data vygenerovat do PDF. Konzument_A zná systém czgprints a jeho metodu createDoc, pro dosažení požadovaného výsledku je potřeba jen tuto metodu zavolat a předat jí data k vygenerování. Pokud vše proběhlo v pořádku, czgprints vygeneruje PDF a vrátí jej aplikaci Konzument_A.

Tabulka MESSAGE

Jelikož je potřeba tyto interakce sledovat, byla vytvořena Oracle databáze, která vede tabulku MESSAGE, jenž obsahuje veškerá data o těchto interakcích mezi systémy. Každá jedna interakce, provolání, využití metody, je představováno jedním řádkem v této tabulce. Vždy, když některý systém využije metody jiného systému, v této tabulce je o tom záznam. V předchozím příkladu, ve kterém jsme si popsali interakci mezi systémy Konzument_A a czgprints, je celý cyklus hodně zestručněn a abstrahován o nedůležité. To ovšem neplatí pro tabulku MESSAGE, která obsahuje velké množství informací o této konzumaci metod, z tohoto důvodu zde představím pouze sloupce, které nás zajímají a které budeme využívat v aplikaci.

Jedná se o sloupce v následující tabulce.

Tabulka 2 Abstrakce struktury tabulky Message [zdroj:vlastní]

Název sloupce	**Typ sloupce**	**Popis**
REQUEST_TIME	timestamp	přesný čas, kdy byla metoda volána
RESPONSE_TIME	timestamp	přesný čas odpovědi na požadavek
APPLICATION	varchar2	název aplikace, která poskytuje konzumovanou metodu
MSG_TYPE	varchar2	název konzumované metody

MSG_VERSION	Number	verze metody
MSG_SRC_SYS	Number	systemID v seznamu systému, patřící volající aplikaci
MSG_TAR_SYS	Number	také systemID, ale systému, jenž poskytuje metodu
EXCEPTION	varchar2	výjimka jenž mohla nastat při volání

Data z těchto sloupců budou sloužit jako zdroj pro data grafové databáze. Aplikace, jenž metodu poskytuje, je ve zprávě uvedena jako **application**, název této metody je **msg_type** a aplikace, jenž jí konzumuje je zde vedena pod jejím systémovým ID v parametru **msg_src_sys**. Podle tohoto mapování bude provedena implementace konverze. Je potřeba zmínit, že tabulka obsahuje velké množství záznamu, řádově desítky milionů, a proto databáze automaticky jednou za měsíc provádí archivaci tabulky. Archivovaná tabulka má prefix „A_“ a sufix ve formátu „_YYYYMM“, tedy například pro prosinec 2016 by se tabulka jmenovala A_MESSAGE_201612. Na to bude brán zřetel při implementaci.

4 Návrh

V této kapitole se hned v první části obeznámíme s návrhem modelu grafové databáze. Vytyčíme entity, které se budou v databázi nacházet a k nim určíme jejich atributy. V druhé části si předvedeme základní strukturu naší aplikace, popíšeme jednotlivé třídy a rozhraní a vysvětlíme, jaké role budou v aplikaci zastávat.

4.1 Návrh modelu grafové databáze

Aktuálně víme, co je to grafová databáze, máme k dispozici zdrojová data z databáze relační a nyní nás čeká navrhnout podobu samotného grafového modelu. Jak uvádí Ian Robinson a spoluautoři knihy Graph Databases (2015, s. 27) „*Proces modelování může být nejlépe popsán jako pokus o vytvoření grafové struktury, která vyjadřuje otázky, na které se chceme ptát naší domény.*“ Pozn.: překlad autora. Celý postup vytvoření modelu shrnuje autor do pěti bodů, nejdůležitějšími jsou však tyto:

1. Popsat cíle klienta, které očekává od našeho modelu
2. Přepsat tyto cíle do otázek na naši doménu
3. Z těchto otázek určit entity a relace

Podstatná jména z vět by měla přestavovat jednotlivé popisy. Slovesa pak popisují vztah mezi těmito uzly. Vlastní jména představují jednotlivé instance, což pro nás znamená vrcholy grafu.
Tento postup vytváření domény je určitě vhodné použít pro objemné návrhy s velkým množstvím vrcholů a vztahů různých typů, kde nemusí být na první pohled jasné, jak by měl model vypadat. V případě, že bychom se řídili těmito postupy, bod jedna máme již díky analýze požadavků splněn. Víme totiž, že požadavky na grafovou databázi, tedy cíle klienta, jsou:

- Rozlišit, která aplikace má jaký vztah, s kterou metodou. Tento vtah může být buďto aplikace metodu konzumuje nebo aplikace metodu poskytuje.
- Je také velmi důležité, aby bylo možné zjistit, kolikrát celkem, již tato konzumace proběhla.
- Informovat o tom, jaké metody aplikace poskytuje, případně konzumuje.

Z těchto požadavků vznikají otázky:

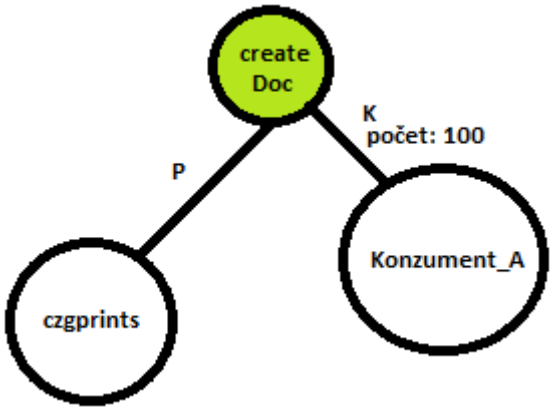
- Jaký má aplikace vztah s metodou?

- Které všechny metody poskytuje aplikace?
- Které všechny metody konzumuje aplikace?
- Kolikrát celkem aplikace konzumovala metodu?

Otázky nám jasně mluví o Entitách: aplikace a metoda, které mezi sebou mají vztah konzumuje či poskytuje. Z aplikací a metod se tedy stávají popisy (labels) a jednotlivé instance budou představovat konkrétní metody a aplikace. Mezi těmito instancemi pak bude vztah konzumuje či poskytuje, kde směr vztahu bude vždy od aplikace k metodě. (Graph Databases Robinson Ian et al. c2015)

4.1.1 Model grafové databáze

Toto je více než dostačující pro první náčrt grafu, který si ukážeme na již zmíněném příkladu s aplikacemi czgprints, Konzument_A a metodou createDoc, jenž jsou již konkrétními instancemi entit Aplikace a Metoda.



Obrázek 4 Základní model grafové databáze [zdroj: vlastní]

Z ukázky je vidět, že relace, „K“ (konzumuje), má na rozdíl od vztahu „P“ (poskytuje), navíc vlastnost, která říká, kolikrát byla daná metoda aplikací konzumována a tím tedy nejen splňuje požadavek, ale také se stává o trochu složitějším vztahem, což se promítne dále při samotném psaní kódu aplikace.

Co se týče vlastností jednotlivých entit grafu, je nutné přidat identifikátory aplikací, takové, jaké jsou potencionálním uživatelům našeho výsledného REST API známé. Chci tím říci, že sice budeme mít nějaké grafové ID entity, ale to není dopředu známé, dokud jej sami nezjistíme dotazem, a pro uživatele je to úplně neznámá informace. Přesto ale potřebujeme nějaký jednoznačný identifikátor, kterým bude možné v metodách kontrolérů přistoupit ke správné instanci. Naštěstí víme, že ke každé aplikaci existuje nějaké unikátní systém ID, a to nám je již předem známo z číselníku systémů. U entity aplikace nám tedy k názvu aplikace přibude ještě atribut systemId.

Shrneme-li si tedy jaké vlastnosti má která entita dostaneme:

- Aplikace – název, ID systému, kolekce poskytovaných metod, kolekce konzumovaných metod
- Metoda – název a verze
- Vztah „konzumuje“ – aplikaci, metodu a celkové využití

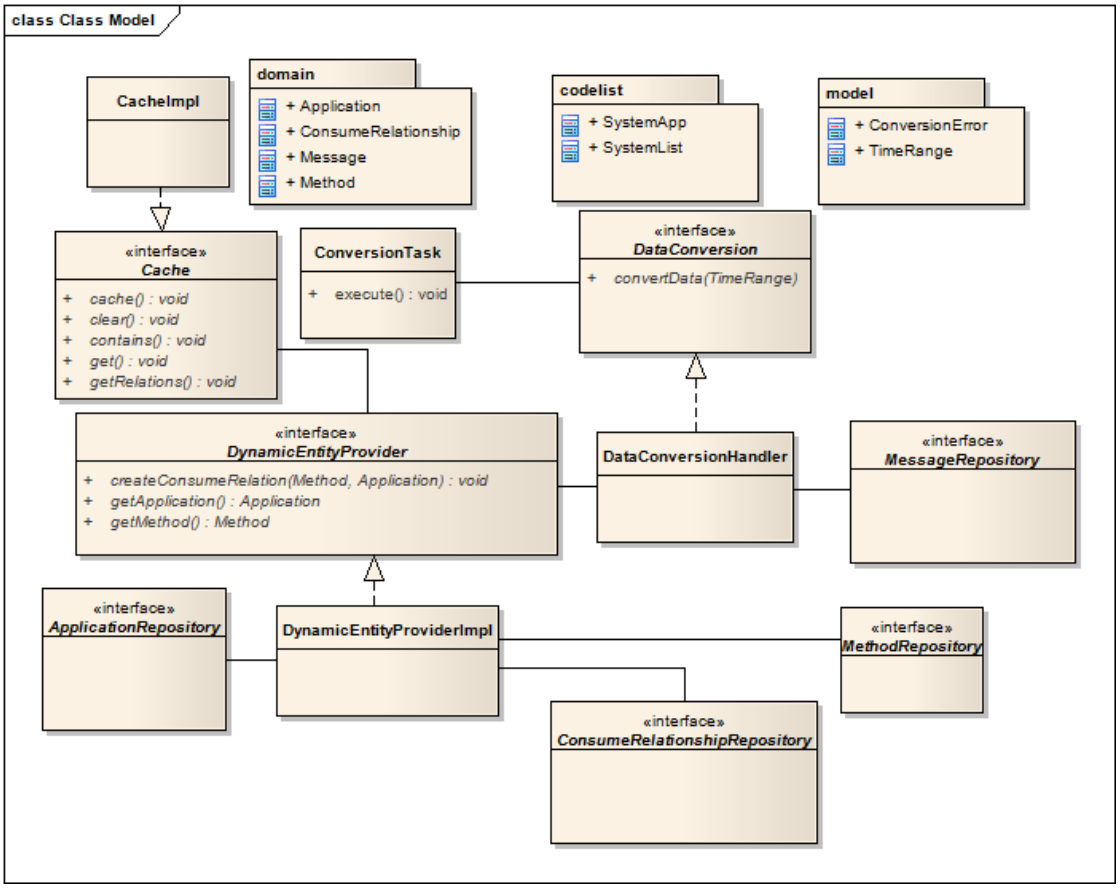
Abychom zaručili, že v grafu budou všechny tyto aplikace se systém ID již od začátku, bude potřeba jako první vytvořit strukturu grafu pomocí číselníku systémů. Dále bude nutné vymyslet práci s tímto číselníkem do budoucna, jelikož se může stát, že některý systém se vyloučí nebo naopak přibyde.

4.2 Návrh aplikace

V této fázi máme dokončenou analýzu požadavků na aplikaci, máme k dispozici informace o struktuře tabulky relační databáze, jenž pro nás bude hlavním zdrojem dat a v minulé kapitole jsme navrhli model grafové databáze. Nyní se tedy můžeme přesunout k návrhu samotné aplikace.

Hlavním jádrem aplikace bude jakýsi převaděč, který bude mít za úkol provádět konverzi dat z relační databáze do databáze grafové. Konverze bude spočívat ve získání dat z relační databáze podle určitých kritérii a zpracování těchto dat do podoby, ve které tato data uložíme do grafu. Jelikož jeden záznam o interakci odpovídá jednomu řádku v relační databázi, je potřeba procházet vstupy po řádcích. Celý tento proces konverze bude spouštěn automaticky, pomocí plánovače. V případě, že se objeví nový systém, vztah, či metoda, je potřeba implementovat dynamické chování, které zaručí, že se data bez chyby převedou. Hlavním cílem, o který se při implementaci budeme snažit, je převést data o interakci do grafové databáze ať jsou jakákoliv, a tím se nemyslí převést vše bez ohledu na chyby, ale nevyhodit chybu konverze jen v případě, kdy grafová databáze nezná interakci či entitu, o které je v relační databázi zmínka. Toto automatické vytváření entit do grafové databáze může být na škodu, pokud v tom nebude přehled, bylo by tedy dobré mít určitý náhled na seznamem entit v grafu anebo určitým způsobem informovat o vytvoření nové entity.

Vezmeme-li toto shrnutí funkcionality k vytvoření návrhu aplikace, můžeme začít rozpoznávat jednotlivé třídy a jejich metody, které se budou v aplikaci nacházet. Pro přehledné zobrazení návrhu si vytvoříme jednoduchý diagram tříd pomocí nástroje Enterprise Architect.



Obrázek 5 Návrh modelu tříd [zdroj: vlastní]

Na obrázku výše je vidět předběžná struktura, podle které by mohla být aplikace navržena. Nejprve je třeba říci, že návrh není hlavním cílem a tento řídký diagram tříd slouží pouze pro vizuální představu struktury aplikace. Jsou zde pouze třídy specifické pro konkrétní problém aplikace a třídy jako kontroléry nebo konfigurační třídy jsou tedy vynechány a popsány budou níže v praktické části při konfiguraci Spring frameworku. Při vytváření názvů tříd jsme se řídili tím, že jednou z největších chyb vývojáře je šetřit slovy, a že názvy by měly být samo popisné a srozumitelné. Z toho důvodu jsou některé názvy delší.

Třídy jsou pro přehlednost a jejich společnou logickou úlohu rozděleny do balíčků. Na příkladu výše vidíme balíčky **domain**, **codelist** a **model**. Třídy a rozhraní z návrhu, které se starají o hlavní služby aplikace, přidělíme také do jednoho balíčku a pojmenujeme jej **service**. Další balíček se bude jmenovat **repository** a bude seskupovat všechny repositáře z návrhu. Bylo by také správné oddělit třídy **cache** do stejnojmenného balíku. Výslednou strukturu aplikace si pro přehlednost ukážeme později.

4.2.1 Databázová doména a repositáře

Podíváme-li se na balíček domain, vidíme čtyři třídy: **Application**, **ConsumeRelationship**, **Method** a **Message**. Všechny tyto třídy modelují entity specifické pro naši problematickou doménu. První tři zmíněné třídy představují entity grafové databáze, a třída Message odpovídá jednomu záznamu o interakci z databáze relační. Pojmenování třídy znamenající interakci systémů jako „zpráva“, se může zdát trochu matoucí. Tento název je však takový, jelikož daná interakce ve skutečnosti představuje zprávu zaslanou mezi systémy a relační databáze eviduje záznamy v tabulce s názvem Messages, vznikal by tedy zbytečný zmatek, pokud bychom pojmenovali jinak.

Tyto třídy budou mít naimplementované privátní atributy a jejich veřejné selektory a modifikátory, přes které se bude s atributy manipulovat. Dodržíme tak jeden ze základní principů Objektově Orientovaného Modelování, kterým je Zapouzdření.

Jelikož se jedná o databázové entity, budeme chtít instance těchto tříd perzistovat, upravovat je, dotazovat se na data a k tomu nám poslouží repositáře. Ke každé třídě je vytvořen jeden repositář a ten má na starost práci s databází pro danou třídu. V základu je potřeba, aby měl každý repositář minimálně CRUD (Create, Read, Update, Delete) operace. To znamená, že například ApplicationRepository bude umět vytvořit, číst, pozměnit a smazat Application a obdobně u ostatních. Samozřejmě, pro určitou funkcionalitu aplikace bude potřeba provést specifický dotaz a například repositář MessageRepository bude mít trochu jiné metody, než repositáře grafových entit a zde postačí pouze metoda pro čtení. Na to vše je potřeba brát zřetel před tím, než se začneme zabývat implementací.

4.2.2 DataConversionHandler

Jak bylo řečeno dříve, základním kamenem aplikace je převaděč, konkrétně z návrhu za tuto funkcionalitu zodpovídá DataConversionHandler. Návrh funkcionality této komponenty je zamýšlen tak, že tato třída se stará o získání dat z relační databáze a jejich převod na data databáze grafové. Omezení dotazu je provedeno objektem TimeRange, který obsahuje počáteční a konečné datum intervalu ze které budou data získána. Jelikož jeden řádek v relační databázi představuje jeden záznam o interakci systémů, je potřeba zařídit, aby se tato data převáděla po řádcích. To se z počátku nemusí zdát jako velký problém, je ovšem

třeba myslet na to, že produkční databáze může obsahovat miliony až desítky milionů záznamů a je nutné si implementaci získávání těchto dat promyslet.

Třída je spojena pouze s repositářem MessageRepository, který obstarává zprávy o interakci a s třídou DynamicEntityProvider. Tento návrh je zamýšlen tak, že DataConversionHandler se stará pouze o získání dat a převod. Jediné, co by pro svou práci měl potřebovat, je zpráva obsahující informace o konzumentovi, poskytovateli a metodě, které převede na jim odpovídající databázové entity, a následně u nich vytvoří vztah. Problém nastává v případě, kdy se ve zdrojových datech objeví metoda, či aplikace, která není v grafové databázi a jelikož cílem převodu je uchovat informace o jakékoliv interakci, i té doposud neznámé, je nutné navrhnout dynamické vytvoření těchto entit. Toto dynamické do-vytváření grafového modelu již úplně nesouvisí se samotnou konverzí a bylo by dobré, aby se DataConversionHandler o tento problém nemusel starat, ale delegoval jej někomu jinému. Přesně pro toto chování byl navržen DynamicEntityProvider.

4.2.3 DynamicEntityProvider

Stará se o veškeré komplikace a alternativní scénáře kolem získání grafových entit. Zaručuje se tím, že na každý požadavek o získání metody, či aplikace, která je v aktuálním vztahu interakce, tuto entitu poskytne, ať už jejím získáním z databáze, či vytvořením. K tomu je poskytovatel spojen s repositáři grafových entit. Poskytnutí entit bude při normálním chování vypadat tak, že na žádost handleru o entitu, se poskytovatel přes repositář doptá do grafové databáze a pokud existuje, vrátí jí handleru. Při tomto chování je důležité myslet na to, že tato akce bude probíhat pro každý záznam, a jak jsme si již řekli, těchto záznamů může být až desítky milionů za měsíc, a takové doptávání se do databáze je velmi náročné na výkon a v tomto počtu by šlo o extrémní zpomalení. Navíc v modelu grafu budou vždy jen desítky aplikací a stovky metod a v takovém množství záznamů, se budou často opakovat dotazy na stejnou entitu, a to není vůbec praktické. Pro řešení tohoto problému jsme si navrhli systém cachování grafových entit, který je přiblížen níže.

Aby bylo použití poskytovatele z pohledu DataConversionHandleru jednoduché a handler nebyl zatěžován vnitřním chováním poskytovatele, je jeho funkcionalita vystavena přes rozhraní, pod kterým se provider zavazuje, že dané metody implementuje.

4.2.4 Cache

Z textu u DynamicEntityProvidera víme, jaké jsou důvody návrhu rozhraní Cache a třídy CacheImpl, která toto rozhraní implementuje. Hlavní princip spočívá v uchování již dotazovaných entit z databáze v paměti RAM, se kterou je práce mnohem rychlejší než se samotnou databází. Poskytovatel entit tak vždy před dotazem do databáze nejprve nahlédne do cache, zda neobsahuje tuto entitu a pokud ano, nemusí se na nic dotazovat, a za použití správné datové struktury v cache vrátí entitu s konstantní asymptotickou složitostí. Pokud v cache nebude, dotáže se na ní a do cache jí přidá, tím se po načtení většiny entit do RAM bude pracovat téměř bez potřeby dotazování se do databáze a celý převod se tak několikanásobně zrychlí.

4.2.5 Conversion Task

Tuto třídu pravděpodobně není nutné nijak dopodrobna rozebírat. Jedná se o třídu, která představuje úlohu, jenž bude plánována časovačem a zaručí automatický převod dat. Z toho důvodu využívá rozhraní DataConversion, jehož implementací je DataConversionHandler. Důvody rozhraní jsou obdobné jako ve vztahu handleru s providerem. U této časované úlohy je potřeba při implementaci myslet na to, že čas převodu bude muset mít možnost volit hodinu, den, popř. den v měsíci. Více se touto třídou budeme zabývat při implementaci.

4.2.6 SystemList a SystemApp

V předchozí kapitole jsme si vysvětlili potřebu určitým způsobem pracovat s číselníkem systémů. K tomu v aplikaci bude sloužit třída SystemList, která bude představovat celý číselník aplikací SystemApp, které evidují název a systémové ID systému. Tento seznam bude představovat celkovou strukturu grafové databáze co se týče aplikací.

Myšlenka je taková, že před každým převodem se načte číselník. Zprávy (Message) se budou procházet a na základě ID systému, nebo jeho názvu, se vybere systém z číselníku a poté se bude pokračovat v konverzi. Pokud se však v datech nachází vztah se systémem, jenž není v číselníku, je i tak potřeba tento vztah uložit, a proto tento systém vytvoříme a uložíme do číselníku. Pokud se tak stane pro daný systém znovu, již bude obsažen v číselníku a bude s ním zacházeno podle klasického scénáře.

Číselník bude ukládán a načítán ze souboru CSV, který bude možné upravovat. Bude se tak moci přidat systém a jeho ID a předejde se tak automatickému přidání v kódu.

Je důležité podotknout, že SystemApp je navržena jako systém číselníku, nemá tedy při konverzi stejný význam jako aplikace (Application). Ta se bere jako entita grafu, jenž je v něm přítomna.

5 Implementace

V práci jsme se doposud seznámili s teoretickými základy ohledně webových aplikací, provedli analýzu požadavků a zdrojových dat relační databáze a dle těchto požadavků jsme provedli návrh grafové databáze a samotné aplikace. Nyní se dostáváme do konečné fáze a tou je implementace aplikace. Implementace se bude skládat z několika kroků, ve kterých postupně projdeme a vysvětlíme konfiguraci sestavení aplikace, správu závislostí, způsob, jakým vypisovat události aplikace a získat tak informace o aktuálním dění a implementaci samotné funkcionality. Budeme se zde také velmi stručně zabývat testováním napsaného kódu.

5.1 Konfigurace aplikace

Návrh struktury aplikace jsme si již přiblížili a nyní se můžeme konečně přesunout ke konfiguraci aplikace. Jak již víme z analýzy požadavků, bude se jednat o webovou aplikaci v jazyce Java. Při vývoji aplikací v tomto jazyce, máme k dispozici vývojářskou sadu, kterou využíváme pro psaní kódu aplikace. V tomto případě ze sady použijeme knihovny Java7. I přes svou obsáhlost, nám však tato knihovna nebude stačit a bude třeba využít knihoven třetích stran. Těchto knihoven využijeme v aplikaci například pro připojení do databáze, mapování doménových entit na entity databázové, testování kódu, a mnoho dalšího. Je tedy jasné, že těchto knihoven bude hodně a je třeba mít nad jejich správou a způsobu integrace knihoven do projektu, určitou kontrolu.

Především integrace knihoven je stěžejní, jelikož klasický způsob by pravděpodobně spočíval ve stažení všech potřebných knihoven ručně ze stránek poskytovatele těchto archivů, nakopírování do složky k projektu a následné připojení pomocí vývojového prostředí, v němž je projekt tvořen. Tento způsob je v celku pracný a do budoucna, v případě požadavku na povýšení verze jedné z knihoven, nepraktický.

Při vývoji aplikace, je také třeba rozlišovat kdy se použijí jaké knihovny. V produkčním provozu na serveru nepotřebujeme knihovny použité pro testování na lokálním stroji. Na základě prostředí, je třeba také spravovat jaké zdroje budou součástí výsledného archivu aplikace, v produkčním balíku není potřeba mít zdrojové soubory pro testování, testovací konfigurační soubory a další.

Vše doposud zmíněné, se může zdát jako obtížně zvládnutelný úkol, a už to může vývojáře odradit od vývoje své aplikace. Naštěstí existují technologie, které

nám vše výše zmíněné umožňují přehledně spravovat. Řeč je o nástrojích pro správu sestavení aplikace. V případě Java aplikací je možné vybrat například mezi Ant, Maven či Gradle a dalšími. Seznámení s vlastnostmi těchto nástrojů a jejich vzájemné porovnání by dalo na samotnou práci, či minimálně článek a jelikož by pro účely sestavení aplikace, kterou se tato práce zabývá, splnil účel kterýkoliv z těchto nástrojů, nebudu se zde zabývat jejích porovnáním. Zvoleným nástrojem pro naši aplikaci je Maven.

5.1.1 Maven

Jak jsme si již v úvodu kapitoly řekli, jedná se o nástroj pro správu aplikace a jejího sestavení. Je založen na konceptu objektového modelu projektu (project object model-POM). (Apache Maven, c2002-2016)

POM

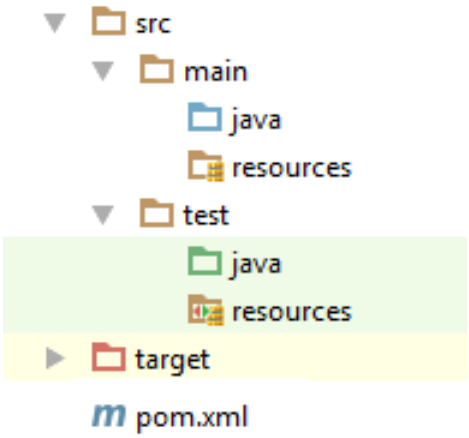
Jedná se o XML soubor, který obsahuje veškerou konfiguraci a závislosti, které jsou potřeba pro sestavení projektu. Nachází se v hlavním adresáři projektu. Minimalistické POM má následující podobu:

```
<project>
    <modelVersion>4.0.0</modelVersion> //verze objekt. modelu
    <groupId>cz.zoubelu</groupId>   //id skupiny
    <artifactId>struct</artifactId> //libovolný název artefaktu
    <version>1.0-SNAPSHOT</version>
</project>
```

Vzorec: groupId:artifactId:version, tvoří přesný kvalifikátor artefaktu. V případě naší aplikace, bude artefakt kvalifikován jako: cz.zoubelu:struct:1.0-SNAPSHOT. Dále jsou tu věci jako balení (packaging) aplikace, které je defaultně nastaveno na JAR. Náš požadavek je balení do archivu WAR, musíme tedy mezi element <project> přidat následující řádek:

```
<packaging>war</packaging>
```

Maven projekt má specifickou strukturu, co se týče rozdělení zdrojů. Konkrétně *src/main/java* pro zdrojové kódy aplikace a *src/main/resources* pro zdroje, jenž aplikace využívá. Dále obdobně pro testy *src/test/java* a *src/test/resources*. (Apache Maven, c2002-2016)



Obrázek 6 Ukázka struktury Maven projektu [zdroj: vlastní]

Na obrázku 6, je vidět navíc adresář *target*, který obsahuje Mavenem vygenerované soubory, které jsou výsledkem sestavení (classes, balík WAR/JAR, atd.). Definice celé této základní struktury Maven projektu, je definována v Super POM. (Apache Maven, c2002-2016)

Super POM

Jedná se o základní POM Mavenu. Každé POM rozšiřuje toto Super POM, pokud není nastaveno jinak. Stejně jako u POM se jedná o XML soubor, v tomto jsou však definovány základní nastavení Mavenu na základě jeho verze. Nachází se zde konfigurace centrálního repositáře, repositáře pluginů, také samotné pluginy, které definují chování základních příkazů Mavenu (install, clean, site), a další.

Příkazy

Maven nahlíží na projekt jako na artefakt a jeho sestavení a distribuce je řízena životním cyklem sestavení. Maven je založen na třech cyklech: *default*, *clean* a *site*. Cyklus *default* se stará o nasazení projektu, *clean* o čištění a *site* vytváří dokumentaci k projektu. Celý životní cyklus default je složen z několika fází, které jsou při použití default cyklu spuštěny jeden po druhém. Sekvence příkazů je následující: *validate*, *compile*, *test, package, verify, install, deploy*. Nejpoužívanějším příkazem je příkaz *install*. Tento příkaz spustí všechny příkazy v pořadí před ním a poté vytvoří a nainstaluje artefakt do lokálního repositáře. (Apache Maven, c2002-2016)

Repositáře

V Mavenu se repositářem myslí uložiště sestavených artefaktů a závislostí. Repositář může být pouze lokální či vzdálený. Lokální repositář, je automaticky, či

ručně vytvořený adresář na lokálním disku, kam se ukládají všechny stáhnuté artefakty a sestavené projekty.

V případě vzdálených repositářů se může jednat o jakékoliv jiné uložiště artefaktů, na které se přistupuje pomocí protokolů např.: file:// a http://. Může představovat interní repositář firmy s artefakty, které jsou přístupné pouze interně anebo se může jednat o repositář, který veřejně poskytuje své artefakty k využití.

Pokud v projektu definujeme závislost na artefakt, který se nenachází v lokálním repositáři, je ve výchozím nastavení artefakt stažen z http://repo.maven.apache.org/maven2/. (Apache Maven, c2002-2016)

Závislosti

Mnoho vývojářských prostředí má možnost integrace s Mavenem, díky tomu lze jednoduše vytvořit nový projekt a typická Maven struktura je i společně s pom.xml vygenerována automaticky. V případě existujícího projektu je možné začít spravovat knihovny projektu. Pro vysvětlení integrace knihoven třetích stran do projektu, si ukážeme integraci knihoven spring-core, spring-web a spring-mvc, o kterých víme, že určitě použijeme, do projektu.

Tyto knihovny jsou označovány za závislosti, proto je potřeba v pom.xml přidat element, který bude obsahovat seznam těchto závislostí. Jedná se o element <dependencies>. Přidáme tedy do našeho pom.xml.

```
    <dependencies>
        //... závislosti
    </dependencies>
</project>
```

Nyní je potřeba vyplnit námi chtěné spring-core, spring-web a spring-mvc závislosti. Tyto závislosti získáme z centrálního repositáře Mavenu. Nejjednodušší způsob, jak nalézt dostupné verze artefaktu, je navštívit například stránku https://mvnrepository.com/. Na této stránce se nachází seznam nejrůznějších artefaktů z několika repositářů, rozdělený do kategorií, s informacemi o stahování a popularitě artefaktu.

Na této stránce stačí vyhledat artefakt, zvolit verzi, kterou chceme ve svém projektu a níže na stránce pod popisem knihovny zvolit záložku maven a zkopírovat XML kód do našeho pom.xml mezi elementy dependencies. Výsledek tedy pro všechny závislosti s verzí 4.2.5.RELEASE vypadá takto:

```
<dependencies>
    <!-- SPRING -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
```

```
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>

    <properties>
        <spring.version>4.2.5.RELEASE</spring.version>
    </properties>
```

V příkladu jsme si verze u jednotlivých závislostí extrahovali do vlastnosti `<spring.version>`, která je v elementu properties. Název vlastnosti je volitelný. Učinili jsme tak, jelikož artefakty patří do stejné skupiny springframework a časem bude pravděpodobně zapotřebí verzi změnit, ať už kvůli kompatibilitě s jinými knihovnami, nebo jen pro povýšení na vyšší verzi obsahující nějaké opravy či záplaty a v takovém případě stačí změnit pouze na jednom centrálním místě. Nemluvě o přehlednosti. S rostoucím projektem bude narůstat počet závislosti až na desítky a pokud nás bude zajímat, jaká verze knihovny se používá, stačí pouze nahlédnout do vlastností.

K samotným závislostem. Je pravděpodobné, že knihovny, které do svého projektu budeme takto vkládat, sami potřebují nějaké závislosti. Abychom nemuseli pracně hledat a vkládat závislost závislosti téměř do nekonečna, existuje v Mavenu vlastnost nazvaná Transitivní závislosti, která za nás toto řeší. V podstatě za nás Maven dohledá potřebné závislosti a vloží k projektu automaticky. Úrovně závislostí, kam až se můžeme dostat jsou neomezené, a proto se může stát, že se do projektu dostanou závislosti, které nechceme, jsou zbytečné, či je potřebujeme v jiné verzi. Pro omezení závislostí existují proto tyto funkce: Správa závislostí, Vyloučené závislosti, Volitelné závislosti, Rozsah závislosti a Zprostředkování závislosti.

Nebudeme se zde dlouze rozepisovat k jednotlivým funkcím a pouze si přiblížíme ty, jenž jsou důležité a dále používané v tomto projektu.

Rozsah závislosti

Neboli Dependency Scope, určuje, jak bude se závislostí zacházeno a kdy bude použita. Existuje 6 typů:

- compile – výchozí, dostupné ve všech classpath v projektu, propagovány do závislých projektů
- provided – podobně jako compile, ovšem zde se očekává, že závislost bude poskytována z JDK či webového kontejneru. Není transitivní.
- runtime – závislost požadována pouze za runtime, není dostupná z kompilační classpath
- test – závislost je požadována pouze při testování
- system – stejně jako provided, ovšem zde je potřeba specifikovat JDK explicitně
- import – pouze pro závislost typu pom specifikovanou v elementu <dependencyManagement>

V projektu použijeme především test a provided, zbytek závislostí nebude mít rozsah závislosti specifikován, bude tedy v compile.

Vyloučené závislosti

Anglicky Excluded Dependencies, používá se v případě, kdy máme závislost na nějaký projekt, artefakt, a ten má závislost na jiný projekt, který my v našem projektu nechceme. Například v našem případě jsme přidali závislost na spring-core, který má závislost na projektu commons-logging. V budoucím vývoji projektu tento artefakt potřebovat nebudeme, a proto jej můžeme vyloučit.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

V projektu, jenž je cílem této bakalářské práce nebude tolik závislosti, abychom se museli bát o výslednou velikost balíku, a proto nebude nutné procházet artefakty a vylučovat nepotřebné závislosti.

Sestavení aplikace

Již jsme si řekli, že Maven je nástroj, jenž řídí sestavení výsledného balíku aplikace. Způsob, jakým bude sestaven, se blíže specifikuje v pom.xml v elementu <build>. Tento build může být podmíněn profilem, například pro produkční balík nebudeme chtít zahrnovat různé properties soubory, jelikož je chceme vyjmout

někam do složky na serveru, aby bylo možné jejich hodnoty měnit, bez potřeby znovu sestavit a nasadit war archiv. Lze tedy specifikovat produkční build, který konfigurační soubory vyjme a tento build se spustí, jen pokud při spuštění příkazu specifikujeme název profilu do argumentu. Například pro instalaci může vypadat takto: *mvn install -P nazevProfilu.*

Dále je možné určit pluginy, které se při sestavení použijí, což se nám velice hodí, jelikož jak víme, pro spuštění naší aplikace budeme potřebovat Apache Tomcat 7, který je jednoduché stáhnout a instalace ani konfigurace není nijak náročná, avšak, je tu způsob, kterým to jde ještě snáze. Do sestavení si přidáme plugin tomcat7, který umožní spuštění aplikace na Tomcat serveru.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.2</version>
            <configuration>
                <path>/</path>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Tento plugin vytvoří vestavěnou instanci Apache Tomcat serveru, na které se automaticky nasadí naše aplikace a vystaví jí na lokální adrese. Element <path> specifikuje na jaké přesně URL bude aplikace dostupná, v tomto případě bude dostupná přímo na localhost:8080/. Zajímavé a velmi vítané pro vývoj je na tomto pluginu to, že ve výchozím nastavení nepoužívá jako zdroj aplikace soubory ze složky target, ale přímo zdroje ze složky src. To znamená, že aplikace běží dynamicky a změna například v HTML či CSS stylech bude viditelná okamžitě.

5.2 Konfigurace Logování

V této fázi máme připraveny závislosti aplikace společně s konfigurací sestavení war archivu. Od této doby začneme konfigurovat jednotlivé frameworky a implementovat chování aplikace. Abychom měli přehled o tom, co se v aplikaci děje a věděli, že vše funguje, jak má, popřípadě mohli efektivněji reagovat na chyby, pokud tomu tak nebude, nakonfigurujeme si logování. Spring v základu používá Jakarta Commons Logging API (JCL), který používá algoritmus vyhledávání jiných logovacích frameworků na známých místech na classpath a snaží se použít ten nejvhodnější. Bohužel, tento algoritmus je v celku problematický. Z toho důvodu v naší aplikaci použijeme Simple Logging Facade for Java (SLF4J), který je za runtime efektivnější, jelikož nevyhledává jiné frameworky, ale používá compile-

time vázání. Umí přemostit mezi jednotlivými logovacími frameworky a umožní nám tak použít jeden centrální s jednou konfigurací. To, co nastavíme je, že přemostíme Spring logování na SLF4J a poté poskytneme vázání na Log4j. První krok k tomuto přechodu jsme si již provedli při vyjmutí commons-logging závislosti z artefaktu spring-core v kapitole 4.1.1 v sekci Vyloučené závislosti. (Spring Framework Documentation, c2004-2016)

Dalším krokem, je přidání závislostí na artefakty slf4j-api, slf4j-log4j12, log4j a jcl-over-slf4j. Poté, co máme tyto jar na classpath, je potřeba poskytnout konfiguraci pro log4j. Defaultně se framework snaží tuto konfiguraci najít na classpath v podobě log4.properties, anebo log4j.xml souboru. Vytvoříme si tedy log4j.properties soubor v adresáři src/main/resources/ a zároveň připravíme i pro test do src/test/resources/. Obsah souboru je následovný.

```
log4j.rootLogger=info, stdout, conversionDetailLog,conversionErrorLog

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
%-5p %c{1}:%L - %m%n

# Logování do souboru
log4j.appender.conversionDetailLog=org.apache.log4j.DailyRollingFileAp
pender
log4j.appender.conversionDetailLog.File=${catalina.base}/logs/neo.log
log4j.appender.conversionDetailLog.layout=org.apache.log4j.PatternLayo
ut
log4j.appender.conversionDetailLog.datePattern='.'yyyy-MM-dd
log4j.appender.conversionDetailLog.layout.ConversionPattern=%d{yyyy-
MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n


# Logování error logů do souboru - v tomto souboru by se měly nacházet
pouze chybové zprávy
log4j.appender.conversionErrorLog=org.apache.log4j.DailyRollingFileApp
ender
log4j.appender.conversionErrorLog.File=${catalina.base}/logs/neo-
error.log
log4j.appender.conversionErrorLog.datePattern='.'yyyy-MM-dd
log4j.appender.conversionErrorLog.Threshold=error
log4j.appender.conversionErrorLog.layout=org.apache.log4j.PatternLayou
t
log4j.appender.conversionErrorLog.layout.ConversionPattern=%d{yyyy-MM-
dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Pro Log4j definujeme čtyři „loggery“: info, stdout, conversionDetailLog a conversionErrorLog. První nemá bližší definici a znamená to, že logování bude zobrazovat úroveň INFO. Logger stdout je nastaven na logování zpráv do konsole System.out, což je skvělé pro práci na lokálním stroji ve vývojovém prostředí, ale při použití v produkci je nutné si uvědomit, že System.out bude logovat do catalina.out

a tento soubor by mohl narůstat do obřích rozměrů, proto v budoucnu stdout z konfigurace na serveru odstraníme. Pro produkci jsme si připravili logování do souborů neo.log a neo-error.log, kde neo.log bude obsahovat všechny informace, které by jinak obsahovala konsole a neo-error.log, který bude obsahovat pouze zprávy na úrovní ERROR. To skvěle poslouží při debuggování chyb vzniklých na produkci. Dále jsou nastaveny vzory datumu a konverzí. To zaručí jednotvárnost zobrazení datumu v souborech a tím pohodlnější vyhledávání.

5.3 Konfigurace Spring Frameworku

Díky Mavenu jsme do projektu přidali knihovny z frameworku Spring a nyní můžeme kód z této knihovny využívat ke konfiguraci webové aplikace.

Jak je zmíněno dříve v práci, aplikace bude spouštěna na Apache Tomcat ve verzi 7 a vyšší, a to z toho důvodu, že v těchto verzích je použit Servlet verze 3.0 (3.1 v případě Tomcat 8) a ten obsahuje ServletContainerInitializer, který podporuje implementace servlet containeru za použití klasického Java kódu, konkrétně třídu WebApplicationInitializer, která tak plně nahrazuje klasický soubor pro upřesnění nasazení webové aplikace - web.xml. Tímto se naprosto zbavíme jakékoliv konfigurace přes XML a vše bude jednotvárně v Java konfiguraci.

5.3.1 Konfigurace Webového kontextu

Dříve, než přejdeme k samotnému vytváření konfiguračních tříd, je potřeba si rozvrhnout, jak konfigurace naší aplikace bude zhruba vypadat. Víme, že část konfigurace bude nahrazovat web.xml a definovat dispečer, bude se tedy týkat webové části aplikace. Druhá část konfigurace se bude zabývat zbytkem, jenž představuje kontext aplikace. Bylo by tedy dobré toto rozdělení držet i v naších třídách. Vytvoříme si tedy ve složce src balíčky config.web a config.app.

Jako první nakonfigurujeme webovou část. Vytvoříme si tedy třídu, která bude nahrazovat web.xml a to tak, že bude implementovat rozhraní WebApplicationInitializer, z balíčku org.springframework.web, a pojmenujeme ji WebInitializer. Tato třída poté musí implementovat metodu onStartup s parametrem ServletContext, která je deklarována v rozhraní. Jak jsme si uvedli v teoretické části v kapitole 1.2.5 Spring MVC, ve web.xml je třeba zaregistrovat dispečer a definovat jeho mapování. Také je to místo, kde se nastavuje aplikační kontext. Pro nás bude veškerá tato konfigurace právě v metodě onStartup. Poté tělo metody onStartup vypadá následovně.

```
// Vytvoření aplikačního kontextu pro Spring Dispatcher
AnnotationConfigWebApplicationContext dispatcherServletCtx = new
AnnotationConfigWebApplicationContext();
//registrace konfigurační třídy Dispatcheru
dispatcherServletCtx.register(Dispatcher.class);

//registrace dispatcheru do servletContextu
ServletRegistration.Dynamic dispatcher =
servletContext.addServlet("dispatcher", new
DispatcherServlet(dispatcherServletCtx));

//nastavení pořadí načtení a mapování
dispatcher.setLoadOnStartup(1);
dispatcher.addMapping("/");
```

Nyní máme nakonfigurovaný DispatcherServlet pro servletContext, dále pro něj vytvoříme konfiguraci, kterou jsme zde zaregistrovali jako Dispatcher.class. V této třídě je potřeba nakonfigurovat, kde bude dispečer skenovat pro kontroléry. Dále zde také budeme konfigurovat cesty ke zdrojům a engine s view resolverem pro Thymeleaf. Spring pro konfiguraci Dispatcheru pomocí Java konfigurace nabízí abstraktní třídu WebMvcConfigurerAdapter, která dopředu implementuje potřebné metody a potomkům nabízí přetížit a definovat pouze ty, o které je zájem. Vytvoříme tedy danou třídu.

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = {"cz.zoubelu.controller"})
public class Dispatcher extends WebMvcConfigurerAdapter {}
```

Jako první je vidět anotace @EnableWebMvc. Tato anotace, v případě, že je použita v kombinaci s @Configuration, naimportuje Spring MVC konfiguraci z WebMvcConfigurationSupport. Dále tu je @ComponentScan. Tato anotace indikuje, ve kterém balíčku hledat třídy, které mají jednu z @Component, @Service, @Controller, @Repository anotací. K těmto anotacím stačí vědět, že třída takto anotovaná, je po specifikování v @ComponentScan automaticky registrovaná jako bean definice v rámci aplikačního kontextu. V tomto případě je sken nastaven na balíček, ve kterém se budou nacházet všechny kontroléry, tedy třídy s anotací @Controller. Dále nastavíme vše potřebné pro Thymeleaf šablony. Nebudeme si zde ukazovat konfiguraci všech metod, jež jsou k tomuto potřeba, jediné, co si zde předvedeme je konfigurace view resolveru, abychom tak navázali na teoretickou část, ve které byl slíben konkrétní příklad. Šablony budou typu HTML a budou se nacházet v adresáři WEB-INF/templates/, konfigurace tedy vypadá následovně.

```
@Bean
public ServletContextTemplateResolver thymeleafResolver() {
        ServletContextTemplateResolver resolver = new
ServletContextTemplateResolver();

      resolver.setPrefix("/WEB-INF/templates/");
      resolver.setSuffix(".html");
```

```
        resolver.setTemplateMode("HTML5");
        resolver.setCharacterEncoding("UTF-8");
        resolver.setCacheable(false);
        return resolver;
}
```

Nakonec přidáme nastavení pro zdroje. Provedeme tak přetížením metody addResourceHandlers. V těle metody pak nastavíme handler a lokaci pomocí kódu.

```
registry.addResourceHandler("/js/**").addResourceLocations("/WEB-
INF/js/");
```

Obdobně přidáme pro styly a obrázky. Poslední, na co nesmíme zapomenout, je vytvořit balíček controllers. Zároveň vytvoříme kontrolér a pohled, na kterém vyzkoušíme, zda je vše nakonfigurováno správně.

```
@Controller
public class WebController {

        @RequestMapping(value = {"/"})
         public String index() {
             return "visualisation";
         }
}
```

V této fázi můžeme zkusit výsledek konfigurace. Pomocí příkazů mvn clean install a následně mvn tomcat7:run zkusíme spustit aplikaci.

Dle výpisu z konsole vidíme, jak probíhá inicializace jednotlivých komponent.

```
Spring WebApplicationInitializers detected on classpath: [cz.zoubelu.config.web.WebInitializer]
Initializing Spring FrameworkServlet ‚dispatcher‘
DispatcherServlet:488 – FrameworkServlet ‚dispatcher‘ initialization started
AnnotationConfigWebApplicationContext:578 – Refreshing WebApplicationContext
AnnotationConfigWebApplicationContext:207 – Registering annotated classes: [class:.. ]
SimpleUrlHandlerMapping:341 – Mapped URL path [/js/**] onto handler of type…
DispatcherServlet:507 – FrameworkServlet ‚dispatcher‘: initialization completed in 5679ms
Starting ProcotolHandler [„http-bio-8080“]
```

Z výpisu uvedeného výše je poznat, že WebInitializer byl detekován a následně se spustila inicializace dispečera. Dále vidíme, že mapování dispečera i zdrojů proběhlo úspěšně a inicializace se dokončila. Pokud do prohlížeče zadáme adresu localhost:8080, vidíme pohled visualisation.html. Celý proces dispečinku je možné logovat při přepnutí na debug úroveň v log4j.properties.

5.3.2 Konfigurace Aplikačního kontextu

Nyní, když už máme nakonfigurovaný MVC framework, se můžeme zabývat konfigurací samotné aplikace. Na kontextu bude asi nejdůležitější implementace profilů. Víme, že pro naší aplikaci potřebujeme jiný kontext pro testy, než je potřeba na produkci. V testovacím kontextu je nežádoucí pracovat s produkčními databázemi a budeme potřebovat jistou kompenzaci v podobě databází, které jsou jen v paměti a vznikají a zanikají se spuštěným testem, nebo alespoň ukládají data do souboru na disku.

Konfigurace připojení k relační databázi

Za relační databázi k tomuto účelu slouží H2 databáze. Tato databáze je univerzální a lze jí nastavit většinu parametrů rovnou v JDBC URL.

Jelikož bude vytvořena pouze při spuštění testu, musíme vždy znovu vytvořit celou strukturu databáze. K tomu použijeme třídy z frameworku Spring. DataSourceInitializer, kterému nastavíme zdroj dat, tedy připojení k databázi, a ResourceDatabasePopulator, kterému nastavíme skripty, jenž má vždy při startu testovacího kontextu spustit. V těchto skriptech bude vytvoření tabulky Message, a vložení testovacích dat. Připojení k databázi bude konfigurováno na ComboPooledDataSource, který bude společný pro H2 i Oracle databázi.

Vytvoříme konfigurační třídu ApplicationConfig a pomocí anotace @Autowired si vyžádáme „natažení“ vytvořené instance ComboPooledDataSource ze Spring IoC kontejneru, do anotované proměnné. Dále vytvoříme dvě konfigurační třídy registrující beanu ComboPooledDataSource, kde každá z těchto tříd bude mít jiný profil. Třídy pojmenujeme DataSourceProd a DataSourceTest. To, která instance bude nastavena do proměnné v ApplicationConfig třídě, bude tedy záviset na vybraném profilu. Konkrétní implementace ApplicationConfig.

```
@Configuration
public class ApplicationConfig {

    @Autowired
    private ComboPooledDataSource dataSource;
}
```

Implementace testovacího i produkčního dataSource bude obsahovat stejnou definici bean pro ComboPooledDataSource.

```
@Bean
public ComboPooledDataSource getDataSource() {
    ComboPooledDataSource cpds = new ComboPooledDataSource();
    try {
        cpds.setDriverClass(driver);
        cpds.setJdbcUrl(url);
        cpds.setUser(username);
        cpds.setPassword(password);
        cpds.setMinPoolSize(1);
        cpds.setMaxIdleTime(280);
        cpds.setMaxPoolSize(3);
    } catch (Exception e) {
        log.error("Failed to initialize datasource.", e);
    }
    return cpds;
}
```

Nemuseli bychom tuto bean vytvářet dvakrát, pokud bychom jednotlivé parametry vyextrahovali do konfiguračních souborů. V tomto případě, je však konfigurace zdroje v Javě pro produkci, z hlediska bezpečnosti žádoucí. Jednotlivé třídy pak budou mít anotace @Profile(„test“) a @Profile(„production“).

Jako další je třeba nakonfigurovat driver, JDBC URL a autentifikační údaje k H2 databázi. JDBC URL nastavíme jako databázi v paměti s dialektem Oracle.

```
url=jdbc:h2:mem:test_mem;DB_CLOSE_DELAY=-
1;MODE=Oracle;AUTO_SERVER=FALSE
```

Driver je použit org.h2.Driver a autentifikační údaje si můžeme zvolit. Takto nakonfigurovaný testovací zdroj dat již můžeme začít používat. Třída, resp. rozhraní z návrhu, které bude se zdrojem pracovat, je MessageRepository. Tento repositář musí získat data z tabulky a umět výsledek omezit časovým intervalem. Tabulka bude mít každý měsíc jiný název vzhledem k sufixu označující rok a měsíc. Z těchto důvodů si zvolíme použití JdbcTemplate, opět z knihovny Spring. Ta umožňuje práci s databází na nižší úrovni, to znamená, že SQL dotazy si budeme psát sami a mapování výsledků na kolekci objektů Message, na nás připadají také. Zdroj dat, se kterým budeme pracovat, je JdbcTemplate možné nastavit pomocí konstruktoru nebo setteru. Abychom mohli používat JdbcTemplate, musíme vytvořit třídu, která jí bude mít jako závislost (dependency), bude přes ní provádět dotazy a abstrahovat tak potřebu pracovat s template napřímo. Tato třída bude implementovat metody rozhraní MessageRepository a proto si jí nazveme MessageRepositoryImpl. Ve svém konstruktoru bude vyžadovat právě JdbcTemplate a přes něj jí bude injektována. Pokud vše vytvoříme a doplníme definice bean do kontextu, vypadá obsah třídy ApplicationConfig takto.

```
@Autowired
private ComboPooledDataSource dataSource;

@Bean
public JdbcTemplate getJdbcTemplate() {
    return new JdbcTemplate(dataSource);
}

@Bean
public MessageRepository getMessageRepo() {
    return new MessageRepositoryImpl(getJdbcTemplate());
}
```

Konfigurace zdrojů a repositáře pro relační databázi je hotová, zbývá připojení ke grafové databázi.

Způsob, jakým budeme pracovat s grafovou databází z kódu se od toho s relační značně liší. Namísto práce na nižší úrovni, kde si dotazy a mapování výsledků na objekt určujeme sami, zde přenecháme tuto práci na OGM frameworku. Pro práci s Neo4j databázemi má Spring svou knihovnu Spring Data Neo4j (dále jen SDN), která umožnuje velké množství funkcí a konfigurací, které velmi usnadňují práci s grafovou databází. Samotný popis této knihovny by byl velmi obsáhlý a bohužel není možné tu pokrýt veškeré informace o ní. Pokusíme se však co nejstručněji vysvětlit některé hlavní části, použité při konfiguraci.

Konfigurace připojení ke grafové databázi

Samotné modelování grafových entit a použití repositářů pomocí knihovny je pro tuto práci velmi důležité a věnujeme tomu tedy další kapitolu. V tuto chvíli tedy jen ke konfiguraci testovacího a produkčního připojení ke grafové databází Neo4j.

SDN umožňuje 2 typy konfigurace zdroje, na základě toho, kde grafová databáze poběží. Neo4j může běžet jako vestavěná instance na JVM, nebo klasicky na vzdáleném serveru, ke kterému se připojí pomocí HTTP. V této práci bude využito obou způsobů, a to vestavěné instance pro testovací účely a vzdáleného serveru pro produkční prostředí. Vytvoříme dvě konfigurační třídy GraphConfig a GraphTestConfig. SDN umožňuje Java konfiguraci přes třídu Neo4jConfiguration. Jejím rozšířením přetížíme inicializační metody bean pro Session a SessionFactory, a definujeme jim vlastní konfiguraci specifickou pro náš kontext. Session konfiguraci nebudeme potřebovat měnit a vrátíme pouze instanci z předka. Pro SessionFactory je však nutné specifikovat, v jakém balíčku se nachází grafové entity. Konfigurace testovacího kontextu vypadá následovně.

```
@Configuration
@Profile("test")
public class GraphTestConfig extends Neo4jConfiguration {

    @Bean
    public SessionFactory getSessionFactory() {
        return new SessionFactory("cz.zoubelu.domain");
    }

    @Bean
    public Session getSession() throws Exception {
        return super.getSession();
    }

}
```

K dokončení správné konfigurace je potřeba ještě přidat anotace @EnableNeo4jRepositories a @EnableTransactionManagement. První anotaci poskytneme atribut v podobě balíčku, ve kterém se nacházejí Neo4j repositáře, tedy „cz.zoubelu.repository“. Tyto anotace jsou potřeba vždy, nehledě na profil, a proto je přidáme do ApplicationConfig. Poslední věc pro testovací připojení je specifikovat driver pro neo4j-ogm. K tomu stačí vytvořit v src/test/resources soubor ogm.properties, který knihovna ogm automaticky vyhledává na classpath (lze také specifikovat přímo v konfiguraci v Java kódu, jak bude ukázáno dále v práci). V tomto souboru nakonfigurujeme driver.

```
driver=org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver
```

Pro úplné fungování je třeba přidat do pom.xml závislosti na neo4j knihovnách, ale to již nebudeme dále zmiňovat, jelikož je to samozřejmost.

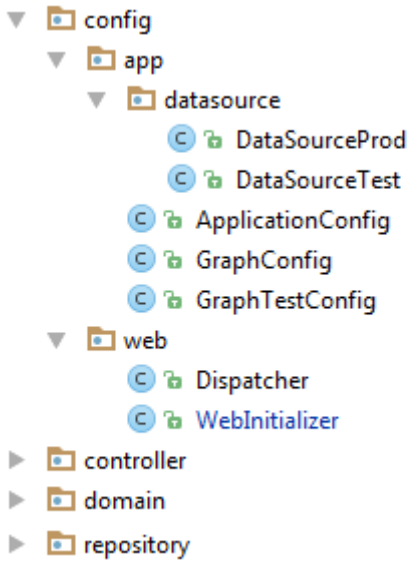
Produkční konfigurace bude mírně odlišná, a to z toho důvodu, že driver bude HTTP a nakonfigurujeme jej přímo v Java kódu, v konfiguraci. Navíc přidáme také informace o vzdáleném serveru. Tato konfigurace bude přidána jako parametr do SessionFactory. Obsah konfigurační třídy GraphConfig s anotací @Profile(„production“):

```
@Bean
public SessionFactory getSessionFactory() {
    return new SessionFactory(getConfiguration(),"cz.zoubelu.domain");
}

@Bean
public Session getSession() throws Exception {
    return super.getSession();
}

public Configuration getConfiguration() {
    Configuration config = new Configuration();
config.driverConfiguration().setDriverClassName("org.neo4j.ogm.drivers
.http.driver.HttpDriver").setURI("http://127.0.0.50:7474").
            setCredentials("jmeno", "heslo");
    return config;
}
```

Konfiguraci aplikačního kontextu pro databázové zdroje máme hotovou a aby naše aplikace věděla o naší konfiguraci, je zapotřebí přidat kontext do naší webové aplikace. Učiníme tak v metodě onStartup ve WebInitilizeru, a jelikož se spouštění aplikace přes WebIniilizer bude brát jen jako produkční, vybereme jen ty konfigurační třídy, kterých se produkční kontext týká a nastavíme spouštěcí profil na „production“.

```
AnnotationConfigWebApplicationContext appContext = new
AnnotationConfigWebApplicationContext();

appContext.register(ApplicationConfig.class,GraphConfig.class,
DataSourceProd.class);

servletContext.addListener(new ContextLoaderListener(appContext));

servletContext.setInitParameter("spring.profiles.active",
"production");
```

Aktuální strukturu konfiguračních tříd a vytvořených balíčku ilustruje obrázek 7.



Obrázek 7 Aktuální struktura aplikace [zdroj: vlastní]

5.4 Implementace aplikace

Aplikační kontext je naimplementován do takové míry, že se můžeme přesunout ke psaní samotné aplikace. V průběhu implementace bude samozřejmě zapotřebí do kontextu ještě pár věcí přidat a pravděpodobně i vytvořit nové konfigurační třídy, v nich se však nebude objevovat nic jiného, než s čím jsme se již seznámili, nebo to nebude tak zásadně důležité.

Začneme tím, že si vytvoříme všechny zbývající potřebné třídy a balíčky z návrhu a hned poté si vytvoříme ve složce src/test/java/ testovací třídu. Ve skutečnosti vytvoříme dvě testovací třídy. Jedna z nich bude abstraktní a druhá bude jejím potomkem. Tento vzor nám dovolí vše společné naimplementovat v abstraktní třídě a v samotných testovacích toto chování sdílet.

5.4.1 Testování

Od začátku konfigurace aplikace stále zmiňujeme dvě prostředí a dva profily, a jeden z nich je vždy testovací. V této kapitole si vysvětlíme, k čemu jsme takto prostředí připravovali.

Existuje několik způsobů, jak otestovat naši aplikaci. Rozhodně je však nutné, aby aplikace otestována byla, jelikož i ti nejlepší vývojáři dělají chyby. Někomu se může zdát testování zbytečné, časově náročné, a tak implementují svou aplikaci bez pokrytí testy. I přesto, že testování může být z počátku časově náročné, a ne vždy je možné si dovolit úplný Test Driven Development, určitě nejsou zbytečné. Testy slouží pro programátory jako pojistka, kontrola vlastního kódu, simulace chování programu s testovacími daty, ověření funkcionality po zásahu do kódu a další. To v

budoucnu ušetří spoustu času, debuggování a nepříjemných okamžiků, kdy při předvádění aplikace klientovi něco nebude fungovat. Samozřejmě, při implementaci malých programů, s pár řádky kódu není sada testů tak nutná jako u velkého systému s několika scénáři, ale i tak by jeden malý test neuškodil.

Při vývoji této aplikace byl kladen důraz na testování, ovšem, spíše jako již výše zmíněná pojistka a simulace chování kódu pro různé vstupy a ověření správného chování při integraci komponent. Tím se myslí, že pro každou hlavní funkcionalitu nejprve vytvoříme test s jednoduchou kontrolou a teprve poté dokončíme implementaci kódu a test použijeme pro debuggování tohoto kusu kódu (nejedná se však o ryzí Test Driven Development).

Junit

Jeden z nejvyužívanějších frameworků pro tvorbu testů v Java a Android aplikacích. Existuje pár nejlepších praktik a doporučení, na které je dobré při implementaci testů myslet. Pár z nich jsou například.

- Junit testy by měly být opakovatelné, jednoduché.
- Junit testy by měly být rychlé, nezávislé jeden na druhém.
- Nespoléhat se na pořadí testů ve kterém poběží.
- Měly by provolávat kód samotné aplikace a kontrolovat jeho správné provedení.
- Pojmenovávat testy srozumitelně podle účelu testu

Je také dobré pro přehlednost přidávat komentář k assert metodám pro jasné chybové hlášky.

Pro testy budeme potřebovat testovací data, která budou simulovat data reálná. Pro správné fungování testů, je nutné vytvořit jednoduché databázové schéma a v něm potřebné tabulky, co možná nejpřesněji k reálné tabulce, jenž bude poté využívána na produkčním prostředí. Dále je potřeba tuto tabulku naplnit daty. K tomu jsme si připravili SQL skripty pro vytvoření tabulky Message a testovacích dat. Tyto skripty přidáme do „populátoru“ v DataSourceTest, který jsme si připravili při konfiguraci testovacího kontextu.

Testování relační databáze

Abychom mohli testy spouštět s naší konfigurací z aplikačního kontextu pod určitým profilem, musíme nakonfigurovat TestRunner. Pro JUnit 4 se jedná o třídu

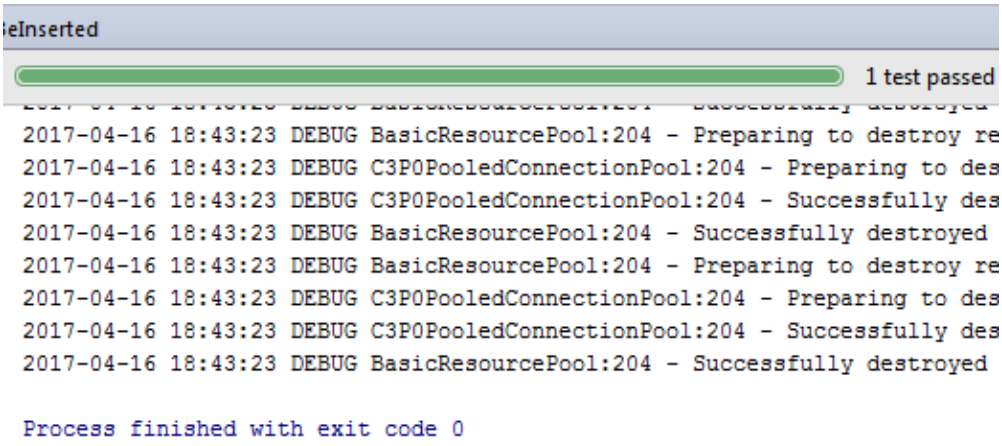
SpringJUnit4ClassRunner. Nastavení bude stejné pro všechny testovací třídy, a proto jej naimplementujeme v AbstractTest.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ApplicationConfig.class,
DataSourceTest.class, GraphTestConfig.class})
@ActiveProfiles("test")
public abstract class AbstractTest {}
```

Nyní se budeme věnovat třídě obsahující testy. Testem se bere veřejná metoda s anotací @Test. JUnit pak nabízí metody určené pro kontrolu výsledků jako například assertEquals(), assertTrue, assertNotNull a další. První test napíšeme tak, aby nám ověřil nejen fungující komunikaci s databází, ale také správné spuštění skriptů s testovacími daty. Následující metoda testuje, zda jsou v databázi všechny testovací záznamy. Testuje tak, dle celkového počtu záznamů.

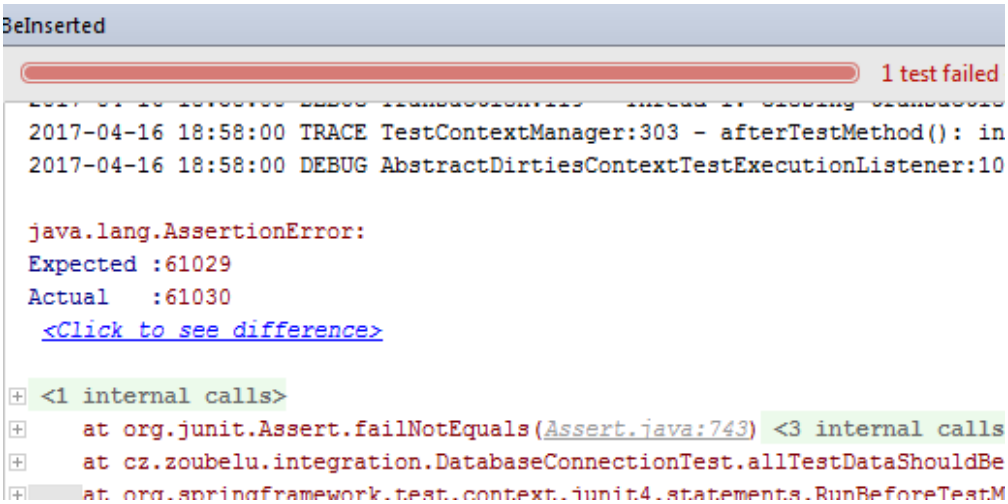
```
@Autowired
private JdbcTemplate jdbcTemplate;

@Test
public void allTestDataShouldBeInserted(){
    String query ="select count(*) from MESSAGE";
    Long size = jdbcTemplate.queryForObject(query, Long.class);
    Assert.assertEquals(61030, size.longValue());
}
```

Výsledek úspěšného testu ve vývojovém prostředí IntelliJ Idea ilustruje obrázek 8.



Obrázek 8 Vyhodnocení úspěšného testu [zdroj: vlastní]

Pokud bychom pro ukázku pozměnili očekávané číslo 61030 například na 61029, výpis assertEquals by vypsal chybu, jak je vidět na obrázku 9.



Obrázek 9 Výsledek neúspěšného testu [zdroj: vlastní]

Díky testu víme, že konfigurace testovací instance je správná a můžeme začít implementovat a testovat práci s daty ve třídě DataConversionHandler. Nejprve si však naimplementujeme grafové entity a jejich repositáře.

5.4.2 Implementace grafového modelu

V teoretické části jsme si navrhli, jaké entity se budou v aplikaci nacházet a jak takový model bude vypadat. Nyní si tento model naimplementujeme pomocí Spring Data Neo4j. Tato knihovna již byla použita při konfiguraci připojení grafové databáze, a tam jsme zmínili, že bližší konfiguraci, co se týče entit a repositářů, uvedeme v další kapitole, a tou je tato.

V konfiguraci jsme v SessionFactory nastavili balíček, ve kterém má Spring hledat grafové entity. Jedná se o balíček cz.zoubelu.domain, a ten v tuto chvíli již obsahuje prázdné třídy Application, Method a ConsumeRelationship. Abychom aplikaci řekli, že se jedná o grafovou entitu, použijeme jednu z anotací Objektově-grafového mapování od neo4j, kterými jsou @NodeEntity (označující vrchol) a @RelationshipEntity (označující vztah). Pro vlastnosti vrcholu či vztahu použijeme anotace @Property a @Relationship, kde druhá anotace definuje vztah k entitě. Konkrétní ukázka aplikace podle návrhu grafového modelu.

```
@NodeEntity
public class Application {

    @GraphId
    private Long id;

    @Property(name = "systemId")
    private Integer systemId;

    @Property(name = "name")
    private String name;

    @Relationship(type = "PROVIDES", direction=Relationship.OUTGOING)
```

```
    private List<Method> providedMethods;

    @Relationship(type = "CONSUMES", direction=Relationship.OUTGOING)
    private List<ConsumeRelationship> consumeRelationships;
}
```

@GraphId je povinný údaj, který musí být typu Long. Vyžaduje si tak SDN pro znovu-spojení entity s grafem. V teorii jsme v návrhu grafového model zmínili, že vztah „konzumuje“, je složitější, a to se projeví při implementaci. Je tomu tak proto, že vztah „PROVIDES“ je jednoduchý vztah, nemá žádné atributy a není potřeba jej zvláště vytvářet jako entitu. Toto však neplatí pro ConsumeRelationship. Tento vztah musí být vytvořen jako třída a musí mít definované @StartNode a @EndNode, což je počáteční a koncový uzel vztahu.

```
@RelationshipEntity(type = "CONSUMES")
public class ConsumeRelationship {
    @GraphId
    private Long id;
    @Property
    private Long totalUsage;
    @StartNode
    private Application application;
    @EndNode
    private Method method;
}
```

Každá z uvedených entit má navíc gettery a settery pro všechny své atributy a také konstruktory. Co se týče repositářů pro práci s těmito entitami, jejich konfiguraci již máme také připravenou. Kontext skenuje repositáře v balíčku cz.zoubelu.repository. Abychom nastavili naše repositáře jako grafové, nabízí SDN rozhraní GraphRepository<T>, kde za parametr T se doplní entita, pro kterou bude repositář sloužit. Náš repositář by měl být tedy rozhraní, které je potomkem tohoto rozhraní. Tato GraphRepository již sama poskytuje CRUD operace pro entitu, kterou jí nastavíme a není tedy potřeba, aby náš repositář měl definice těchto metod.

Samozřejmě budou v reálném použití zapotřebí i jiné databázové operace a k tomu slouží anotace @Query, které v parametru vložíme Cypher dotaz. Tato anotace se používá na úrovni metody a pokud je potřeba dotaz parametrizovat atributy této metody, lze tak učinit pomocí placeholderů v dotazu. Víme, že v budoucnu budeme potřebovat získat aplikaci z grafu na základě systémového ID. Abychom si předvedli, jak takový dotaz vypadá, vytvoříme si v repositáři tuto metodu s parametrem Integer.

```
public interface ApplicationRepository extends
GraphRepository<Application> {

    @Query("MATCH(a:Application) WHERE a.systemId={0} RETURN a")
    Application findBySystemId(Integer systemId);
}
```

Repositář za nás, bez jakékoliv další konfigurace, dotaz provede a výsledek namapuje na entitu Application. Obdobně tedy vytvoříme ostatní repositáře. Pro kontrolu funkčnosti si vytvoříme jednoduchý test, který ověří, zda se aplikace do grafové databáze uloží, a zároveň, zda metoda save repositáře provede aktualizaci hodnot v databázi, aniž by vytvořila novou entitu.

```
@Test
public void shouldUpdateApplicationInformation() {
    applicationRepo.save(new Application("test", null));
    Application app = applicationRepo.findByName("test");

    Long initId = app.getId();
    app.setName("modified");
    applicationRepo.save(app);

    Application modifiedApp =
            applicationRepo.findByName("modified");
    Assert.assertEquals(initId, modifiedApp.getId());
}
```

Po proběhnutí testu s jistotou víme, že konfigurace repositářů a grafových entit, stejně tak jako spuštění testovací instance grafové databáze, funguje.

5.4.3 Implementace konverze a dynamického vytváření entit

V této fázi opustíme od konfigurace a přesuneme se k psaní kódu, který bude tvořit hlavní funkcionalitu aplikace.

Nebudeme zde opakovat text z návrhu aplikace, ale pouze si připomeneme, z čeho vycházíme. Je zapotřebí vymyslet, jak provést konverzi zprávy, tedy Message, na vztah mezi entity Application a Method. Přitom je důležité, aby konverze probíhala po řádcích, ale tak, aby neukládala celý výsledek dotazu z databáze do RAM.

Pro získání dat z relační databáze používáme JdbcTemplate a ta nabízí více možností, jak pracovat s výsledkem po řádku. Dva nepoužívanější jsou RowMapper a RowCallbackHandler. Oba mají v podstatě stejnou funkčnost, a to tu, že prochází ResultSet, řádek po řádku. RowMapper je dobré použít, pokud bychom chtěli získat z ResultSetu list objektů, kde řádek je záznam a my definujeme mapování. I přesto, že v řádku budeme mapovat na objekt Message, jde zde o více než jen o mapování a hlavní činností pro každý řádek bude konverze na grafové entity. Z toho důvodu použijeme RowCallbackHandler. Jedná se o rozhraní, které deklaruje metodu processRow a DataConversionHandler bude toto rozhraní i metodu implementovat.

Kolik řádků bude najednou do ResultSetu načteno, si nastavíme pomocí parametru fetchSize u JdbcTemplate. Tím můžeme ovlivnit výkon a rychlost převodu. Náš převaděč bude tedy nejen implementovat metodu convertData

z rozhraní DataConversion, ve které na základě atributů zavolá metodu z MessageRepository pro získání dat, ale také se postará o zpracování tohoto výsledku. Dále bude pracovat s třídou SystemsList, která bude obsahovat seznam všech systémů z číselníku, který je potřeba pro udržení struktury, jak jsme vysvětlili v návrhu. Samozřejmě bude používat také rozhraní DynamicEntityProvider.

Začneme implementací metody convertData.

```
public List<ConversionError> convertData(String tableName,
                                         TimeRange timeRange) {
    if (systemsList.listFileExists()) {
        systemsList.loadList();
    }

    messageRepository.fetchAndConvertData(tableName, timeRange, this);
    return persistCacheAndFinish();
}
```

Pokud se nepodaří načíst ze souboru, seznam má jako pojistku celý číselník naprogramovaný v kódu v konstruktoru. Pokud se zrovna stane, že v této iteraci dojde k objevení nového systému neznámého pro číselník, je dále v kódu ošetřeno výpisem do logu o vytvoření nového systému a administrátor pak může zkontrolovat a provést případnou opravu dat. Dále se Message repositáři metodou fetchAndConvertData předají parametry, kde poslední specifikuje handler, který má ResultSet zpracovat. V tomto případě předá svou instanci. Metoda fetchAndConvertData spouští následující dotaz.

```
private static final String QUERY_BY_TIMERANGE = "select
ID,REQUEST_TIME,RESPONSE_TIME,APPLICATION,MSG_TYPE,MSG_VERSION,MSG_SRC
_SYS,MSG_TAR_SYS from $table where request_time between ? and ?";
```

Tělo metody pak vypadá takto.

```
if (timeRange != null) {
    String query = QUERY_BY_TIMERANGE.replace("$table", tableName);
    this.jdbcTemplate.query(query, handler, timeRange.getStartDate(),
            timeRange.getEndDate());
} else {
    fetchAndConvertData(tableName, handler);
}
```

Způsob dynamického vkládání názvu tabulky do dotazu, jak je vidět na příkladu výše, je náchylný na SQL injection útoky. Ponecháme však takto, jelikož aplikace je na interní síti a krom toho, název tabulky bude určovat pouze plánovač, který bude mít defaultně nastaven název tabulky přímo v kódu jako „MESSAGE“ a pouze v případě, kdy bude vybírat z archivních tabulek přidá prefix a sufix, pro konkrétní měsíc.

V příkladu výše je také vidět, že pokud je timeRange null, zavolá se metoda fetchAndConvertData bez tohoto parametru. Tato metoda má stejné chování pouze postrádá klauzuli WHERE vymezující časový úsek.

Nyní k implementaci samotného zpracování dat v metodě processRow ve třídě DataConversionHandler.

```
public void processRow(ResultSet resultSet) throws SQLException {
    Message message = null;
    try {
        message = mapper.mapRow(resultSet);
        convertSingleMessage(message);
    } catch (Exception e) {
        log.error("Error occurred during conversion.", e);
        errors.add(new ConversionError(
                "Failed to convert message with ID: " + message !=
null ? message.getMsg_id() : "N/A" + ". Reason: " + e.getMessage()));
    }
}
```

Zde začneme mapováním řádku na objekt Message. K tomu jsme si vytvořili třídu MessageMapper, jejíž referenční proměnná se v kódu jmenuje mapper. Dále se volá metoda convertSingleMessage s již vytvořeným objektem message. Pro chybové stavy si vytvoříme v DataConversionHandleru kolekci s ConversionError, konkrétně Set. Tyto chyby tak budeme moci posbírat, aniž bychom přerušili konverzi a následně vypíšeme všechny naráz. Vypisujeme však i ve chvíli, kdy se chyba stane, aby bylo možné dohledat čas a okolnosti chyby v logovacím souboru. Ukažme si metodu convertSingleMessage, ve které se děje celá konverze zprávy na vztah.

```
Application providingApp = getProvidingApplication(msg);

Method consumedMethod = provider.getConsumedMethod(providingApp,
msg.getMsg_type(),msg.getMsg_version());

Application consumingApp = getConsumingApplication(msg);

log.info(String.format("Saving relationship - Provider: %s, Method:
%s, Consumer: %s.", providingApp.getName(),
        consumedMethod.getName(), consumingApp.getName()));

provider.createConsumeRelation(consumingApp, consumedMethod);
```

Získání aplikací

Metody getProvidingApplication a getConsumingApplication se starají o zjištění, zda je aplikace v číselníku systémů. K tomu slouží metody getSystemByName a getSystemById, které jsou implementované ve třídě SystemsList. Pokud se metodě getSystemById nepodaří nalézt systém, vytvoří jej, a pro název použije hodnotu ID. Pokud se to však nepodaří metodě getSystemByName, je to horší, jelikož systémové ID je kritické pro unikátní identifikaci aplikace v REST rozhraní. Vytvoří tedy systém s vygenerovaným ID a přidá jej zároveň do pomocného listu nově vytvořených systémů. V metodě getProvidingApplication se poté kontroluje, zda systém se kterým aktuálně pracuje není náhodou v tomto seznamu. Pokud tomu tak je, pokusí se získat ID systému ještě z parametru msg_targ_sys. V případě, že selže i to, vypíše

zprávu o tom, že byl vytvořen nový systém s vygenerovaným ID, aby o tom administrátor věděl. Ukázka těla metody getProvidingApplication.

```
SystemApp system = systemsList.getSystemByName(msg.getApplication());

if (systemsList.getNewlyCreated().contains(system)) {
    if (msg.getMsg_tar_sys()!=null &&!msg.getMsg_tar_sys().equals(0)){
        system.setId(msg.getMsg_tar_sys());
    } else {
        log.info(String.format("Added new system w. name: %s, and
generated system ID: %s", system.getName(), system.getId()));
    }
}
return provider.getApplication(system);
```

Ať se systém najde, nebo se vytvoří, dále se posílá na DynamicEntityProvider, aby na základě něj vrátil grafovou entitu, která se v metodě convertSingleMessage napojí na vztah. V metodě getApplication se pak nahlédne do cache, zda je přítomna a pokud není, snaží se jí získat z databáze či nově vytvořit a přidat do cache.

```
public Application getApplication(SystemApp system) {
    if (cache.contains(system)) {
        return cache.get(system);
    } else {
        Application app =
applicationRepo.findBySystemId(system.getId());
        if (app != null) {
            cache.cacheApplication(app);
            return app;
        } else {
            return createApplication(system);
        }
    }
}
```

Tímto se získává aplikace, jak poskytující, tak konzumující. Získání konzumující je o to snazší, že není zapotřebí kontrolovat, zda je systém nově vytvořen a rovnou jej předáme provideru.

Získání metody

Pro celkový vztah je nutné ještě získat metodu, která v něm figuruje. Tu pro nás získá provider metodou getConsumedMethod, které parametrem předáme poskytující aplikaci, název a verzi metody.

```
public Method getConsumedMethod(Application providingApp, String name,
Integer version) {

    if (cache.contains(providingApp, name, version)) {
        return cache.get(providingApp, name, version);
    } else {
        Method method =
methodRepository.findProvidedMethod(providingApp, name, version);
        if (method != null) {
            cache.cacheMethod(providingApp, method);
            return method;
        } else {
            return createMethod(providingApp, name, version);
```

```
        }
    }
}
```

Zajímává je implementace metody findProvidedMethod, ve které jsme si napsali náš vlastní dotaz.

```
MATCH (m:Method)<-[:PROVIDES]-(a:Application) WHERE m.name={mName} AND
m.version={mVersion} AND id(a)={id} RETURN m
```

Jednotlivé parametry jsme pak metodě definovali pomocí anotace @Param.

```
@Param("id") Application providingApp, @Param("mName") String
methodName, @Param("mVersion") Integer methodVersion
```

Cache, kterou v předchozím příkladu používáme, si naimplementujeme sami. To provedeme tak, že si vytvoříme rozhraní nad třídou pracující s HashMapy, kde hash budeme vytvářet pomocí jedinečných identifikátorů entity. Metodu z cache získáváme na základě tří parametrů, a to včetně poskytující aplikace, která s jedinečnou identifikací metody na první pohled nemá nic společného. Existují však metody, které mají název i verzi stejnou a zároveň jsou poskytovány více aplikacemi. Konkrétním příkladem může být metoda ping, kterou poskytují všechny služby.

Co se týče chování, to je stejné, jako tomu bylo u aplikace. Pokud nenalezne metodu v cache, pokusí se jí získat z databáze, či vytvořit. Vytvoření metody pak vypadá takto.

```
private Method createMethod(Application providingApp, String name,
Integer version) {
    Method method = new Method(name, version);

    log.info(String.format("Creating method: %s, version: %s for app
%s.", method.getName(), method.getVersion(),
            providingApp.getName()));

    if (providingApp.getProvidedMethods() != null) {
        providingApp.getProvidedMethods().add(method);
    } else {
        providingApp.setProvidedMethods(Lists.newArrayList(method));
    }
    applicationRepo.save(providingApp);
    cache.cacheMethod(providingApp, method);

    log.info(String.format("Application: %s successfully saved.",
providingApp.getName()));
    return method;
}
```

Vytvoření metody je jiné, a jelikož se v případě vzniku nové metody musí aktualizovat i seznam poskytujících metod u aplikace, je vytvořena tak, že jí přidáme do kolekce poskytujících metod aplikace a následně tuto aplikaci uložíme.

Vytvoření vztahu

Na základě konzumenta a metody se pokusíme zjistit, zda vztah existuje v cache nebo v databázi. K dotázání se do databáze si opět vytvoříme metodu s anotací @Query, které nastavíme následující skript.

```
MATCH (m:Method)<-[c:CONSUMES]-(a:Application) WHERE id(m)={mId} AND
id(a)={appId} RETURN c
```

Pokud se podaří nalézt, uložíme vztah do cache a inkrementujeme hodnotu totalUsage, indikující počet celkového využití metody touto aplikací. V opačném případě vztah vytvoříme metodou createRelationship. Obsah těla metody createConsumeRelation.

```
ConsumeRelationship relationship = null;

if (cache.contains(consumer, method)) {
    relationship = cache.get(consumer, method);
} else {
    relationship = relationshipRepo.findRelationship(consumer,
method);
}

if (relationship != null) {
    cache.cacheRelation(relationship);
    log.debug(String.format("Relationship: %s CONSUMES -> %s, already
exists, incrementing total usage by 1.", consumer.getName(),
method.getName()));

    relationship.setTotalUsage(relationship.getTotalUsage() + 1);
} else {
    createRelationship(consumer, method);
}
```

Uložení do grafové databáze

Tímto jsme se ocitli na konci převodu. Po projetí všech záznamů se dostáváme na poslední řádek metody convertData, ve kterém voláme metodu persistCacheAndFinish.

```
return persistCacheAndFinish();
```

Tuto metodu si napíšeme tak, aby provedla vše potřebné k úspěšnému dokončení konverze. Zde je implementace.

```
private List<ConversionError> persistCacheAndFinish() {
    provider.persistCachedRelations();
    systemsList.saveList();
    return Lists.newArrayList(errors);
}
```

Metoda persistCachedRelations vypíše hlášení o ukládání vztahů do databáze a všechny vztahy co jsou v cache uloží.

```
public void persistCachedRelations() {
    log.info("Saving cached relations into the database.");
    relationshipRepo.save(cache.getRelations());
    log.info("Saved.");
}
```

5.4.4 Implementace REST API

V tuto chvíli máme k dispozici všechna data v grafové databázi a je načase pro ně vytvořit rozhraní, přes které je budeme poskytovat uživatelům, či sami sobě jako zdroj pro vizualizaci grafu. Již dříve jsme si ukázali, jakým způsobem vytvoříme kontrolér ve Spring MVC. Také jsme zmínili, že takový kontrolér je nucen vracet pro jednotlivá mapování jeden z objektů: String, View nebo ModelAndView.

Jakým způsobem tedy zařídíme, abychom mohli v odpovědi vracet JSON? K tomu slouží anotace @RestController. Tato anotace v podstatě spojuje dohromady anotace @Controller a @ResponseBody. Právě druhá anotace nám umožní vrácení JSON v odpovědi. Požadavkem ale je, vracet v response JSON jednotlivých grafových entit, tedy Application, Method a ConsumeRelationship. Musíme tedy použít nějaký generátor, který nám z instancí těchto tříd automaticky vygeneruje JSON. K tomu použijeme JSOGGenerator z knihovny com.voodoodyne.jackson.jsog. K jeho konfiguraci použijeme anotaci @JsonIdentityInfo z com.fasterxml.jackson.annotation. Tuto anotaci definujeme u všech tří entit. Ukázka pro aplikaci.

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@NodeEntity
public class Application {}
```

Tato knihovna nabízí také anotace jako @JsonIgnoreProperties, které lze specifikovat, jaké atributy entity v JSON nechceme a tím jej mít čistější a přehlednější.

Entity máme připraveny, vytvoříme si tedy rest kontrolér, který pojmenujeme RestGraphController, jelikož jeho hlavní funkcí je poskytovat data z grafu. Mapování kontroléru je třeba si předem trochu promyslet. Je velmi důležité, jakou vytvoříme architekturu URL pro naše rozhraní. Víme, že rozhraní bude poskytovat minimálně údaje o poskytovaných metodách aplikace, údaje o konzumovaných metodách aplikace, seznam systémů z číselníku a data pro vizualizaci. Identifikátorem aplikace bude jeho systémové ID. Vzhledem k těmto informacím navrhneme následující mapování.

- Konzumované metody - /application/{id}/consumes.
- Poskytované metody - /application/{id}/provides
- Celkové informace o aplikaci - /application/{name}/info
- Seznam systémů z číselníku - /codelist
- Data celého grafu - /graph

Celý rest kontrolér by měl mít své vlastní mapování, aby se oddělilo od mapování klasických webových kontrolérů. Přidáme tedy URL kontext „/api“. Všechny URL vypsané výše budou mít tedy společný kořenový kontext.

```
@RequestMapping(value = "/api")
public class RestGraphController {}
```

Dále vytvoříme metodu pro vrácení metod, které jsou konzumovány aplikací se systémovým ID zadaném v URI.

```
@RequestMapping(value = "/application/{systemId}/consumes", method =
RequestMethod.GET)
public List<ConsumeRelationship>
getConsumedMethods(@PathVariable("systemId") Integer systemId) {
    Application app = applicationRepo.findBySystemId(systemId);
    return app.getConsumeRelationship();
}
```

Metoda se doptává na data a z toho důvodu využívá HTTP metodu GET. Teď, když už máme anotované všechny třídy a vytvořili jsme si metodu getConsumedMethods, jenž je na příkladu výše, můžeme aplikaci spustit a vyzkoušet naše API.

K tomu budou potřeba data a jelikož se v žádném případě nesmí provádět testování na produkčních datech a jinou databázi k dispozici nemáme, napojíme si pro sebe lokálně produkční profil na testovací instance. Toto provedeme tak, že produkčním konfiguracím zdrojů nastavíme stejnou konfiguraci jako testovacím, změníme scope u Maven závislostí pro tyto databáze a přidáme do /main/resources/ složku se skripty se strukturou a daty. Abychom si některá data mohli převést do grafu, vytvoříme si v kontroléru pomocnou metodu mapovanou na /insert, přijímající 4 parametry. Rok, měsíc, den a hodinu. V této metodě přijmeme parametry a provoláme metodu z DataConversionHandler, která provede konverzi nad daty za daný den od půlnoci až do specifikované hodiny. Nejedná se o nejlepší způsob, jak tento problém řešit, ale je nejrychlejší a můžeme si tak určit interval dat, který se převede. Navíc se nám metoda bude hodit dále při implementaci vizualizace.

Spustíme tedy aplikaci a do prohlížeče zadáme URL: http://localhost:8080/api/insert?year=2016&month=06&day=01&hour=01.

Jelikož známe testovací data, víme, že po převodu by měla být v grafu aplikace se systém ID 1001, která za první hodinu využila minimálně 2 metody, a to createGetDoc a calculateProduct. Dotážeme se tedy na end-point …/api/application/1001/consumes, abychom viděli, zda se tato data vrátí jako JSON.


```
[
  {
    "@id": "1",
```

```
    "id": 3,
    "totalUsage": 4,
    "method": {
      "@id": "2",
      "id": 4,
      "name": "createGetDoc",
      "version": 100
    }
  },
  {
    "@id": "3",
    "id": 8,
    "totalUsage": 2,
    "method": {
      "@id": "4",
      "id": 6,
      "name": "calculateProduct",
      "version": 110
    }
  }
]
```


Jde vidět, že JSON se vytvořil v pořádku a se správnými daty. Vytvoříme tedy ostatní metody, jejichž implementaci si zde již nebudeme ukazovat, jelikož se jedná o stejný princip.

5.4.5 Plánovač

Aby se splnily požadavky, musí převod dat probíhat automaticky za nějakou periodu. Vzhledem k funkčním požadavkům na aplikaci, se jedná o převod jednou za měsíc. Je však třeba myslet na to, že požadavky se mohou změnit a převod by mohl být denní, a to je pro nás komplikace. Měsíční převod totiž může probíhat pouze nad archivními tabulky, které jsou již uzavřené a mají unikátní sufix, kdežto převod denní by mohl běžet přímo nad tabulkou Message. K tomu si vytvoříme pomocné utility třídy, které na začátku konverze poznají, za jakou frekvenci převod probíhá a dle toho vrátí název tabulky.

Aplikace bude také hodně pracovat s časovými objekty jako Timestamp, Date a Calendar a veškerou práci s nimi budou poskytovat právě utility.

Pro implementaci této automatizace použijeme cron4j scheduler, který je schopen spouštět úlohu na základě vzoru plánování. Tento vzor je pro nás řetězec znaků, který popořadě definuje:

1. V kterou minutu hodiny se má spustit
2. V kterou hodinu dne

3. V který den v měsíci
4. Který měsíc v roce
5. Který den v týdnu

Existuje tedy několik možných kombinací, jak scheduler využít. Zápis je v podobě řetězce, kde znakem '*', je možné nahradit řád a říci tak, že se pro tento řád stane pokaždé. Například vzor pro úlohu, který se má spustit každou desátou minutu hodiny, by vypadal takto: 10 * * * *.

Ve výchozím nastavení budeme používat převod jednou za měsíc. Aby byl převod co nejrychlejší, budeme jej spouštět po půlnoci, kdy bude pravděpodobně nejmenší provoz na síti a například 5. den měsíce, abychom měli jistotu, že tabulka za předchozí měsíc je již zarchivovaná. Pro takový případ si sestavíme vzor „0 1 5 * *".

Každý 5. den v měsíci, se tedy v jednu hodinu ráno spustí úloha, utility třídy zjistí, o jakou se jedná frekvenci a vrátí název tabulky. V úloze se zavolá metoda convertData z dataConversionHandleru a předá se jí název tabulky. Vše pak proběhne tak, jak známe. Naimplementujeme tedy tuto úlohu a její scheduler. Jelikož budeme chtít spravovat inicializaci bean scheduleru, vytvoříme si pro přehlednost novou konfigurační třídu SchedulerConfig, do které vypíšeme konfiguraci týkající se plánování. Inicializace bean vypadá poté následovně.

```
@Value("${scheduler.pattern}")
private String schedulingPattern;

@Bean
public Scheduler scheduler() {
    Scheduler s = new Scheduler();
    SchedulingPattern pattern = new
SchedulingPattern(schedulingPattern);
    ConversionTask task = conversionTaskImpl();
    String scheduleTaskId = s.schedule(pattern, task);
    task.setScheduledTaskId(scheduleTaskId);
    task.setPattern(pattern);
    return s;
}

@Bean
public ConversionTask conversionTaskImpl() {
    return new ConversionTask();
}
```

Defaultní vzor časování definujeme v konfiguračním souboru, který si vytvoříme a nazveme jej neo-config.properties. Tímto budeme moci konfigurovat jiný vzor pro test a pro produkci, a zároveň jej také změnit na serveru, aniž bychom museli přepsat kód aplikace a znovu jí muset nasazovat. V budoucnu by však pro tento

případ bylo pěkné vytvořit jednoduchý kontrolér, pomocí kterého bude možné časovač nastavovat a řídit z webového prohlížeče.

Takto nastavený scheduler je připraven k použití a dokud nezavoláme metodu scheduler.start(), spuštění úlohy neproběhne. V ukázce kódu uvedeného výše vidíme, že úloha má název ConversionTask. To je naše implementace cron4j Task, definující chování metody execute. Tělo této metody naimplementujeme následovně.

```
log.info("Starting conversion.");
long startTime = System.currentTimeMillis();

Frequency frequency =
DateUtils.recogniseSchedulerFrequency(pattern.toString());

TimeRange timeRange = DateUtils.getTimeRangeByFrequency(frequency,
pattern.toString());

String tableName = getTableName(frequency);
log.info("Recognised frequency is " + frequency + ". Querying table: "
+ tableName);

List<ConversionError> errors = dataConversion.convertData(tableName,
timeRange);

long stopTime = System.currentTimeMillis();
log.info(String.format("Conversion took %s ms, -> %s minutes.",
stopTime - startTime,
        ((stopTime - startTime) / 60000)));
log.info("Conversion ended with " + errors.size() + " errors.");
sendEmail(errors);

log.info("Clearing cache.");
dataConversion.clearCache();
```

V ukázce kódu výše je vidět vše, co jsme popsali. Zjistí se frekvence, na základě té se vybere časový úsek a název tabulky, a spustí se konverze. Dále je tu vidět stopování času konverze a výpisu této informace. Zajímavá je metoda sendEmail, která přijímá seznam chyb. Ta, v případě chyb vzniklých při konverzi, dá vědět administrátorovi e-mailem co se stalo, aby byl o problému informován a mohl jej řešit. Poslední řádek vyčistí cache, aby nezabírala paměť. Tímto probíhá časový úkol, který provádí automatickou konverzi.

5.4.6 Vizualizace

V této kapitole se budeme zabývat vzhledovou částí aplikace. Vizualizaci si zde představíme jako takovou pěknou tečku za implementací. K vykreslení grafu zde použijeme JavaScript knihovnu D3 (Data-Driven Documents). Tuto knihovnu si zde nebudeme dopodrobna teoreticky rozebírat a pouze si představíme jednotlivé funkce, které budou pro vizualizaci potřeba. Implementace se inspiruje ukázkovým

příkladem z dokumentace Spring Data Neo4j uvedeném zde: https://github.com/neo4j-examples/movies-java-spring-data-neo4j-4.

K vizualizaci grafu bude zapotřebí lehké úpravy dat z grafové databáze v serverové části aplikace, tedy v Java kódu, tak, aby vyhovovala D3 formátu, a poté samotná implementace JavaScriptu. Úprava v serverové části znamená přidání metody na získání celého grafu do RelationshipRepository. Opět tak učiníme přidáním @Query.

```
@Query("MATCH (n)-[r]->(m) RETURN n as application, m as method, r as
rel")
```

Pro práci s touto metodou si vytvoříme třídu VisualisationImpl, která bude implementací rozhraní Visualisation. Té vytvoříme metodu visualiseGraph. Nyní je potřeba si na úrovni Javy upravit data na D3 formát. Žádoucí formát, je v podstatě seznam vrcholů a vztahů, kde vrcholy mají popisek (label) a vztahy mají indexy koncových bodů.

Úprava spočívá ve vytvoření dvou seznamů. Jeden pro vrcholy a druhý pro vztahy. Následně pak procházíme jednotlivé vztahy z databáze, které mají vždy vzor vrchol – vztah – vrchol. V každé iteraci pak přidáváme aplikace a metody do vrcholů. Přidání do vrcholů probíhá tak, že se nejdříve dotážeme na index daného vrcholu ze seznamu a pokud se nám vrátí, znamená to, že již v seznamu je a nemusíme jej přidávat. Pokud ne, přidáme jej do seznamu. V obou případech si vrácený index ukládáme do paměti, jelikož tyto indexy pak slouží k určení koncových bodů vztahu, který si uloží index počátečního vrcholu a index koncového vrcholu.

Poté vytvoříme metodu v REST kontroléru mapovanou na URL /api/graph, ve které se bude volat metoda visualiseGraph. Ve chvíli kdy máme takto připravená data, provedeme implementaci kódu v Javascriptu. Vytvoříme si soubor visualisation.js do webapp/WEB-INF/js a přidáme jej společně s knihovnou D3 do šablony visualisation.html.

```
<script src="http://d3js.org/d3.v3.min.js"
type="text/javascript"></script>

<script type="text/javascript" src="../js/visualisationD3.js"
th:src="@{js/visualisationD3.js}"></script>
```

Navíc je ještě potřeba přidat element s id="graph" na kterém budeme tvořit. Na tento element, již za použití D3 knihovny, přidáme další element s názvem svg.

```
var svg = d3.select("#graph").append("svg")
    .attr("width", "100%").attr("height", "100%")
    .attr("pointer-events", "all");
```

Nejprve jsme pomocí d3 vybrali náš element s ID graph a poté jsme v něm vytvořili element svg s atributy výška, šířka a pointer-events.

Pro pohyblivost a práci s grafem budeme používat force layout z knihovny D3, který umožňuje simulovat fyzické síly působící na částice. Náš graf se pak bude chovat „živě“ a budeme s ním moci hýbat. Připravíme si tedy force layout do proměnné.

```
var force = d3.layout.force()
    .charge(-200).linkDistance(30).size([width, height]);
```

V ukázce kódu výše vidíme nastavení hodnoty charge, ta slouží k tomu, aby se při pohybu grafu uzly rozložily a nepřekrývaly se. Záporná hodnota definuje, jak silně se mají uzly odpuzovat, kdežto kladná hodnota způsobí jejich přitahování. LinkDistance pravděpodobně netřeba vysvětlovat, jedná se o vzdálenost mezi propojenými uzly. Dále pomocí funkce d3.json provoláme naše REST API.

```
d3.json(URL+"api/graph", visualise);
```

Visualise je název funkce, ve které se data zpracují. Obsah této funkce je rozepsán dále. Jako první předáme získaná data layoutu force ke zpracování.

```
force.nodes(graph.nodes).links(graph.links).start();
```

V parametrech funkce vidíme graph, to jsou naše data ze serveru, která jsme upravovali. Metoda start spustí asynchronní propočet, který sestaví náš graf. Nyní je potřeba vytvořil elementy, které budou graf tvořit. Připravíme si kód pro uzel grafu.

```
var node = svg.selectAll(".node")
    .data(graph.nodes).enter()
    .append("circle")
    .attr("class", function (d) {
        return "node " + d.label
    })
    .attr("r", function (d) {
        return d.label == ("application") ? 15 : 10;
    }).call(force.drag);
```

Z každého uzlu se vytvoří kruh, tomu se dynamicky přiřadí CSS třída na základě typu vrcholu (label je buďto method nebo application), dále se přidá atribut r, který značí poloměr kruhu a ten jsme si podmínili tím, zda se jedná o aplikaci či metodu. Chceme totiž, aby aplikace byla v grafu větší než metoda. Poslední funkcí je zavolání force.drag, pomocí které můžeme s uzlem hýbat. Ještě si navíc přidáme title, aby alespoň při najetí myší bylo vidět, o jakou aplikaci či metodu se jedná.

```
node.append("title").attr("class", "title")
    .text(function (d) {
        return d.title;
    })
```

Poté takto přidáme i vztahy.

```
var link = svg.selectAll(".link")
    .data(graph.links).enter()
    .append("line").attr("class", function (d) {
        return "link " + d.type
    });
```

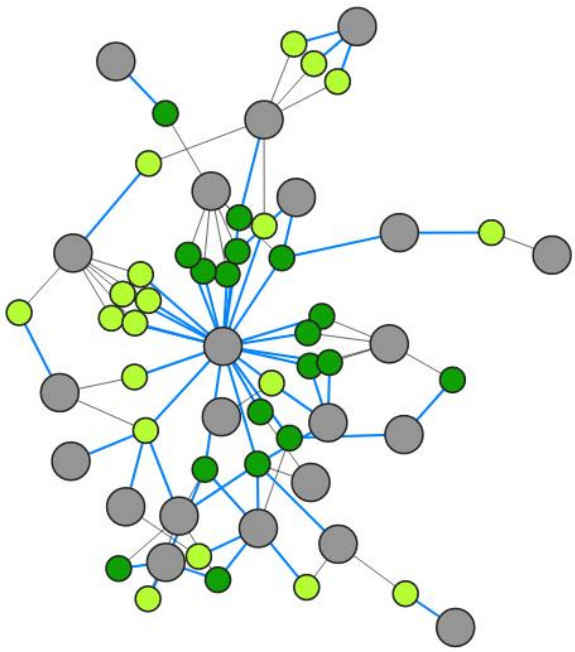
Pro každý vztah se přidá element line, s CSS třídou dynamicky vytvořenou obdobně jako u vrcholů. Tyto CSS třídy takto rozdělujeme, abychom mohli barevně odlišit

typy entit. Nakonec přidáme událost tick, která se bude starat o překreslení grafu při změně.

```
force.on("tick", function () {
    link.attr("x1", function (d) {
            return d.source.x;
        })
        .attr("y1", function (d) {
            return d.source.y;
        })
        .attr("x2", function (d) {
            return d.target.x;
        })
        .attr("y2", function (d) {
            return d.target.y;
        });

    node.attr("cx", function (d) {
            return d.x;
        })
        .attr("cy", function (d) {
            return d.y;
        });
});
```

JavaScript soubor jsme společně s knihovnou D3 přidali do visualisation.html a tento pohled je mapován na „/“. Spustíme si tedy aplikaci a vložíme data z testovacího skriptu zadáním stejné adresy jako v případě, kdy jsme testovali REST API. http://localhost:8080/api/insert?year=2016&month=06&day=01&hour=12

To spustí převod dat do grafové databáze. Pokud se vše převedlo v pořádku, zadáním adresy localhost:8080 se dostaneme na náhled visualisation.html a uvidíme, zda se náš graf opravdu zobrazí.



Obrázek 10 Výsledný graf vizualizace [zdroj: vlastní]

Graf se zobrazil a je možné s ním hýbat. Vzdálenost vztahů se zdá přijatelná a náboj také, i když některé vrcholy se překrývají. Co se týče správnosti, namátkovou kontrolou pomocí title, který zobrazí název entity, vše vypadá správně spojené s tím, čím má. Styly grafu jsou napsané tak, že šedé vrcholy jsou aplikace. Modré vztahy označují vztah „konzumuje“ a černé „poskytuje“. Zelené vrcholy jsou pak metody, barevně dále odlišené na verze 100(světle zelené) a 110 (tmavě zelené).

6 Shrnutí praktické části

Praktická část začala analýzou požadavků na aplikaci. V této analýze jsme si přiblížili, jakou funkcionalitu přesně je požadováno implementovat, v jakém jazyce bude aplikace napsána, jakou implementaci Java Servletu budeme používat a také typ grafové databáze, se kterou budeme v aplikaci pracovat. V další kapitole jsme provedli analýzu integrační platformy, vysvětlili si, jak funguje, jaké systémy jí tvoří a jak vypadá komunikace mezi těmito systémy. Poté jsme se seznámili se způsobem ukládání informací o těchto interakcích mezi systémy a představili si strukturu tabulky relační databáze, která tyto informace obsahuje.

Tato analýza byla základem pro návrh modelu grafové databáze, ve kterém jsme z funkčních požadavků vytyčily jednotlivé entity a vztahy, které se budou v grafové databázi vyskytovat. Poté jsme se mohli přesunout k návrhu struktury samotné aplikace, ve které jsme navrhli model tříd a jednotlivé komponenty u kterých jsme přiblížily jejich role, funkcionalitu, a způsob, jakým budou využívat ostatní komponenty k dosažení hlavní činnosti.

Následovala implementace, která začala konfigurací sestavení aplikace a správou závislostí. Vysvětlili jsme si zde, jakým způsobem lze do našeho projektu integrovat knihovny třetích stran a jak je možné s nimi dále pracovat. Dále také jak vytvořit výsledný archiv WAR a jak lze ovlivnit obsah tohoto archivu. Poté jsme si představili vypisování informací o událostech, neboli logování, a ukázali si konkrétní konfiguraci logování v naší aplikaci.

S takto připraveným výpisem pro kontrolu, jsme využily informací z teoretické části o frameworku Spring ke konfiguraci aplikačního a webového kontextu. V aplikačním kontextu jsme na názorných ukázkách kódu viděli, jak vypadá konfigurace napojení na testovací instanci relační a grafové databáze.

V další části jsme si vysvětlili, jak v aplikaci funguje testování zdrojového kódu a k čemu testování obecně je. Na příkladu jsme si ukázali, jak vytvořit JUnit test a otestovali v něm správnost konfigurace a spuštění testovacích databází.

Nakonec jsme provedli implementace jednotlivých komponent z návrhu, které řešily hlavní problematiku aplikace, specifickou pro naši doménu.

7 Závěr

Cílem této bakalářské práce bylo vytvořit aplikaci, která převede data z relační databáze do databáze grafové a nad daty z této databáze vystaví webové rozhraní, které data poskytne v univerzálním formátu nezávislém na jazyce aplikací, jež budou rozhraní využívat. Samotnou realizaci předcházelo teoretické přestavení jednotlivých technologií a metodik, souvisejících s vývojem webových aplikací. Dále jsme v práci provedli analýzu požadavků, která vytyčila přesnější specifikace výsledné aplikace a vytvořila tak základ pro návrh aplikace. Poté následovala analýza zdrojových dat relační databáze, na základě které jsme provedli podrobný návrh modelu grafové databáze. Následně jsme provedli návrh struktury aplikace, ve které jsme přiblížili funkcionalitu jednotlivých komponent. Implementace aplikace poté vycházela z těchto návrhů a ve výsledku se podařilo uskutečnit zadaný cíl práce. Ze zadaných cílů se dále podařilo tento převod zautomatizovat a také implementovat vizualizaci dat grafové databáze. Výsledná aplikace tedy splňuje veškeré stanovené cíle zadané v této práci, i tak se zde ale najde množství možných vylepšení a rozvinutí potenciálu aplikace. Například rozhraní aplikace by mohlo poskytovat více metod vracejících data na komplexnější dotazy a při konverzi by se mohla sbírat data o interakci v rozmezí jednoho dne. Front-end část aplikace by tak navíc mohla obsahovat lineární graf zobrazující vytíženost jednotlivých metod v čase.

Výsledná webová aplikace je nasazena a spuštěna na serveru ve firmě, pro kterou byla vytvářena. Firma aplikaci využívá a podporuje její další rozvoj.

Tato bakalářská práce může být přínosná pro každého, kdo řeší podobný problém s reprezentací dat, zakládajících se na vztazích a měl by zájem o vlastní implementaci aplikace podobného charakteru.

Slovník pojmů

Framework – Framework slouží jako podpora při vývoji softwaru. Nejčastěji se jedná o sadu nástrojů a knihoven, které abstrahují opakující se úkoly a umožňují je pře použít a tím usnadnit práci při vytváření softwaru.

Front-End – Front-end představuje uživatelské rozhraní softwaru, které je určeno pro běžného uživatele. S tímto rozhraním uživatel interaguje a rozhraní tak přijímá vstupy od uživatele a zobrazuje výstupy z datové vrstvy uživateli.

Back-End – Představuje tu část softwaru, která pracuje s daty. Tato část není přístupná běžnému uživateli. Nejčastěji je back-end propojen s front-endem, který pro něj přijímá data, se kterými pak dále podle určité logiky pracuje a výsledek navrátí na front-end k zobrazení. Může být také napřímo využíván i jiným softwarem.

Vizualizace dat – Vizualizace je postup, při kterém vyjadřujeme data pomocí obrazu.

Seznam použité literatury

Literární zdroje

[1] LACKO, Ľuboslav. Databáze: datové sklady, OLAP a dolování dat s příklady v Microsoft SQL Serveru a Oracle. Brno: Computer Press, 2003. ISBN 80-722-6969-0.

[2] RICHARDSON, Leonard a Sam. RUBY. *RESTful web services*. Farnham: O'Reilly, c2007. ISBN 0596529260.

[3] ROBINSON, Ian, James WEBBER a Emil EIFREM. *Graph databases*. Second edition. ISBN 978-149-1930-892.

Internetové zdroje

[4] *Apache Maven*, c2002-2016 [cit. 2016-10-06]. Dostupné z: https://maven.apache.org/ guides

[5] *Spring Framework Documentation*. c2004-2016 [cit. 2017-03-25]. Dostupné z: https://docs.spring.io/spring/docs/current/spring--framework-reference/htmlsingle/

[6] *Spring: Good Relationships: The Spring Data Neo4j Guide Book*. c2010-2016 [cit. 2016-07-23]. Dostupné z: http://docs.spring.io/spring-data/neo4j/docs/4.1.0.M1/reference/html/

[7] *Spring: Introduction to Spring.* c2010-2016 [cit. 2016-07-29]. Dostupné z: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html

[8] *Spring: The IoC Container*. c2010-2016 [cit. 2016-07-29]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html

[9] *Spring: Web MVC framework*. [cit. 2016-07-23]. Dostupné z: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[10] *Tutorials Point: Basic MVC Architecture*. c2016 [cit. 2016-07-23]. Dostupné z: http://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm

Seznam obrázků

Seznam tabulek

Seznam příloh

Další přílohy

Příloha CD – Zdrojový kód aplikace s testovacími daty

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2016/2017

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Zoubek Lukáš	Jablonského 386/1, Čáslav - Čáslav-Nové Město	I1300846

TÉMA ČESKY:

Webová aplikace poskytující data o interakci aplikací v integrační platformě

TÉMA ANGLICKY:

Web application providing data about the interaction of applications in integration platform

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

1.Úvod
2.Technologie webových aplikací
3.Analýza
4.Návrh
5.Implementace
6.Závěr

SEZNAM DOPORUČENÉ LITERATURY:

1. SCHILDT, Herbert. Mistrovství - Java. Brno: Computer Press, 2014. Mistrovství. ISBN 978-80-251-4145-8.
2. KROENKE, David a David J. AUER. Databáze. Brno: Computer Press, 2015. Mistrovství. ISBN 978-80-251-4352-0.
3. ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....