ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Nástroj pro měření datové kvality pomocí SQL dotazů
Student:	Vojtěch Malý
Vedoucí:	Ing. Tereza Mlynářová, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Seznamte se s problémem datové kvality v datovém skladu.
2) Seznamte se s obecnou architekturou úloh v Jenkins CI.
3) Zhodnoťte použití technik reaktivního programování nebo knihovny Spring Integration oproti implementaci úloh v Jenkins CI.
4) V jazyce Java implementujte nástroj pro spouštění úloh měřících datovou kvalitu pomocí SQL dotazů a jejich následné vyhodnocování, přičemž SQL dotazy mohou využívat výsledky jiných SQL dotazů, a tím mohou tvořit orientovaný graf závislostí.
   Jednotlivé úlohy řeší spouštění různých typů měření, které slouží pro vyhodnocení stavu datové kvality ve zkoumaném úložišti dat.
5) Vyberte grafové algoritmy pro transformaci a procházení orientovanými grafy tak, aby jejich použití bylo použitelné ve vyvinutém nástroji na měření kvality dat.
6) Navrhněte a implementujte algoritmus pro paralelní zpracování jednotlivých úloh vycházející z analýzy z bodu 5).

Seznam odborné literatury

[1] C.Batini, M. Scannapieca. Data Quality: Concepts, Methodologies and Techniques. Springer, 2006.
[2] Jenkins. GitHub repository (2016), URL: https://github.com/jenkinsci/jenkins
[3] Jenkins. Documentation (2016), URL: https://jenkins.io/doc/
[4] M. Fisher, J. Partner, M. Bogoevici, I. Fuld. Spring Integration in Action. Manning, 2012.
[5] Nickolay Tsvetinov. Learning Reactive Programming with Java 8. Packt Publishing, 2015.
[6] Josef Kolář. Teoretická informatika. Česká informatická společnost, 2004.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 3. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Nástroj pro měření datové kvality pomocí SQL dotazů

Vojtěch Malý

Vedoucí práce: Ing. Tereza Mlynářová, Ph.D.

10. května 2017

Poděkování

Tímto bych rád poděkoval společnosti Simplity s. r. o., především vedoucí práce, Ing. Tereze Mlynářové, Ph.D., a Davidu Šťastnému, za jejich ochotu a pomoc při realizaci této bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. května 2017 .....

Odkaz na tuto práci

Malý, Vojtěch. *Nástroj pro měření datové kvality pomocí SQL dotazů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se zaměřuje na tvorbu nástroje pro spouštění úloh, jenž měří datovou kvalitu. Probrány jsou různé druhy spouštění úloh s důrazem na paralelizaci. Nástroj je serverová aplikace psaná v programovacím jazyku Java. Součástí práce je také analýza knihoven pro integraci aplikací a jejich zhodnocení.

Klíčová slova aplikace, měření datové kvality, orientovaný acyklický graf, paralelní, reaktivní programování, Spring Integration, Java

Abstract

This thesis deals with creating an engine for executing jobs, which measures data quality. Various methods of job execution are discussed with emphasis laid on parallelization. The engine is a server standalone application written in Java programming language. Analysis of libraries for application integration and their evaluation are also part of the thesis.

Keywords application, data quality measurements, directed acyclic graph, parallel, reactive programming, Spring Integration, Java

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době převážná většina firem využívá ke svému provozu informační systémy. Chyby v jejich systému mohou vést ke generování neadekvátních dat, a to stejně tak i v případě špatného používání systému. Čím větší firma je, tím operuje s objemnějším množstvím dat, a tak vzniká více dat nesprávných – snižuje se datová kvalita.

U společností, jakými jsou např. celosvětové banky, je práce s takovými neadekvátními daty problémem vedoucím až k několika milionovým ztrátám. Dle společnosti Gartner[1] a jejich studie založené na informacích z více než 140 společností, jsou průměrné roční ztráty způsobené nízkou datovou kvalitou okolo 8 miliónu amerických dolarů. V tuto chvíli přichází v úvahu měření datové kvality.

Práce je určena pro firmu Simplity s. r. o., která ji využije ve své aplikaci Quality. Tato aplikace slouží jejich zákazníkům ke sledování a měření datové kvality.

Téma jsem si zvolil, neboť společnosti čím dál více využívají znalosti získané ze svých dat. Jako příklad mohu uvést analýza nákupů zákazníků s Tesco Clubcard. Aby tyto znalosti byly pravdivé, musí mít data dobrou datovou kvalitu. Mnoho společností datovou kvalitu neřeší, ale cílem společnosti Simplity je naučit firmy porozumět datům a správně se o ně starat.

V práci se zabývám analýzou aplikace Jenkins CI, knihovny Spring Integration a reaktivním programování. Nabyté znalosti mi poslouží k implementaci nástroje na měření datové kvality, který spouští úlohy, jejichž výpočet slouží k vyhodnocování stavu datového úložiště. Úlohy obsahují mezi sebou závislosti a tvoří tak orientovaný acyklický graf.

Následující kapitola 1 zahrnuje vysvětlení pojmů *business intelligence* a datová kvalita. Dále je v této kapitole popsán vyvíjený nástroj a požadavky na jeho implementaci. Ke konci se podíváme na podobné, již existující nástroje.

Kapitola 2 je věnována popisu aplikace Jenkins CI a její architektuře. Také je představen *framework* Spring Integration a reaktivní programování –

především Reactor a Spring Webflux. Výsledkem této kapitoly je porovnání získaných vědomostí a jejich následné doporučení pro vývoj nástroje.

V následující kapitole 3 jsou popsány algoritmy pro implementaci úloh a jejich spouštění. Nejdříve je popsán způsob, jakým se budou úlohy v nástroji reprezentovat, poté algoritmy, kterými mohou být úlohy spouštěny.

Tuto práci uzavírá kapitola 4, v níž je představen způsob realizace. Kapitola je tvořena popisem zvolené technologie a klíčových tříd aplikace. Konec kapitoly se zabývá měřením a testováním nástroje, tedy různých druhů spouštění úloh, které jsou v závěru porovnány.

Kapitola 1

Popis problému, specifikace cíle

V této kapitole si stručně vysvětlíme pojem *Business intelligence*, což nám umožní lépe pochopit problematiku datové kvality a více osvětlit využití vyvíjeného nástroje. V druhé části si vysvětlíme, co to je datová kvalita a její dimenze. Poté navážeme popisem vyvíjeného nástroje a určíme si požadavky, které má nástroj splňovat. Na konci kapitoly zjistíme, jestli již existují nějaké podobné nástroje.

1.1 Business intelligence

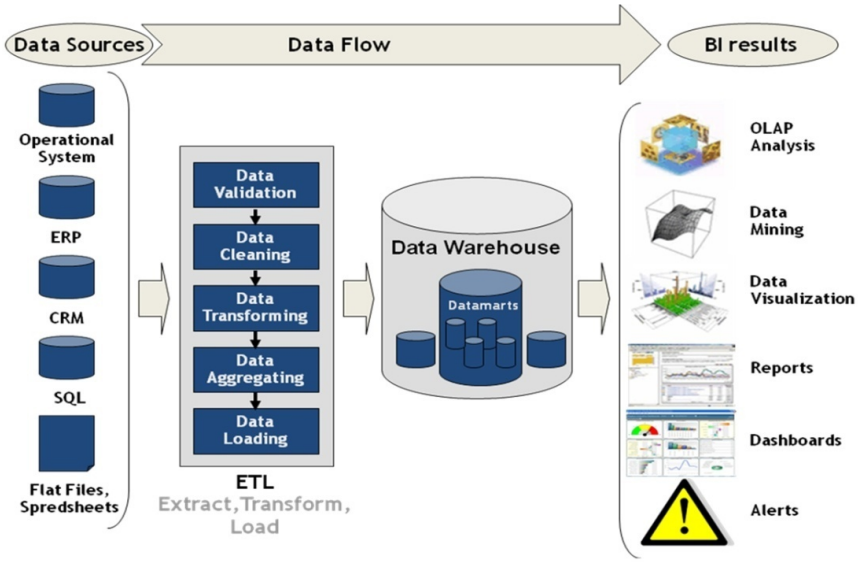
Business intelligence (dále jen BI) je množina konceptů a metodik, které se používají při práci s firemními daty. Účelem těchto konceptů a metodik je konvertovat velké množství dat na znalosti, a tím zlepšit rozhodovací proces, který zvýší obchodní úspěch společnosti.[2]

Proces BI je tvořen z několika částí, viz obrázek 1.1. Na začátku BI procesu je zapotřebí datových zdrojů (angl. *data sources*), kterými mohou např. být databáze, tabulky, soubory a další. Následně jsou data z těchto zdrojů získána, transformována a nahrána do datového skladu. Této části se říká ETL (*extract, transform, load* – získat, transformovat, nahrát).

Na první pohled je pojem datový sklad (ang. *data warehouse* – DWH) poměrně jasný. Je to velké množství dat shromážděných na jednom místě. Pro jistotu ale přidávám formální definici datového skladu:

„Datový sklad je podnikově strukturovaný depozitář subjektově orientovaných, integrovaných, časově proměnných historických dat použitých pro získávání informací a podporu rozhodování.“[2]

Nad naplněným datovým skladem probíhají již různé BI metodiky, z jejichž výstupů (např. analýzy a reporty) si firmy mohou najít skryté souvislosti či predikce a zlepšit tím svoje postavení na trhu.



Obrázek 1.1: Průběh procesu *business intelligence*[3]

Měření datové kvality má své uplatnění ve fázi ETL při integraci dat do datového skladu, kdy se data mohou ještě upravovat, a tím datovou kvalitu zvýšit. Nekvalitní data v datovém skladu mohou vést ke špatné analýze a ke špatným rozhodnutím firmy následovaných finanční ztrátou.

1.2 Datová kvalita a její dimenze

Definovat datovou kvalitu (angl. *data quality*) není lehké. Obecně lze říci, že data jsou kvalitní, pokud splňují požadavky uživatelů na jejich použití.[4]

Datová kvalita se určuje z pohledu několika částí, tedy tzv. dimenzí. Existuje mnoho dimenzí datové kvality, nicméně my si představíme jen čtyři nejzákladnější na příkladu z knihy [4]:

Id	Název	Režisér	Rok	#Předělávek	Poslední
1	Casblanca	Weir	1942	3	1940
2	Dead poets society	Curtiz	1989	0	NULL
3	Rman Holiday	Wylder	1953	0	NULL
4	Sabrina	null	1964	0	1985

Tabulka 1.1: Ukázka dat s problémy datové kvality

V tabulce 1.1 jsou popsány vztahy mezi filmy – jejich jméno, režisér, rok produkce, počet předělávek a rok vydání poslední předělávky. Problémy s da-

tovou kvalitou jsou v buňkách tabulky zvýrazněny modře.

Ve sloupci `Název` je chyba s filmem `Rman Holiday`. Je zde překlep, film se totiž jmenuje `Roman Holiday`. Tato chyba spadá do dimenze **přesnosti (*accuracy*)**. Problém stejné dimenze, tedy přesnosti, je i prohození režisérů mezi filmy 1 a 2. `Wier` režíroval film 2, zatímco `Curtiz` film 1.

U filmu č. 4 režisér chybí zcela. Tato chyba je z dimenze **úplnosti (*completeness*)**. Dále je problém u filmu č. 4 s dimenzí **aktuálnosti (*concurrency*)**, neboť počet předělávek je stále roven 0, i když byla alespoň jedna dotočena.

Poslední dva problémy jsou z dimenze **konzistence (*consistency*)**. U filmu č. 1 nemůže být `Poslední` menší než `Rok`, u filmu č. 4 zase nemůže být `Poslední` jiná hodnota než `NULL`, protože počet předělávek je roven 0.

Přesnost, úplnost, aktuálnost a konzistence jsou tedy čtyři dimenze datové kvality. Existuje jich více, jako např. integrace či duplikace, ale pro základní představu toho, co to datová kvalita je, byl tento příklad dostačující.[4]

1.3 Hlavní cíle

Cílem bakalářské práce je vytvořit nástroj pro spouštění úloh měřících datovou kvalitu. Jednotlivé úlohy řeší spouštění různých typů měření, které slouží pro vyhodnocení stavu datové kvality ve zkoumaném úložišti dat.

Základním typem úloh je SQL (*Structured Query Language*) dotaz nad databází, jehož výsledkem je číslo. Výsledky těchto SQL dotazů však mohou být obsaženy v jiném parametrizovaném SQL dotazu. Viz ukázka v tabulce 1.2, kde ID je unikátní identifikátor úlohy. Úloha č. 1 a 2 jsou pak základní SQL úlohy a úloha č. 3 je parametrizovaná SQL úloha, která využívá ve své WHERE podmínce výsledku z úlohy 1.

ID	**SQL dotaz**
1	select avg(discount) from orders
2	select min(discount) from orders
3	select count(*) from orders where discount > {1}

Tabulka 1.2: Ukázka SQL úloh

Třetím typem úlohy, kterým se budu v práci zabývat, je agregovaný výsledek z výsledku jiných úloh. Vstupem mu budou výsledky jiných úloh (jak SQL, tak parametrizovaných SQL, či jiných agregovaných) a váha každé nich, z čehož se vypočítá vážený průměr podle vzorce 1.1, kde x_i je výsledek a ω_i váha dané úlohy i:

$$\bar{x} = \frac{\sum_{i=1}^{n} \omega_i x_i}{\sum_{i=1}^{n} \omega_i}. \tag{1.1}$$

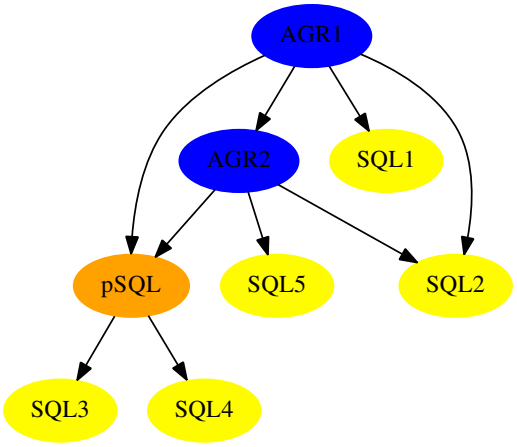
Vzhledem k jednotlivým závislostem mezi úlohami vzniká orientovaný graf. Přikládám definici orientovaného grafu dle [5]:

Definice: *„Nechť H a U jsou libovolné disjunktní množiny a $\sigma : H \to U \times U$ zobrazení.* ***Orientovaným grafem*** *nazveme uspořádanou trojici $G = \langle H, U, \sigma \rangle$, prvky množiny H nazýváme orientovanými hranami grafu G, prvky množiny U uzly grafu G a zobrazení σ incidencí grafu G.“*

Jednotlivé úlohy tedy tvoří uzly grafu a závislosti mezi nimi představují hrany grafu. Jestliže pro $h \in H$ je $\sigma(h) = (u, v)$, pak úloha u potřebuje ke svému výpočtu výsledek z úlohy v.

Vstupem máme zaručeno, že graf nebude obsahovat cyklus (kružnici). Cyklem v grafu rozumíme posloupnost uzlů a hran $S = \langle u_0, h_1, u_1, \ldots, h_n, u_n = u_0 \rangle$, kde uzly $u_0, \ldots, u_{n-1}$ jsou navzájem různé uzly grafu G, a kde $h_i \in H$, $\sigma(h_i) = (u_{i-1}, u_i)$ pro každé $i = 1, 2, \ldots, n$. Graf bez cyklu nazýváme **acyklický**.[5]

Výsledný graf může tedy vypadat např. jako na obrázku 1.2, v němž žlutý uzel značí SQL úlohu, oranžový parametrizovanou SQL úlohu a modrý agregovanou úlohu.



Obrázek 1.2: Ukázková struktura úlohy AGR1

Některé úlohy jsou na sobě nezávislé a nástroj je tedy bude spouštět paralelně. Na obrázku 1.2 se mohou úlohy vykonávat např. v takovémto pořadí:

1. Všechny SQL úlohy v jakémkoliv pořadí.
2. Parametrizovaná SQL úloha pSQL po vypočítání úloh SQL3 a SQL4.
3. Agregovaná úloha AGR2 po vypočítání úloh pSQL, SQL5 a SQL2.
4. Agregovaná úloha AGR1 po vypočítání úloh pSQL, AGR2, SQL1 a SQL2.

Nástroj musí do budoucna počítat s tím, že bude integrován s jinou aplikací, která mu bude dodávat úlohy ke spuštění. Nástroj bude této aplikaci zpět posílat informace, kdy daná úloha byla zařazena do spouštěcí fronty, zda-li při výpočtu úlohy došlo k chybě, anebo je již dokončená, popř. jakého výsledku dosáhla. Integrace by měla probíhat přes REST (*Representational state transfer*) rozhraní, které využívá HTTP (*Hypertext Transfer Protocol*). Integrace však není součástí práce, jelikož by rozsahem přesáhla velikost bakalářské práce. Požadavkem je jen najít správný *framework*, nebo-li aplikační rámec, který obsahuje podpůrné programy a rozhraní.

1.4 Požadavky nástroje

Shrnutí a upřesnění jednotlivých požadavků na nástroj:

1. Nástroj definuje základní rozhraní úloh, aby byl schopen používat dodatečně implementované úlohy.
2. Nástroj obsahuje tři základní implementace úloh:
   a) SQL dotaz nad databází s výsledkem;
   b) Parametrizovaný SQL dotaz obsahující výsledky jiných úloh;
   c) Výpočet agregovaného výsledku z výsledků jiných úloh.
3. Úlohy tvoří orientovaný acyklický graf a musí tak být převedeny do dobře zpracovatelného formátu.
4. Nástroj úlohy spouští a výpočet nezávislých úloh probíhat paralelně.
5. Nástroj je implementován v jazyce Java.
6. Nástroj nemusí být připraven na integraci s jinou aplikací, ale je potřeba zanalyzovat možnosti integrace přes rozhraní REST.

1.5 Podobné nástroje

V současné době neexistuje žádný nástroj jenž by vyhovoval zadaným požadavkům.

Na trhu jsou dostupné aplikace, které měří a monitorují datovou kvalitu. Mezi ně např. patří Data Quality od Informatica[6], InfoSphere Information Server for Data Quality od IBM[7], Ataccama[8] či Trillium Quality[9]. Žádný z těchto softwarů ale není *open source*, tedy nemají otevřený zdrojový kód, a nelze tedy zjistit, jakým způsobem je u nich datová kvalita měřena.

Co však spadá pod licenci *open source* a podobá se očekávanému nástroji způsobem zpracování úloh, je aplikace Jenkins CI[10] a její systém *job* a *pipeline*. Detailněji je Jenkins popsán v následující kapitole 2.1.

Kapitola 2

Analýza knihoven

Úvod kapitoly je tvoře popisem aplikace Jenkins a jeho architektury. Následně se zabývám analýzou knihovny Spring Integration a reaktivním programováním. Získané informace zhodnotím a určím jestli je dobré použít část aplikace Jenkins, Spring Integration či Spring Webflux.

2.1 Jenkins CI

Jenkins CI je *open source* server, který se používá k automatizování různých druhů úloh, jakými jsou např. sestavení, otestování a nasazení softwaru. Je vytvořen v programovacím jazyku Java a lze ho tedy používat na jakémkoliv stroji, na němž je nainstalován Java Runtime Environment (JRE).[11] Hlavní využití programu Jenkins spočívá v automatizování vývoje softwaru pomocí kontinuální integrace (angl. *Continuous integration* – CI).

Kontinuální integrace je založena na častém kompilování zdrojového kódu projektu, obvykle následovaném spuštěním testů.[12] Nejčastější případ nastává ve chvíli, kdy programátor pošle novou část kódu do projektového repositáře (např. GIT), Jenkins poté vytvoří několik úloh (angl. *jobs*), které se spouští postupně za sebou, například:

1. Stažení kódu z repositáře;
2. sestavení projektu;
3. spuštění testů;
4. vyhodnocení kvality kódu;
5. poslání emailu s výsledky autorovi.

Velkou výhodou Jenkins CI je jeho velká popularita, která vede k tomu, že komunita vývojářů je velmi aktivní, a tudíž existuje a stále vzniká velké

množství doplňků, které otevírají mnoho možností jakým způsobem Jenkins využít.[12]

Základním stavebním kamenem aplikace Jenkins je *job*, jinak také *project*, nebo-li úloha. Je to práce, kterou má Jenkins vykonat – jaký příkaz pustit v příkazové řádce, testování projektu, publikování výsledků testů, atd. *Job* po skončení vytváří výsledek zvaný *build*.[11]

Druhým důležitým prvkem jsou tzv. *pipelines*, které umožňují velikou uživatelskou konfiguraci. Jsou to jednotlivé kroky, které Jenkins spouští po logických celcích. Vznikají zapsáním kódu pomocí Pipeline DSL do souboru *Jenkinsfile*. Pipeline DSL (Doménově specifický jazyk) vychází z jazyka Groovy a lze v něm tedy použít známe konstrukce jako podmínky, cykly či funkce. Nejlepší bude ukázat příklad zapsané *pipeline* [11]:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml'
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

Příkaz na řádce 2 říká, aby Jenkins alokoval nějaký uzel, v němž bude *pipeline* spuštěna.

Řádka 4 rozdělí *pipeline* do jednotlivých logických celků, definovaných slovem *stage* 5.

Steps 6 jsou kroky, které se mají ve *stage* provést. Mohou být vykonány i paralelně.

Sh 7 spouští shellový příkaz.

Junit 13 je krok, který lze vykonat pomocí doplňku jUnit plugin, jehož úkolem je sjednotit výsledky všech testů.

Job i *pipeline* mohout mít nastavený *trigger*, nebo-li spouštěč. *Trigger* je kritérium, které splněním spustí jiný *job* nebo *pipeline*. *Trigger* může být např. dosažení určitého času, aktualizace repositáře či dokončení jiného *job* nebo *pipeline*.[11]

Dalšími důležitými pojmy jsou *downstream* a *upstream*. *Downstream* je *job* nebo *pipeline*, který se spustí během vykonávání jiného *job* nebo *pipeline*. *Upstream* je opakem, tedy je to *job* či *pipeline*, který spouští během svého vykonávání jiné *jobs* a *pipelines*.[11]

Jenkins je sestaven z několika *nodes* (česky uzlů), tedy strojů, které jsou schopny spouštět *jobs* nebo *pipelines*. Jenkins pracuje na architektuře Master/Slave, tzn. obsahuje jeden centrální uzel zvaný *master*, který má uloženou konfiguraci, stará se o nahrávání doplňků, zobrazuje uživatelské rozhraní a rozdává práci svým *slaves* (česky otrokům). *Slaves* (*slave agents*) jsou uzly, které jsou připojeny k uzlu *master*, a spouští úlohy, které jim *master* zadá.[11]

Podle výše zmíněných vlastností aplikace Jenkins lze najít podobnost s nástrojem, který je cílem této práce. Nástroj má také za úkol spouštět nějaké úlohy a zpracovávat jejich výsledky. Výpočet jedné úlohy může spustit úlohu jinou. Takový nástroj bude pracovat jako *slave* pro jinou aplikaci, s níž bude spojený a která mu bude posílat úlohy ke spuštění. Dokonce i *pipelines* lze přirovnat ke zpracování úloh, jelikož nástroj musí po každém logickém celku (zařazení do fronty, spuštění, dokončení) odeslat nějaké informace do uzlu *master*.

2.2 Spring Integration

Spring Integration je *open source framework* pro integraci podnikových aplikací pomocí ***message-driven*** (zprávami řízené) architektury. Základem *message-driven* architektury je komunikace mezi jednotlivými částmi aplikací prostřednictvím zpráv (anglicky *messages*). To vede k jednoduššímu řešení problémů díky podobnosti s reálným světem, v němž jsme zvyklí pracovat jako *message-driven* – odpovídáme na telefonáty, emaily, zprávy nebo na nějaké události.[13]

Spring Integration je postaven na dvou gigantech, jimiž jsou Spring Framework[14], jenž je populární *framework* pro vytváření podnikových Java aplikací, a kniha *Enterprise Integration Patterns*[15], která je standardem v oblasti integrace.[13]

2.2.1 Architektura

Spring Integration se skládá ze dvou částí.

První je *messaging framework*, který umožňuje *message-driven* komunikaci v rámci aplikace, tedy v jedné instanci Java Virtual Machine (JVM). Jednotlivé komponenty mohou mezi sebou komunikovat prostřednictvím zpráv a nemusí řešit serializaci (např. převod do *Extensible markup languag*e (XML)), neboť zprávy většinou obsahují *plain old Java object* (POJO).[13]

Druhou částí je integrace několika aplikací – více instancí JVM. Ta probíhá pomocí různých adaptérů, které transformují obsah zpráv na tvar, který cílová aplikace přijímá.[13] Spring Integration 4.3 je dodáván včetně mnoha adaptérů např:

- Filesystem, FTP, FTPS nebo SFTP,
- HTTP (REST),
- Java Database Connectivity (JDBC),
- Mail (POP3, IMAP a SMTP),
- Transmission Control Protocol (TCP),
- Twitter,
- User Datagram Protocol (UDP),
- Web Services (SOAP),
- Web Sockets.

Více viz tabulka [16].

Ze Spring Integration bychom v našem nástroji využili právě její druhou část, tedy integraci více JVM, kde náš nástroj musí komunikovat s jinou aplikací.

2.2.2 Hlavní komponenty

Message (nebo-li zpráva) je jednotka informace posílána mezi jednotlivými komponentami zvanými *message endpoints*. *Message* se skládá z *header* (česky hlavička) a *payload* (česky náklad). *Header* obsahuje data potřebná pro běh *frameworku* jako je ID číslo nebo návratová hodnota. *Payload* reprezentuje data, která mají být skrz *message* přenesena, tedy již výše zmíněné POJO nebo XML či jen textový řetězec.[13]

Message channel je spojení mezi několika *message endpoints*. Po tomto spojení jsou posílány zprávy.[13] Existují čtyři možnosti jak se *message channel* může chovat:

- **Synchronně**

  Message je odeslána a odesílatel čeká na odpověď od příjemce.

- **Asynchronně**

  Message je odeslána a přidána do fronty příjemci. Odesílatel nečeká se na odpověď.

- **Point-to-point**

  Spojení mezi dvěma endpointy. *Message* je vždy doručena, nebo nastala chyba.

- **Publish-subscribe**

  Publisher vysílá zprávy, které zachytává *subscriber*. Počet poslouchajících *subscriber* může být mezi 0 až n. Odesílatel se nezajímá ani o doručení, ani o případné chyby.

Message endpoints jsou komponenty, mezi nimiž vzniká spojení – *message channel*. V těchto místech probíhá zpracování *messages*.[13] Existuje několik typů *message endpoints*:

- **Transformer**

  Transformer je zodpovědný za přeměnu obsahu *message* na jinou a vrácení upravené *message*. Nejčastěji se jedná o přeměnu *payload* (např. z formátu XML do `java.util.String`) ovšem lze editovat i *header*.[17]

- **Filter**

  Filter rozhoduje jestli má být *message* poslána do *message channel*. Obsahuje testovací metodu s návratovým typem *boolean*, která rozhoduje dle *header* či *payload*, jestli *message* poslat, nebo ne.[17]

- **Splitter**

  Splitter přijímá *message* a rozděluje ji do několika dalších *messages*, které pošle dál. Tyto *messages* jsou poté zpracovávány najednou.[13]

- **Aggregator**

  Aggregator můžeme jednoduše popsat jako opačný *Splitter*, tzn. že čeká na příchozí *messages*. Až přijme všechny očekávané, spojí je do jedné *message*.[13]

- **Service activator**

  Service activator je *endpoint*, jenž příchozí *message* pošle nějaké metodě. Tato metoda může message přečíst, zpracovat či upravit a poslat ji dále.[17]

- **Channel adapter**

  Channel adapter je *endpoint*, který spojuje *message channel* s jiným systémem (aplikací). *Channel adapter* většinou provádí nějakou konverzi *message* během jejího odeslání/přijímání. Spring Integration poskytuje několik hotových *channel adapters*, viz 2.2.1.[17]

2.2.3 Integrace pomocí REST

Nyní se podívejme na příklad komunikace dvou aplikací pomocí Spring Integration. Z kapitoly 2.2.1 víme, že existuje několik způsobů, jimiž mohou komunikovat. Jelikož všechny aplikace ve firmě Simplity, pro níž je nástroj vyvíjen, komunikují přes rozhraní REST, tak i v tomto příkladě uvedu komunikaci pomocí REST, skrz `HttpInboundAdapter`.

Příklad je převzatý z [18] a ukáže jak jednoduše poslat *Multipart HTTP* požadavek přes Spring `RestTemplate` a přijmout ho se Spring Integration `HttpInboundAdapter`.

Strana klienta je velmi jednoduchá:

```
RestTemplate template = new RestTemplate();
String uri =
   "http://localhost:8080/inboundAdapter.htm";
Resource s2logo =
    new ClassPathResource("logo.png");
MultiValueMap map = new LinkedMultiValueMap();
map.add("company", "SpringSource");
map.add("company-logo", s2logo);
HttpHeaders headers = new HttpHeaders();
headers.setContentType(new MediaType("multipart",
   "form-data"));
HttpEntity request = new HttpEntity(map, headers);
ResponseEntity<?> httpResponse =
   template.exchange(uri, HttpMethod.POST, request,
   null);
```

Vytvoříme `MultiValueMap` a vložíme do ní data – název firmy a její logo. Zadáme URI na kterou chceme požadavek poslat, a `RestTemplate` se postará o zbytek, tedy odešle HTTP požadavek obsahující `MultiValueMap`.

Na straně serveru musíme vytvořit XML konfiguraci, která nastaví příchozí `HttpInboundAdapter`, připojí k němu *message channel* a *endpoint service activator*, který používá `MultipartReceiver`:

```
<int-http:inbound-channel-adapter
   id="httpInboundAdapter"
channel="receiveChannel"
path="/inboundAdapter.htm"
supported-methods="GET, POST"/>

<int:channel id="receiveChannel"/>

<int:service-activator input-channel="receiveChannel">
    <bean class="MultipartReceiver"/>
</int:service-activator>

<bean id="multipartResolver"
   class="samples.CommonsMultipartResolver"/>
```

`HttpInboundAdapter` bude odchytávat požadavky a konvertuje *message* obsahující `LinkedMultiValueMap`. Následně je zpracuje pomocí *service activator* a jeho metody `receive()`:

```
public void receive(LinkedMultiValueMap<String,
   Object> multipartRequest){
    System.out.println("### Successfully received
       multipart request ###");
    //Zde lze z multiparRequest získat poslaná data
}
```

2.3 Reaktivní programování

Reaktivní programování je programovací paradigma orientované okolo šíření změn a asynchronních datových toků.[19]

Nechť máme příkaz $c = a + b$. Za použití imperativního paradigmatu, by se do proměnné c uložila hodnota z proměnných a a b. Následovali by po tomto příkazu změny proměnných a nebo b, tak by c zůstalo pořád stejné. Ovšem za použití reaktivního paradigmatu by se hodnota v proměnné c aktualizovala vždy při změně a nebo b.[19]

Názorným příkladem může být aplikace Microsoft Excel. Máme-li buňky $A1, B1$ a $C1 = SUM(A1, B1)$, poté se buňka $C1$ aktualizuje vždy při změně $A1$ nebo $B1$.[19]

V dnešní době se reaktivní programování používá především za použití datových toků (angl. *stream*) a funkcionálního programování.

2.3.1 Reaktivní programování a Java 8

V době před vydáním Java verze 8 byl problém tvořit asynchronní aplikace, neboť hlavním elementem, jak zaručit asynchronnost, bylo zpětné volání (nebo-li *callback*). To však často vedlo k termínu, jenž se udává jako *callback hell* (*hell* je česky peklo), tzn. vnořování několika zpětných volání do sebe, což vede k nečitelnosti kódu.[19]

Java 8 však přinesla API (*Application Programming Interface* – aplikační rozhraní) pro funkcionální programování a lambda funkce, které zpřehledňují a zestručňují zápis zpětných volání. Dále představila novou komponentu *Stream*. Ta umí efektivně využívat datové toky, přistupovat k datům s velmi malou odezvou či běžet paralelně, bez nutného nastavení vláken. *Stream* má nevýhodu, že si neumí poradit s operacemi, které mají vyšší odezvu (např. I/O operace). V tuto chvíli nastupují reaktivní API, jako je Reactor nebo RxJava, obsahující reaktivní toky.[20]

Reaktivní toky jsou postaveny na čtyřech Java rozhraních *Publisher*, *Subscriber*, *Subscription* a *Processor*. Jedná se tedy o vzor *publisher-subscriber*, v němž *publisher* poskytuje data svým *subscribers*, které data zpracovávají. *Subscription* je životní cyklus mezi jedním *Subscriber* poslouchajícím jeden *Publisher*. *Processor* reprezentuje vztah mezi *Subscriber* a *Publisher*.[21] Reaktivní stránkou je zde ta skutečnost, že *Publisher* oznamuje svým *Subscriber* nové dostupné hodnoty.[22]

Dvě nejrozšířenější reaktivní API pro jazyk Java jsou tedy Reactor[23] a RxJava[24]. Jelikož se jedná o knihovny postavené na podobném principu, není mezi nimi výrazný rozdíl a většinou vždy lze od jedné knihovny přejít k druhé. V mé práci se však dále budu zabývat jen Reactor API, neboť Spring Framework (který je ve firmě Simplity používán) bude obsahovat v nové verzi 5 jeho plnou podporu a je na něm postaven Spring Webflux – reaktivní rozhraní pro komunikaci webových aplikací pomocí HTTP.[25]

2.3.2 Reactor

Reactor je neblokující reaktivní API pro JVM, který je plně integrován s Java 8 *Stream* a funkcionálním API.[22] Reactor poskytuje podobné operátory, jakým je Java 8 *Stream*, ale tyto pracují s jakýmkoliv datovým tokem (nejen s kolekcemi) a dovolují definovat *pipeline* (obdobné s pipeline v 2.1) transformačních operací, které se aplikují na data skrz tzv. *fluent* API a lambda funkce. Zároveň podporují jak synchronní, tak asynchronní operace a dovolují nám jednotlivé toky spojovat, rozdělovat, filtrovat a mnohem více.[20]

Reactor má dva základní reaktivní typy *Flux* a *Mono*, které implementují *Publisher*.

Flux je standardní *Publisher* reprezentující asynchronní sekvenci o 0 až N vysílaných prvků. Poskytuje tři základní metody podle vysílaného signálu – metoda `onNext()` zavolána vždy pro každý vysílaný prvek, metoda `onError()`

zavolána pokud došlo během zpracování k chybě a metoda `onComplete()` zavolána po zpracování všech prvků.

Flux obsahuje mnoho operací, s nimiž lze v něm obsažená data různě upravovat. Je jich opravdu mnoho a jejich výčet lze najít v dokumentaci.[22] Nyní se podíváme na jednoduchou ukázku, jak *Flux* vytvořit ze zadaných řetězců a jak tyto řetězce transformovat (pomocí lambda výrazu), aby byly psány velkými písmeny. Následně budou řetězce zpracovány pomocí *Subscriber*, který každý prvek vypíše na standardní výstup:

```
Flux<String> flux = Flux.just("red", "green", "blue");

Flux<String> upper = flux
    .map(String::toUpperCase)
    .doOnNext(System.out::println)
    .subscribe();
```

Mono je speciální *Publisher* vysílající maximálně jeden prvek. Proto poskytuje jen dvě základní metody, a to `onError()` a `onComplete()`, viz výše. *Mono* obsahuje jen podmnožinu operací, které má *Flux*. Jinak se používá obdobně jako *Flux*.[22]

2.3.3 Komunikace pomocí Spring Webflux

Spring Webflux je modul, který bude obsažen ve *frameworku* Spring verze 5, podporující reaktivní programování pro webové aplikace. Tento modul obsahuje podporu jak pro reaktivní HTTP a WebSocket klienty, tak pro reaktivní servery webových aplikacích skrz REST či WebSocket.[25]

WebFlux by se v cílovém nástroji použil pro integraci aplikace skrz rozhraní REST. Tedy jako v případě Spring Integration se podíváme na příklad převzatý z [25]:

Klient bude vypadat velmi jednoduše. Webflux obsahuje funkcionální, reaktivní komponentu `WebClient`, která je reaktivní a neblokující alternativou k `RestTemplate` (použita v 2.2). WebFlux obaluje tělo HTTP požadavku i odpovědi do jednoho z reaktivních typů. Zároveň umí JSON, XML a SSE serializaci a lze tedy pracovat s typovanými objekty.[25]

Stačí vytvořit `WebClient` s URL na které je spuštěn server. Poté na něm zavoláme metodu značící typ HTTP požadavku, tedy *get()*. Následně se nastaví v jakém formátu budou data serializovaná a přenesena. Nakonec je odpověď přetransformována pomocí metody `bodyToMono` do `Mono` obsahující objekty typu `Account`.

Server musí být spuštěn pomocí Spring Boot[26] verze 2.0, který podporuje Spring Webflux.

```
WebClient client =
   WebClient.create("http://example.com");

Mono<Account> account = client.get()
    .url("/accounts/{id}", 1L)
    .accept(APPLICATION_JSON)
    .exchange(request)
    .then(response ->
       response.bodyToMono(Account.class));
```

```
@RestController
public class PersonController {

    private final PersonRepository repository;

    public PersonController(PersonRepository
       repository) {
        this.repository = repository;
    }

    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person>
       personStream) {
        return
           this.repository.save(personStream).then();
    }

    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }

    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```

`PersonRepository` berme jako implementaci nějaké databáze, jenž se stará o ukládání a hledání osob.

Anotace `@RestController` zjednodušeně říká, aby *singleton* instance této třídy poslouchala HTTP požadavky a podle typu zavolala příslušnou metodu.

Požadavek typu `POST` na adresu `http://example.com/person` zavolá metodu `create()`. Webflux se postará o to, aby tělo dotazu bylo převedeno do typu `Publisher<Person>`. `Publisher` může být jak `Mono` tak `Flux`. `PersonRepository` již pracuje s těmito reaktivními typy. Zároveň se klientovi pošle zpět odpověď, kterou je návratová hodnota metody `create()`.

Požadavek typu `GET` na adrese `http://example.com/person` vrátí v odpovědi seznam všech osob z databáze. Klient tedy obdrží typ `Flux<Person>` se kterým může následně operovat.

Požadavek typu `GET` na adrese `http://example.com/person/{id}`, kde `{id}` je libovolný textový řetězec, vrátí osobu s daným ID z databáze. Návratový typ je `Mono<Person>`.

2.4 Porovnání knihoven

Hlavním cílem této kapitoly bylo seznámení s Jenkins CI a jeho architekturou, se Spring Integration, s reaktivním programováním a následné vyhodnocení jejich využití.

Jenkins je aplikace zabývající se jinou oblastí něž je datová kvalita. Nicméně jeho architektura a způsob zpracování úloh popsaný v kapitole 2.1 je vyvíjenému nástroji podobný. Jelikož je *open source*, stálo za to zjistit, jestli by Jenkins, nebo aspoň nějaká jeho část, nešly využít v nástroji pro měření datové kvality.

Po prohlédnutí zdrojového kódu[27] a domluvě s vývojáři z firmy Simplity jsme usoudili, že by nebylo dobré Jenkins využít. Protože je vyvíjen od roku 2005, kdy se ještě jmenoval Hudson, není jeho architektura nejnovější. Zároveň je to projekt velmi rozsáhlý a vyjmout z něj jen tu malou a potřebnou část, by bylo velmi obtížné a pravděpodobně neefektivní řešení.

Následně jsem potřeboval doporučit jednu z knihoven pro integraci vyvíjeného nástroje s jinými aplikacemi. K tomu lze využít jak Spring Integration tak reaktivní programování ve formě Spring Webflux.

Spring Integration je již starší framework[13] avšak je stále vylepšován a vychází jeho nové verze, zatímco Spring Webflux ještě zcela oficiálně nevyšel a bude až součástí Spring Framework 5.0, který je plánován na letošní rok 2017. Lze ho ale již vyzkoušet – stáhnout poslední verzi od vývojářů.

Po prostudování a vyzkoušení obou těchto knihoven se přikláním k využití Spring Webflux a to z důvodů, že integrace by měla probíhat pomocí REST rozhraní, které zvládají obě knihovny. Ve Spring Webflux však vidím budoucnost díky trendu reaktivního programování a jeho jednoduchého použití. Zároveň se stará o serializaci a deserializaci přenášených objektů, stačí mít

jen vhodný doménový model. Velkou výhodou je automatický překlad obsahu požadavků i odpovědí do reaktivních typů *Flux* či *Mono*.

Spring Integration bych doporučil, pokud by integrace měla probíhat skrz jiné rozhraní než je REST nebo WebSocket. Jeho nastavení je o něco složitější, kvůli nutnosti psaní XML konfigurace.

Kapitola 3

Zpracování a spouštění úloh

Následující kapitola se zabývá grafovými algoritmy, které jsou potřeba ke zpracování a spouštění úloh. Nejprve se podíváme na to, jak jednotlivé úlohy reprezentovat. Poté si popíšeme různé druhy spouštění úloh, ať již paralelně nebo sekvenčně.

3.1 Zpracování grafu

Jak jsem již zmínil v kapitole 1, jednotlivé úlohy mohou tvořit orientovaný acyklický graf, kde úlohy jsou uzly grafu a závislosti mezi nimi tvoří hrany grafu. Existuje-li hrana vedoucí z uzlu u do uzlu v, pak úloha u potřebuje ke svému výpočtu výsledek z úlohy v viz obrázek 1.2. Pro budoucí účely však bude lepší mít graf opačně orientovaný, definovaný dle [5]:

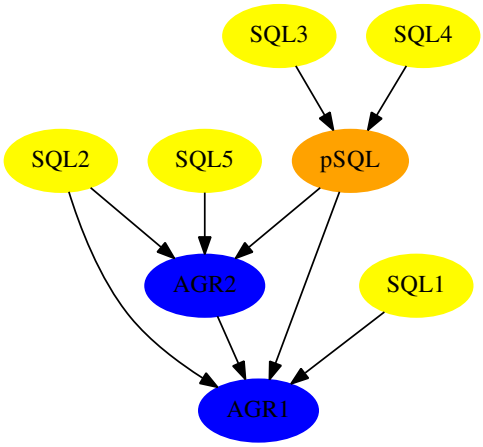
Definice: *„Orientovaný graf $G_2 = \langle H, U, \sigma_2 \rangle$ nazýváme **opačně orientovaným** vzhledem k orientovanému grafu $G_1 = \langle H, U, \sigma_1 \rangle$ (zapisujeme $G_2 = G_1^-$), pokud pro incidence σ_1 a σ_2 platí vztah*

$$\sigma_1(h) = (u, v) \Leftrightarrow \sigma_2(h) = (v, u) \quad \textit{pro každou hranu } h \in H. \text{“} \tag{3.1}$$

Opačně orientovaný graf úlohy AGR1 vidíme na obrázku 3.1. Existuje-li v něm hrana vedoucí z uzlu u do uzlu v, pak úloha u musí být vypočtena dříve než úloha v.

Další problém, jenž by měl nástroj řešit, je správná reprezentace grafu úloh. Nástroj přijímá úlohy, které vždy rekurzivně obsahují pole úloh potřebných k výpočtu. Tato reprezentace se však nehodí pro paralelní spouštění úloh. Podle [5] existují dva druhy reprezentace: maticová a spojová.

Dále se budu zabývat jen spojovou reprezentací, neboť je standardním způsobem vyjádření graf. Oproti maticové formě má navíc lepší asymptotickou paměťovou složitost a pro většinu řešených úloh umožňuje dosáhnout lepší časové složitosti.[5]



Obrázek 3.1: Opačně orientovaný graf úlohy AGR1 z obrázku 1.2

Spojová reprezentace orientovaného grafu $G = \langle H, U, \sigma \rangle$ je tvořena polem Adj obsahujícím $|U|$ ukazatelů na seznamy sousedů jednotlivých uzlů z množiny U. Pro každý uzel $u \in U$ obsahuje seznam $Adj[u]$ jeden záznam pro každou orientovanou hranu $h = (u, v) \in H$ s koncovým uzlem ve v.[5]

3.2 Spouštění úloh

Nyní se podíváme, jakými způsoby může náš nástroj úlohy spouštět, ať již sekvenčně, či paralelně. Důležité je, aby se při spouštění úloh dodrželo správné pořadí. Tedy aby každá úloha měla před svým výpočtem spočítány úlohy, na kterých její výpočet závisí.

Na začátku si nejprve definujeme termíny podle [5] používané v následujících podkapitolách.

Definice: Existuje-li v grafu hrana z uzlu u do uzlu v, poté uzel u nazýváme **předchůdcem** uzlu v nebo obdobně uzel v **následníkem** uzlu u.

Definice: „*Pro libovolný uzel u orientovaného grafu G nazýváme **výstupním stupněm uzlu** u (značíme $\delta^+_G(u)$) počet hran, které mají uzel u za svůj počáteční uzel, **vstupním stupněm uzlu** u (značíme $\delta^-_G(u)$) počet hran, které mají uzel u za svůj koncový uzel.*

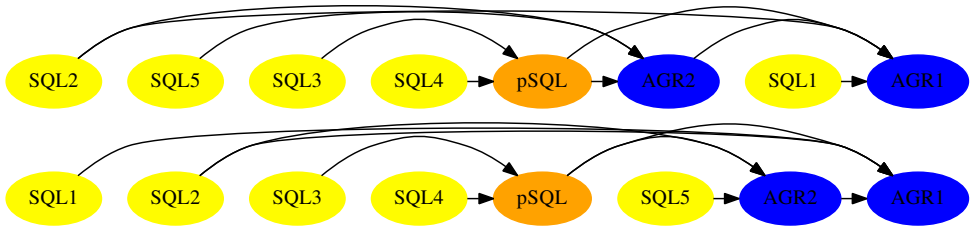
Stupně uzlů souvisejí s množinami jeho následovníků $\Gamma(u)$ a předchůdců $\Gamma^{-1}(u)$. V prostém orientovaném grafu platí:

$$\delta^+(u) = |\Gamma(u)|, \quad \delta^-(u) = |\Gamma^{-1}(u)| \tag{3.2}$$

*Nemá-li uzel u v orientovaném grafu G žádné předchůdce (tzn. $\delta^-(u) = 0$), říkáme, že uzel u je **kořen** grafu G.*“

3.2.1 Sekvenční

Pro sekvenční spouštění úloh nám stačí využít topologického uspořádání uzlů. **Topologické uspořádání** je úplné uspořádání uzlů, v němž je pro každou hranu (u, v) uzel u před uzlem v. Takové uspořádání existuje jen pro acyklické grafy. Topologické uspořádání grafu není vždy jednoznačně určeno.[5] Příklady topologických uspořádání na grafu úlohy AGR1 z obrázku 3.1 vidíte na obrázku 3.2.



Obrázek 3.2: Dvě různá topologická uspořádání úlohy AGR1

První jednoduchý algoritmus na zjištění topologického uspořádání grafu funguje za použití algoritmu prohledávání do hloubky (angl. *depth-first search*, zkráceně DFS).

DFS je založeno na rozdělování uzlu do tří skupin: nové, otevřené a uzavřené. Otevřeným uzlem je po svém prvním objevení, uzavřeným se stává po kompletním prozkoumání všech jeho následníků. Ke každému uzlu u se přidělují dvě značky uchovávané v poli $d[u]$ a $f[u]$, kde $d[u]$ odpovídá okamžiku objevení uzlu a $f[u]$ jeho uzavření.

Přikládám pseudokód algoritmu DFS (algoritmus 1 a 2) pro orientovaný graf $G = \langle H, U \rangle$ za předpokladu spojové reprezentace grafu prostřednictvím seznamů následníků.[5]

Poté se topologické uspořádání najde pomocí algoritmu 3, který má asymptotickou časovou složitost $O(|U| + |H|)$.[5]

Druhý algoritmus na nalezení topologického uspořádání (algoritmus 4) je postaven na postupném odebírání kořenů z grafu. V algoritmu se používá pomocné pole δ, které obsahuje vstupní stupeň uzlů v podgrafech získávaných postupným vypouštěním kořenů. Kořeny se ukládají do množiny M a do zásobníku. Uzly z množiny M se dále používají k upravě jejich následníků. Na zásobníku po dokončení algoritmu najdeme uzly v topologickém uspořádání. Algoritmus má stejnou asymptotickou časovou složitost jako algoritmus založený na DFS, tedy $O(|U| + |H|)$.[5]

Algoritmus 1 Prohledávání do hloubky

```
1:  DFS(G)
2:      for každý uzel u ∈ U do
3:          stav[u] := FRESH
4:          p[u] := NULL
5:      end for
6:      i := 0
7:      for každý uzel u ∈ U do
8:          if stav[u] = FRESH then
9:              DFS-Projdi(u)
10:         end if
11:     end for
12: end
```

Algoritmus 2 Prohledávání do hloubky (pokračování)

```
13: DFS-Projdi(u)
14:     stav[u] := OPEN
15:     i := i + 1
16:     d[u] := i
17:     for každý uzel v ∈ Adj[u] do
18:         if stav[v] = FRESH then
19:             p[v] := u
20:             DFS-Projdi(v)
21:         end if
22:     end for
23:     stav[u] := CLOSED
24:     i := i + 1
25:     f[u] := i
26: end
```

Algoritmus 3 Topologické uspořádání uzlů grafu pomocí DFS

```
1:  TOP-SORT1(G)
2:      vytvoř prázdný seznam uzlů S
3:      pomocí DFS(G) počítej okamžiky uzavření f[v] všech uzlů grafu G
4:      v okamžiku uzavírání ulož každý uzel v na začátek seznamu uzlů S
5:      seznam S obsahuje uzly seřazené podle definice topologického uspořá-
    dání
6:  end
```

Algoritmus 4 Topologické uspořádání uzlů grafu pomocí vypouštění kořenů

```
1:  TOP-SORT2(G)
2:      for každý uzel u ∈ U do
3:          δ[u] := 0
4:      end for
5:      for každý uzel u ∈ U do
6:          for každý uzel v ∈ Adj[u] do
7:              δ[v] := δ[v] + 1
8:          end for
9:      end for
10:     M := ∅; INIT_STACK
11:     for každý uzel u ∈ U do
12:         if δ[u] = 0 then
13:             M := M ∪ {u}
14:             PUSH(u)
15:         end if
16:     end for
17:     while M ≠ ∅ do
18:         w := libovolný uzel z M
19:         M := M − {v}
20:         for každý uzel w ∈ Adj[v] do
21:             δ[w] := δ[w] − 1
22:             if δ[w] = 0 then
23:                 M := M ∪ {w}
24:                 PUSH(w)
25:             end if
26:         end for
27:     end while
28: end
```

Po získání topologického uspořádání stačí jen úlohy jednotlivě za sebou spouštět.

3.2.2 Paralelní

Pro paralelní spouštění úloh mohou být použity následující tři algoritmy:

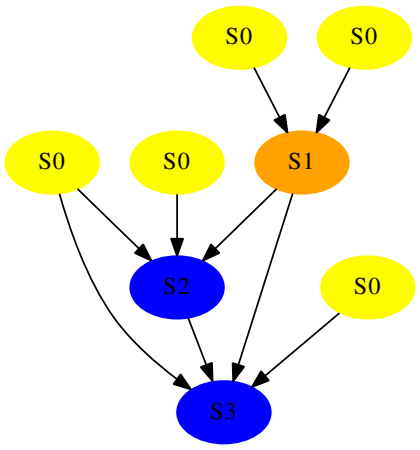
- paralelní spouštění po vrstvách;
- paralelní procházení grafu;
- paralelní spouštění topologického uspořádání.

Paralelní spouštění po vrstvách je založeno na spouštění nezávislých množin úloh, kde každá množina neobsahuje závislosti mezi jednotlivými uzly.

Nechť máme graf $G = \langle H, U \rangle$, poté lze uzly grafu rozdělit do množin S_i pomocí vztahu:

$$\begin{aligned} S_0 &= \text{Kořeny grafu } G, \\ S_{i+1} &= \{v \in U \mid \forall u \in U, (u,v) \in H \Rightarrow u \in \bigcup_{k=0}^{i} S_k\}. \end{aligned} \tag{3.3}$$

Na obrázku 3.3 vidíte rozdělení grafu úlohy AGR1 do nezávislých množin S_i.



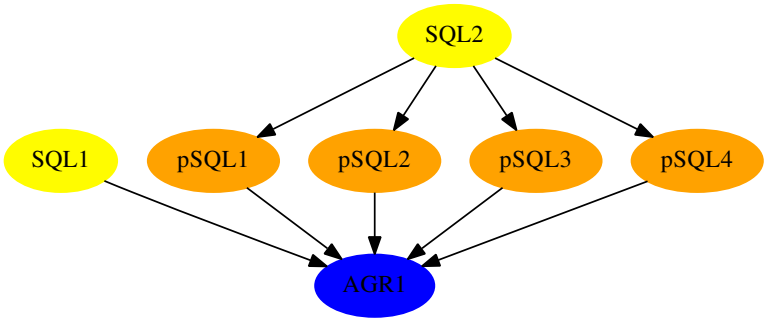
Obrázek 3.3: Graf úlohy AGR1 3.1 s nezávislými množinami S_i

Spuštění následně proběhne podle algoritmu 5:

Algoritmus 5 Paralelní spoutění úloh po vrstvách

```
1: EXECUTE-1(k, S[])
2:     i := 0
3:     for i < k do
4:         for každý uzel u ∈ S[k]  do in parallel
5:             výpočet u
6:         end for
7:     end for
8: end
```

Spouštění po vrstvách však není optimální, jelikož nemusí využívat plný potenciál paralelního výpočtu, tedy zapojení všech jader procesoru. Viz obrázek 3.4, jestliže úloha SQL1 bude počítána mnohem déle než úloha SQL2, poté bude zbytečně blokován výpočet úloh pSQL1, pSQL2, pSQL3 a pSQL4.



Obrázek 3.4: Neoptimální graf úlohy pro paralelní spouštění po vrstvách

Dalším způsobem jak úlohy paralelně spouštět je **paralelní procházení grafu**. Pro každý kořen u grafu $G = \langle H, U \rangle$ se spustí vlákno s jeho výpočtem. Jakmile jeho výpočet skončí, pro každého následovníka uzlu u se vytvoří nová vlákna s jejich výpočtem. Synchronizace bude provedena ve výpočtu úlohy – pokud byl výpočet zavolán méněkrát, než je jeho vstupní stupeň $\delta(u)$, tak se výpočet přeruší, v opačném případě pokračuje, viz algoritmus 6:

Algoritmus 6 Paralelní procházení grafu

```
1:  EXECUTE-2(G)
2:      for každý uzel u ∈ U do
3:          u.count := 0
4:      end for
5:      for každý kořen u ∈ U do
6:          EXECUTE-PARALLEL(u)
7:      end for
8:  end
1:  EXECUTE-PARALLEL(u)
2:      atomic {                                  ▷ Začátek atomické operace
3:          u.count := u.count + 1
4:          if u.count < δ(u) then
5:              end
6:          end if
7:      }                                         ▷ Konec atomické operace
8:      výpočet u
9:      for každý uzel v ∈ Adj[u] do in parallel
10:         EXECUTE-PARALLEL(v)
11:     end for
12: end
```

Problémem u paralelního procházení grafu může být vysoká režie při vytváření mnoha nových vláken.

Poslední možností, jak spouštět úlohy paralelně, a kterou implementuji v nástroji, bude **paralelní spuštění topologického uspořádání** za pomoci Java třídy `java.util.concurrent.CountDownLatch`. Tato třída se používá k synchronizaci několika vláken. Při vytváření instance třídy `CountDownLatch` se nastaví hodnota N. Po zavolání metody `await()` je aktuální vlákno blokováno, dokud není hodnota N snížena voláním metody `countDown()` až na nulu.

K použití tohoto algoritmu, bude každá úloha obsahovat svůj `CountDownLatch` s hodnotou N nastavenou na počtu vstupního stupně uzlu $\delta(u)$ a pole obsahující `CountDownLatch` všech následníku. Výpočet úloh bude blokován funkcí `await()`. Po uvolnění vlákna a dopočítání úlohy se na poli následníků zavolá `countDown()`. Úlohy se budou zpracovávat paralelně, ale jednotlivá vlákna musí být vytvořena v pořadí topologického uspořádání, aby nenastal *deadlock*.

Algoritmus 7 Paralelní spouštění topologického uspořádání

```
1: EXECUTE-3(G)
2:     topsort := TOPSORT(G)
3:     for každý uzel u ∈ topsort do in parallel
4:         DO-EXECUTE(u)
5:     end for
6: end
1: DO-EXECUTE(u)
2:     u.countDownLatch.await()
3:     výpočet u
4:     for každý uzel v ∈ Adj[u] do
5:         v.countDownLatch.countDown()
6:     end for
7: end
```

Kapitola 4

Realizace

V úvodu této kapitoly je představena zvolená technologie. Dále je zde uvedena struktura projektu a popis jednotlivých komponent. Více prostoru je věnováno pro základní rozhraní úloh a poté pro implementace jednotlivých druhů spouštění z předchozí kapitoly 3.2. Následně je představen nástroj jako serverová aplikace. Závěr této kapitoly se věnuje výkonnostním testům spouštění úloh a jejich výsledkům.

4.1 Použitá technologie

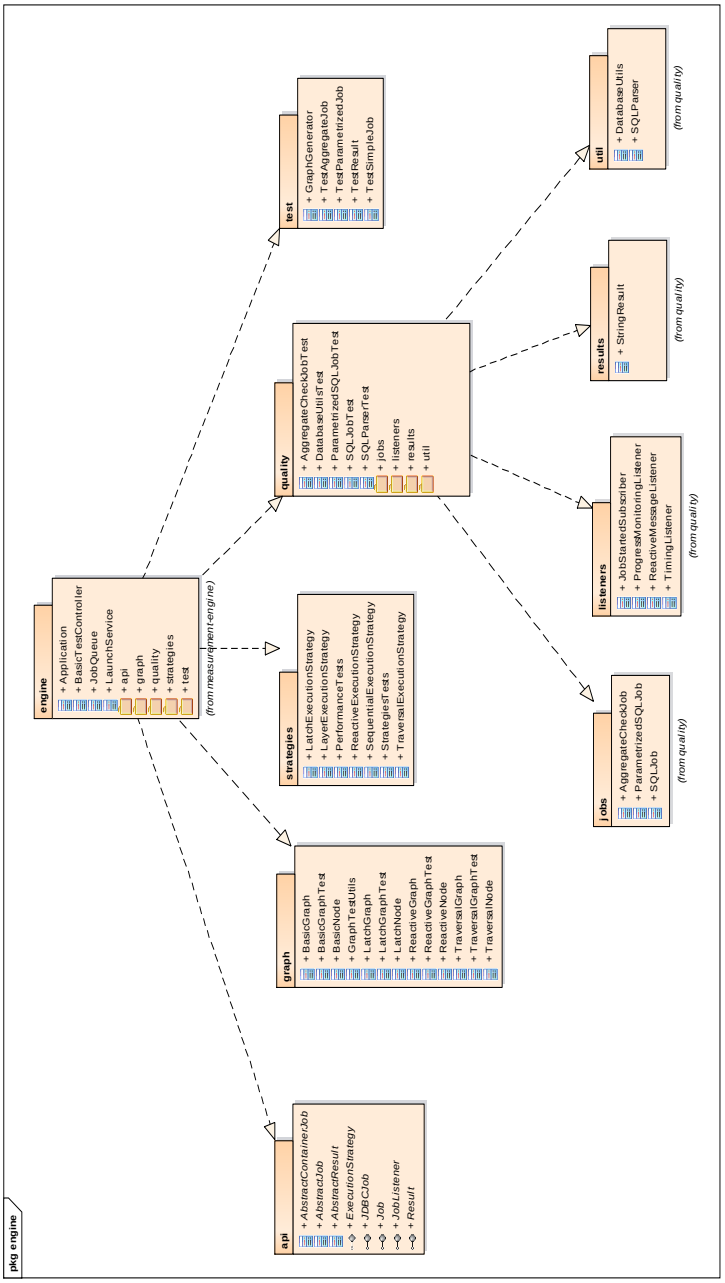
Pro realizaci bylo potřeba použití programovacího jazyku Java, jelikož je standardem pro tvorbu podnikových aplikací a firma Simplity ho používá ve všech svých aplikacích.

Spring Framework byl použit díky své implementaci *Inversion of Control* (též *dependency injection*), což je proces, v němž si jednotlivé objekty určí závislosti na jiných objektech. Tyto závislosti pak zaručí dosazení správného objektu při jeho vytvoření.[17] Spring Boot nám umožnil vytvořit z nástroje samostatnou serverovou aplikaci, která je spuštěna na vnořeném serveru – jakým může být např. Tomcat – bez nutnosti nasazení *Web Application Archive* (WAR) souboru. Spring Integration a Spring Reactive byly zase použity k analýze v kapitole 2.

K testování jednotlivých částí kódu (tzv. *unit testing*) byla použita knihovna jUnit. K vyzkoušení nástroje nad skutečnými daty byla nainstalována databáze Oracle Database 11g Release 2. Některé testy aplikace využívají ukázkové databáze získané ihned po instalaci.

K sestavení aplikace a k provázání jednotlivých knihoven byl využit Apache Maven. Maven je tzv. *build tool*, tedy nástroj pro usnadnění kompilování a sestavení velkých aplikací. Základem je soubor `pom.xml`, ve kterém se definují základní údaje o projektu, jako jsou jméno nebo verze. Dále se v tomto souboru určí závislosti na ostatních knihovnách, které Maven tyto knihovny poté stáhne ze svého repositáře a importuje do projektu.

Projekt jsem vyvíjel v programu IntelliJ IDEA od firmy JetBrains, jelikož to je výborné vývojové prostředí pro vyvíjení projektů v Javě a obsahuje podporu databází, Spring Framework či verzování pomocí systému správy verzí GIT, který byl použit k zálohování projektu.



Obrázek 4.1: Hierarchická struktura projektu

4.2 Struktura projektu

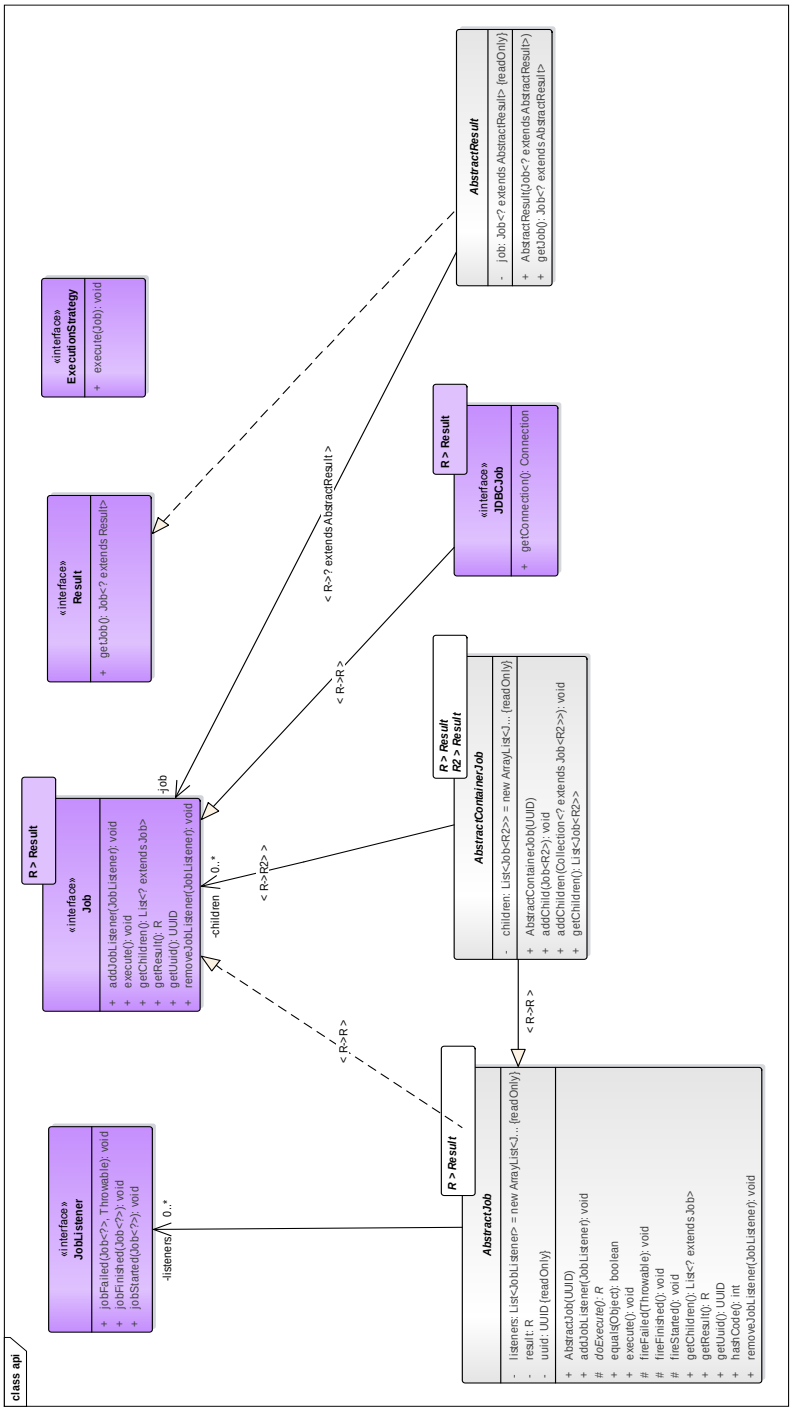
Vzhledem k velikosti projektu, která v budoucnu ještě poroste, je za účelem zvýšení přehlednosti vhodné rozdělit jednotlivé třídy do několika balíčků (tzv. *packages*). Výsledná struktura je zobrazena na diagramu 4.1.

Projekt je na nejvyšší úrovni tvořen balíčkem `engine`, který slouží jako jmenný prostor pro třídy zajišťující chod serverové aplikace a pro těchto dalších pět vnořených balíčků:

- `api` – *základní rozhraní úloh.* Obsahuje základní rozhraní a abstraktní třídy pro práci s úlohami a jejich výsledky.
- `graph` – *reprezentace grafů.* Implementace grafu úloh v závislosti na typu spouštění.
- `strategies` – *spouštění úloh.* Jednotlivé strategie jsou odpovědny za sestavení grafu úloh a jeho následné spuštění.
- `quality` – *implementace úloh.* Implementace základních typů úloh a výsledků.
- `test` – *testovací implementace úloh.* Implementace testovacích úloh a generování grafů pro potřeby výkonnostních testů, bez nutnosti připojení k databázi.

4.2.1 Základní rozhraní úloh

Základní rozhraní pro práci s úlohami a jejich výsledky je uloženo v balíčku `api`. Jedná se o rozhraní (angl. *interfaces*) a abstraktní třídy využívající šablony (generické třídy a funkce). Na obrázku 4.2 lze vidět diagram zachycující vzájemné vztahy mezi třídami balíčku `api`.



Obrázek 4.2: Diagram tříd reprezentujících základní rozhraní úloh

4.2.1.1 Job

`Job` je základní generické rozhraní pro úlohu. Každá úloha musí implementovat toto rozhraní. Generickým typem je jakákoliv třída, která implementuje třídu `Result`. Rozhraní definuje tyto metody:

- `getUuid()` – Vrací hodnotu typu `java.util.UUID`, která bude úlohu jednoznačně identifikovat.
- `getResult()` – Vrací výsledek úlohy, který je generickým typem.
- `getChildren()` – Vrací list úloh, tedy tříd implementujících `Job`, které jsou potřeba k výpočtu této úlohy.
- `execute()` – Spouští výpočet úlohy.
- `addJobListener(JobListener listener)` – Přidává k úloze objekt typu `JobListener`.
- `removeJobListener(JobListener listener)` – Odebírá z úlohy objekt typu `JobListener`.

4.2.1.2 JDBCJob

`JDBCJob` je generické rozhraní rozšiřující `Job` o metodu `getConnection()`. Tato metoda vrací instanci třídy `java.sql.Connection`, která reprezentuje spojení s databází a na níž je možné spouštět SQL dotazy do databáze.

4.2.1.3 JobListener

`JobListener` je rozhraní, které reaguje na zpracování úloh. Definuje tři metody, z nichž je každá spuštěna po jiné dokončené části úlohy:

- `jobStarted(Job<?> job)` – Metoda volána vždy, když je zahájen výpočet úlohy. Parametrem je spuštěná úloha.
- `jobFinished(Job<?> job)` – Metoda volána vždy, když je dokončen výpočet úlohy. Parametrem je opět dokončená úloha.
- `jobFailed(Job<?> job, Throwable t)` – Metoda volána vždy, když dojde během výpočtu úlohy k chybě. Parametrem je úloha, ve které došlo k chybě, a vyhozená výjimka.

4.2.1.4 Result

`Result` je základní rozhraní představující výsledek úlohy. Toto rozhraní definuje jen jednu metodu, a to `getJob()`, která vrací úlohu, pro niž je výsledkem.

4.2.1.5 ExecutionStrategy

`ExecutionStrategy` je rozhraní pro strategie, jimiž se budou spouštět úlohy. Rozhraní definuje metodu `execute(Job job)`, která spustí zadanou úlohu.

4.2.1.6 AbstractResult

`AbstractResult` je abstraktní třída implementující `Result`. Tato třída obsahuje jeden atribut třídy `Job`, který je nastaven parametrem konstruktoru a je vracen v metodě `getJob()`.

4.2.1.7 AbstractJob

`AbstractJob` je abstraktní třída implementující `Job`. Tato třída obsahuje tři privátní atributy:

- `uuid` – Instance třídy `java.util.UUID`, která jednoznačně identifikuje úlohu.
- `result` – Výsledek úlohy.
- `listeners` – Pole obsahující instance `JobListener`.

Tyto atributy jsou použity v metodách z rozhraní `Job`. `AbstractJob` neobsahuje žádné úlohy potřebné k výpočtu, proto metoda `getChildren()` vrací vždy prázdné pole úloh.

Dále třída obsahuje tři *protected* metody `fireStarted()`, `fireFinished()` a `fireFailed(Throwable t)`, jejichž cílem je zavolání metod `jobStarted()`, `jobFinished()` a `jobFailed()` na všech objektech v poli `listeners`.

Tyto tři metody jsou volány z funkce `execute()`:

```
public void execute() {
    fireStarted();
    try {
        result = doExecute();
        fireFinished();
    } catch (Throwable t) {
        fireFailed(t);
    }
}
```

Abstraktní metoda `doExecute()` bude sloužit k přesné implementaci výpočtu úlohy, tedy např. dotazu do databáze a následné zpracování výsledku. Vztah mezi `execute()` a `doExecute()` odpovídá návrhovému vzoru *Template*.

Metody `equals(Object o)` a `hashCode` využívají jen hodnoty z atributu `uuid`.

4.2.1.8 AbstractJobContainer

`AbstractJobContainer` je potomkem třídy `AbstractJob`. Jak z názvu napovídá, tato abstraktní třída je vzorem pro úlohy, které potřebují pro svůj výpočet úlohy jiné. Třída má privátní atribut `children`, jenž je list typu `java.util.ArrayList` a obsahuje instance `Job`.

`AbstractJobContainer` již potřebuje dva generické typy. Kromě generického typu svého výsledku `R` používá i generický typ výsledku svých „dětí“ `R2`.

Přibyly zde dvě veřejné metody `addChild(Job<R2> child)` a `addChildren(Collection<? extends Job<R2>> children)`, jenž přidávají úlohu nebo kolekci úloh do atributu `children`.

4.2.2 Reprezentace grafů

V balíčku `graph` jsou uloženy různé implementace grafů a uzlů grafu v závislosti na způsobu spouštění. Uzly grafu obalují úlohy a přidávají k nim další informace potřebné k jejich výpočtu. Jednotlivé implementace si jsou velmi podobné, ovšem použití dědičnosti je zde zbytečné.

4.2.2.1 Node

`Node` je abstraktní implementací uzlu grafu. Obsahuje tři *protected* atributy:

- `uuid` – `java.util.UUID` obalované úlohy.
- `job` – Je obalovaná úloha, tedy objekt typu `Job`.
- `inDegree` – Číslo typu `int`, jenž udává vstupní stupeň uzlu.

`Node` dále obsahuje tzv. *gettery*, což jsou veřejné funkce, které jen vrací příslušné atributy, a *setter* pro `inDegree`, který ho umožňuje nastavovat. Upraveny jsou také metody `equals()` a `hashCode()`, které pracují na základě `uuid`.

4.2.2.2 BasicNode a BasicGraph

`BasicNode` je potomkem `Node`, který implementuje rozhraní `Callable`. Toto rozhraní je podobné rozhraní `Runnable`, tedy říká o objektu, že může být spuštěn v jiném vlákně. V tomto vlákně se spustí metoda `call()`. Oproti `Runnable` vrací `Callable` výsledek a může vyhodit výjimku.[28]

`BasicNode` navíc obsahuje privátní atribut `depth` a jeho *getter* a *setter*. `Depth` je číslo typu `int`, které značí do jaké nezávislé množiny z kapitoly 3.2.2 spadá. Dále jsou v této třídě metody `execute()` a `call()`, jenž jsou téměř totožné, jen `execute()` se používá při sekvenčním spouštění, zatímco `call()` při paralelním. Stačilo by mít jen jednu metodu `call()`, ovšem kvůli přehlednosti jsem ponechal obě dvě.

`BasicGraph` je graf tvořen `BasicNode` uzly. Spojová reprezentace grafu je uložena v privátním atributu `map`, jenž je typu `HashMap<BasicNode, List<BasicNode>>`. Je to tedy mapa, kde klíče jsou jednotlivé uzly a hodnotou je vždy list následníků daného uzlu. Tato mapa se sestaví pomocí rekurzivní metody `buildReversedGraph(Job<?> currentJob)`, jenž je volána v konstruktoru.

`BasicGraph` dále obsahuje metodu pro získání topologického uspořádání grafu `topSort()` podle algoritmu 4. Na získání topologického uspořádání tedy používám algoritmus, který je založen na postupném odebírání kořenů. Výhodou oproti algoritmu založeném na DFS je i jeho využití v následující metodě pro získání nezávislých množin. Metoda `getLayers()` vrací nezávislé množiny uzlů, ve kterých mohou být úlohy spuštěny paralelně, viz 3.2.2. Využívá se k tomu atributu `depth`, který se nastavuje během odstraňování kořenů v `topSort()` a značí, v jaké nezávislé množině leží.

4.2.2.3 TraversalNode a TraversalGraph

`TraversalNode` společně s `TraversalGraph` umožňují paralelní procházení grafu viz algoritmus 6.

`TraversalNode` je taktéž potomkem `Node`, který však implementuje rozhraní `Runnable`. `Runnable` říká, že objekt může být spuštěn v jiném vlákně, kde se zavolá jeho metoda `run()`. Tato třída rozšiřuje `Node` o atribut `finishedPredecessors`, jenž udává počet již dokončených následníků uzlu, dále o `successors`, což je pole následovníků uzlu, a o `executorService`.

Rozhraní `java.util.concurrent.ExecutorService` je v programovacím jazyku Java od verze 1.5 a umožňuje spuštění asynchronních úloh na pozadí. Aby bylo možné nějaký objekt spustit pomocí metody `submit()`, musí implementovat rozhraní `Callable` či `Runnable`.[28]

Ukázka metody `run()`:

```
public void run() {
    synchronized (this) {
        if (++finishedPredecessors < inDegree) {
            return;
        }
    }
    job.execute();
    successors.forEach(executorService::submit);
}
```

`TraversalGraph` je opět graf tvořen `TraversalNode` uzly. Metoda `getRoots()` vrací všechny kořeny grafu.

4.2.2.4 LatchNode a LatchGraph

`LatchNode` společně s `LatchGraph` umožňují paralelní spouštění topologického uspořádání, viz algoritmus 7.

`LatchNode` je znovu potomkem `Node` a implementuje rozhraní `Runnable`. Navíc obsahuje dva atributy:

- `latch` – Objekt typu `java.util.concurrent.CountDownLatch`, který blokuje výpočet úlohy, dokud není jeho hodnota nastavena na 0.
- `decrements` – Pole referencí na instance `CountDownLatch` všech následníků uzlu, na kterých se volá metoda `countDown()` po dokončení výpočtu.

Ukázka implementace metody `run()`:

```
public void run() {
    try {
        latch.await();
        job.execute();
        decrements.forEach(CountDownLatch::countDown);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

`LatchGraph` je velmi podobný `BasicGraph` – zde se jen při vytváření grafu musí vytvořit `CountDownLatch` pro všechny uzly a správně jej napojit na následníky a předchůdce.

4.2.2.5 ReactiveNode a ReactiveGraph

`ReactiveNode` a `ReactiveGraph` jsou skoro totožné s `TraversalNode` a s `TraversalGraph`. Ovšem zde je použito neblokující reaktivní API Reactor (viz kapitola 2.3.2) místo `ExecutorService`. `ReactiveNode` tudíž neimplementuje rozhraní `Runnable`, má jen metodu `execute()`:

```
public void execute() {
    synchronized (this) {
        if (++finishedPredecessors < inDegree) {
            return;
        }
    }
    job.execute();
    Flux.fromIterable(successors)
        .flatMap(value ->
            Mono.just(value)
                .subscribeOn(Schedulers.elastic())
                .doOnNext(ReactiveNode::execute),
            64)
        .subscribe();
}
```

Reactor je zde použit k vytvoření `Flux`, který obsahuje všechny následníky uzlu. Operátor `flatMap()` rozdělí hodnoty, které `Flux` obsahuje, na samostatně fungující `Mono`. Toto `Mono` se paralelně zpracovává díky příkazu `Schedulers.elastic()`. Argument 64 v operátoru `flatMap()` udává počet paralelně pracujících `Mono`.[22]

4.2.3 Spouštění úloh

Spouštění úloh dle kapitoly 3.2 je uloženo v balíčku `strategies`. Tento balíček obsahuje komponenty implementující rozhraní `ExecutionStrategy`, které spouští zadané úlohy. Jednotlivé strategie jsou zodpovědné za vytvoření správného grafu a následné vypočítání úloh:

- `SequentialExecutionStrategy` – Vytvoří pro zadanou úlohu `BasicGraph`, z něhož získáme pomocí metody `topSort()` topologické uspořádání grafu. Úlohy jsou poté spuštěny sekvenčně v pořadí topologického uspořádání.
- `LayerExecutionStrategy` – Vytvoří pro zadanou úlohu `BasicGraph`. Nezávislé množiny, které jsou poté za sebou paralelně spouštěny dle algoritmu 5, získáme voláním metody `getLayers()`.

- `TraversalExecutionStrategy` – Vytvoří pro zadanou úlohu `TraversalGraph`, kde nám metoda `getRoots()` vrátí kořeny grafu, ve kterých začne následné procházení (viz algoritmus 6).
- `ReactiveExecutionStrategy` – Podobné `TraversalExecutionStrategy`. Je však použit `ReactiveGraph` a spouštění obstarává reaktivní API Reactor z kapitoly 2.3.2.
- `LatchExecutionStrategy` – Vytvoří pro zadanou úlohu `LatchGraph`. Poté z metody `topSort()` získáme topologické uspořádání, které je následně paralelně spuštěno za synchronizace pomocí `CountDownLatch)` (viz algoritmus 7).

4.2.4 Implementace úloh

Základní implementace úloh pro měření datové kvality se nachází v balíčku `quality`. Člení se na další čtyři balíčky:

- `jobs` – Implementace `Job`.
- `listeners` – Implementace `JobListener`
- `results` – Implementace `Result`.
- `util` – Pomocné třídy, které se využívají např. při výpočtech úloh.

V budoucnu se plánuje rozšíření tohoto balíčku v závislosti na dalších typech úloh.

4.2.4.1 Jobs

Balíček `jobs` obsahuje tři implementace úloh dle požadavků v kapitole 1.4.

Prvním typem úlohy je `SQLJob`, tedy SQL dotaz nad databází. `SQLJob` dědí z `AbstractJob` a implementuje `JDBCJob`. V metodě `doExecute()` se jen nad databází zavolá daný SQL dotaz a hodnota se následně uloží do výsledku `StringResult`.

Druhým typem úlohy je `ParametrizedSQLJob`, tedy parametrizovaný SQL dotaz. `ParametrizedSQLJob` rozšiřuje `AbstractJobContainer`, jelikož může obsahovat nějaké úlohy, které potřebuje ke svému výpočtu. Také implementuje `JDBCJob`, protože komunikuje s databází. Metoda `doExecute()` nejdříve správně doplní do SQL dotazu výsledky ze závislých úloh a poté SQL dotaz provede. Jeho výsledek se opět uloží jako typ `StringResult`.

Posledním typem úlohy je `AggregateCheckJob`, nebo-li agregovaný výpočet z výsledku jiných úloh. `AggregateCheckJob` tedy musí dědit z `AbstractJobContainer`. Obsahuje pole čísel typu `Long`, které je použito k výpočtu váženého průměru z výsledků zadaných úloh. Jako výsledek je znovu použit `StringResult`.

4.2.4.2 Listeners

Implementace rozhraní `JobListener` patří do balíčku `listeners`. Nyní obsahuje jen ukázkové implementace, sloužící především k vypisování informací na výstup. V budoucnu se v nich však může odehrávat např. přenos výsledků do jiné aplikace.

Pokud úloha obsahuje `TimingListener`, tak po dokončení jejího výpočtu se na výstup vypíše doba, která byla potřeba k dopočítání úlohy. `ProgressMonitoringListener` zase vypisuje aktuální stav úloh, tedy kolik jich bylo spuštěno, kolik jich úspěšně skončilo a kolik jich skončilo s chybou. `ReactiveMessageListener` opět jen vypisuje informace na výstup, ovšem je to ukázka reaktivního API Reactor z kapitoly 2.3.2.

4.2.4.3 Results

V balíčku se nacházejí implementace rozhraní `Result`. Nyní však obsahuje jen jednu implementaci, a to `StringResult`, což je jednoduchý výsledek, který obsahuje jeden atribut typu `String`.

4.2.4.4 Util

Balíček `util` slouží jako jmenný prostor pro pomocné třídy, kterými jsou např. `DatabaseUtils` nebo `SQLParser`.

`DatabaseUtils` disponuje podpůrnými funkcemi při práci s databázemi. Nyní obsahuje jen jednu statickou funkci `getOneResult()`, která zkontroluje, zda daný SQL výsledek obsahuje jen jeden záznam s jedním sloupcem a pokud ano, tak ho vrátí.

`SQLParser` nahrazuje v textovém řetězci parametry typu {`index`}, kde `index` je přirozené číslo, za hodnoty z dodaného pole. V nástroji se využívá v `ParametrizedSQLJob`, v němž je potřeba parametry nahradit výsledky SQL dotazů.

4.2.5 Testování

Kromě *unit* testů, které testují funkčnost jednotlivých částí kódu, jsem se věnoval i výkonnostnímu testování v závislosti na druhu spouštění úloh. Třídy potřebné k výkonnostním testům jsou uloženy v balíčku `test`.

V tomto balíčku jsou tři implementace úloh, které spouští metodu `sleep()`:

```
private void sleep(long duration) {
    boolean sleep = true;
    long startTime = new Date().getTime();
    while (sleep) {
        if ((new Date().getTime()) - startTime >=
           duration) {
            sleep = false;
        }
    }
}
```

Ta slouží k napodobení spouštění SQL dotazu, tedy zablokuje vlákno na určitý čas v milisekundách dle parametru `duration`. Každá z implementací volá metodu sleep s jiným argumentem:

- `TestAggregateJob` – 5 milisekund;
- `TestParametrizedJob` – 500 milisekund;
- `TestSimpleJob` – 300 milisekund.

`TestResult` je naivní implementace výsledku úlohy, která ukládá hodnotu typu `GraphGenerator`. `GraphGenerator` slouží ke generování testovacích grafů. Metoda `generateJob()` vygeneruje graf s celkovým počtem uzlů $n^3 + n^2 + n + 1$, kde n je vstupní parametr metody. Přesněji se vygeneruje $n + 1$ úloh `TestAggregateJob`, n^2 úloh `TestParametrizedJob` a n^3 úloh `TestSimpleJob`.

Výkonnostní testy jsou ve třídě `engine.strategies.PerformanceTests` a lze s nimi testovat všechny strategie. Konstanta `NODES_NUMBER` určuje parametr N pro generování grafu pomocí `GraphGenerator`. Konstanta `THREAD_POOL` říká, kolik maximálně vláken lze vytvořit.

4.2.6 Serverová aplikace

Balíček `engine` obsahuje čtyři třídy, které se zabývají tím, aby nástroj fungoval jako serverová aplikace.

Nejdůležitější částí je třída `Application`, která v hlavní Java metodě `main()` spustí server a nasadí na něj naši aplikaci pomocí *frameworku* Spring Boot. Po doběhnutí této metody máme aplikaci spuštěnou na adrese `http://localhost:8085`.

`JobQueue` je *singleton*, tzn. že v aplikaci existuje jen jedna instance této třídy. Tato instance tvoří frontu úloh, které byly aplikaci poslány k výpočtu. Využívá k tomu `java.util.concurrent.BlockingQueue`, což je implementace fronty pro vícevláknové použití.[28]

`LaunchService` je taktéž *singleton*. Zároveň je potomkem třídy `Thread` a může tedy být její metoda `run()` spuštěna ve vlastním vlákně. K tomu dochází ihned po vytvoření instance za použití anotace `@PostConstruct`. `LaunchService` obsahuje v sobě `JobQueue` a také jednu z implementací `ExecutionStrategy`. V těle metody `run()` je poté nekonečný cyklus, který bere prvky z fronty a spouští je na zadané strategii.

`BasicTestController` je jen ukázkou toho, jak by mohla vypadat integrace pomocí Spring Webflux z kapitoly 2.3.3. Díky této třídě poslouchá naše aplikace HTTP požadavky, např. GET požadavek na adrese `http://localhost:8085/{number}`, kde `number` je jakékoliv číslo, spustí metodu `runTest`. Tato metoda jen vygeneruje graf pomocí `GraphGenerator` s parametrem `number`. Vygenerovaný graf je poté přidán do fronty na spuštění.

4.3 Výsledky výkonnostních testů

Nyní se podíváme na výsledky výkonnostních testů, jenž probíhaly na notebooku s procesorem Intel Core i5-4210U s dvěma jádry a čtyřmi vlákny, s 8 GB RAM DDR3, s Windows 10 a Java verzí 1.8. Grafy pro výkonnostní testy budou generovány pomocí `GraphGenerator` viz kapitola 4.2.5.

Nejdříve jsem potřeboval zjistit optimální počet vláken, který má být vytvořen. Výpočet úloh je většinu času tvořen dotazem do databáze, což je I/O operace. Při paralelním zpracování I/O operací je doporučeno využívat mnohem více jader, než jich má procesor k dispozici, i když se zvýší režie přepínání kontextu. V tabulkách 4.1 a 4.2 můžeme vidět závislost doby výpočtu úloh s danou strategií v závislosti na počtu vytvořených vláken. V tabulce 4.1 jsou zaznamenány výsledky z grafu, který byl vygenerován pomocí `GraphGenerator.generateJob()` s parametrem $N = 6$ (258 uzlů), v tabulce 4.2 s parametrem $N = 10$ (1111 uzlů). Výsledky v obou těchto tabulkách jsou v sekundách zaokrouhlených na desetiny a počítány byly jako průměr z 10 měřených pokusů. `ReactiveStrategy` zde není testována, jelikož Reactor si řídí počet vláken dynamicky sám.

Počet vláken	**Layer**	**Latch**	**Traversal**
4	20,9	20,7	20,7
8	11,4	11,1	11,3
16	7,1	6,1	6,6
32	4,3	3,9	4,2
64	4,0	3,6	3,5
128	3,9	3,7	3,8
256	4,5	4,1	3,9

Tabulka 4.1: Doba výpočtu úloh na grafu $N = 6$

Počet vláken	Layer	Latch	Traversal
4	90,2	90,3	90,3
8	47,8	47,8	47,6
16	25,9	25,5	25,4
32	14,4	14,1	14,3
64	11,9	11,4	11,3
128	11,6	11,3	11,2
200	11,4	10,8	11,4
256	11,8	11,3	11,4

Tabulka 4.2: Doba výpočtu úloh na grafu $N = 10$

Z těchto výsledků můžeme vyvodit několik zajímavých závěrů. První je závislost mezi velikostí grafu a počtem vláken. Doba výpočtu se s rostoucím počtem vláken snižuje, dokud není dosažena určitá mez, kdy větší počet vláken již dobu výpočtu zvyšuje, což je způsobeno přepínání kontextu. Na menším grafu z tabulky 4.1 je tato mez mezi 64 až 128 vlákny, zatímco na větším grafu z tabulky 4.2 až mezi 200 a 256 vlákny. Z těchto dvou měření můžeme říci, že optimální počet vláken je roven 64, neboť v případě prvního měření 4.1 se při větším počtu vláken rychlost výpočtu zpomaluje a v případě druhého měření již není tak výrazný rozdíl v rychlosti mezi 64 a 200 vlákny.

Není ovšem možné z měření určit skutečný optimální počet vláken, neboť záleží na mnoha faktorech – velikosti grafu, závislosti mezi jednotlivými úlohami, době trvání jednotlivých úloh nebo procesoru a paměti počítače.

Nyní se podíváme na výkon jednotlivých strategií na grafu vytvořeného s parametrem N, kde počet uzlů grafu je $n^3 + n^2 + n + 1$. U paralelních strategií byl nastaven maximální počet vláken na 64. Výsledky vidíme v tabulce 4.3, přičemž hodnoty v ní uvedené jsou opět v sekundách zaokrouhlených na desetiny vypočtených z průměru 10 měřeních.

N	Layer	Latch	Traversal	Reactive	Sequential
3	2,2	1,7	2,2	1,8	13,0
4	2,3	2,2	2,2	2,3	28,1
5	2,9	2,7	2,9	3,2	51,5
6	3,9	3,5	3,6	4,2	85,2
7	5,1	4,9	4,8	5,6	131,2
8	6,7	6,4	6,4	7,2	191,0
9	9,2	8,5	9,0	9,6	266,5
10	12,1	11,3	11,9	11,8	359,2
20	78,9	77,9	77,4	76,8	-
30	253,3	252,5	252,5	233,2	-
40	592,8	587,1	594,5	592,3	-

Tabulka 4.3: Doba výpočtu úloh v závislosti na velikosti grafu

Z výsledku můžeme vyčíst jednoznačný úspěch paralelního spouštění úloh, jenž u grafu velikosti $N = 10$ dosahuje až skoro 32krát rychlejšího času než u sekvenčního spouštění. Rozdíly mezi jednotlivými paralelními strategiemi jsou velmi malé a je tedy těžké určit nejlepší z nich, jelikož se může jednat jen o odchylky v měření. `ReactiveStrategy` dosahuje velmi dobrých výsledku především na větších grafech. Důvodem je však dynamické vytváření vláken, které si Reactor řídí sám. Mohl tedy vytvořit více než 64 vláken, které používají ostatní strategie, což se projeví právě lepším časem u velkých grafů. `LayerStrategy` také nedosahuje špatných výsledků, ovšem nikdy nebyla nejrychlejší strategií. Výhodou pro ni bylo, že se testovalo na grafu, který z většiny času výpočtu umožňuje plné zapojení všech jader procesoru. `LatchStrategy` a `TraversalStrategy` jsou, co se týká doby běhu, velmi vyrovnané strategie.

Na základě těchto výsledku bych doporučoval použití `LatchStrategy`. Oproti `TraversalStrategy` je výhodou, že nevytváří tolik vláken a ve většině případech dosahovala lepších výsledků.

Závěr

Cílem této práce bylo vytvoření nástroje pro měření datové kvality. Datová kvalita je měřena spouštěním různých úloh, např. SQL dotazů, které jsou mezi sebou závislé, a tak tvoří acyklický orientovaný graf. Nástroj tedy musí tyto úlohy umět spouštět ve správném pořadí, a to i paralelně. Dále bylo potřeba zanalyzovat aplikaci Jenkins CI, knihovnu Spring Integration a reaktivní programování. Na základě této analýzy jsem měl zvážit použití Jenkins CI k implementaci a vybrat správné řešení pro budoucí integraci nástroje s jinou aplikací.

Vytvořený nástroj umožňuje sekvenční i paralelní spouštění úloh. Pro paralelní spouštění je implementováno několik algoritmů, které byly výkonnostně otestovány. Pro úlohy je vytvořeno rozhraní, které obsahuje základní úlohy, jakým je třeba SQL dotaz, ale umožňuje i jednoduché přidávání dalších typů úloh. Zároveň nástroj funguje samostatně jako serverová aplikace. K implementaci nebylo využito aplikace Jenkins CI, jelikož to je velmi složitý projekt a vybrat z něj jen tu část podobnou našemu nástroji by bylo složité. Pro budoucí integraci nástroje s jinou aplikací jsem doporučil v případě využití rozhraní REST nebo WebSocket novou knihovnu Spring Webflux, zatímco v jiném případě je vhodný Spring Integration.

Aplikace je připravena do budoucnosti na případná vylepšení. Kromě integrace s jinou aplikací, kterou jsem v práci analyzoval, by bylo možné přidat další implementace úloh měřících datovou kvalitu a jejich výsledků. V budoucnu by nástroj mohl podporovat příkazy pro vypnutí, pozastavení a pokračování v rámci výpočtu úloh.

Literatura

[1] Friedman, T.: Organizations Perceive Significant Cost Impact From Data Quality Issues. [online], Srpen 2009, [cit: 2017-03-03]. Dostupné z: https://www.gartner.com/doc/1131012/findings-primary-research-study-organizations

[2] Lacko, L.: *Business Intelligence v SQL Serveru 2008*. Brno: Computer Press, a. s., 2009, ISBN 978-80-251-2887-9, 456 s.

[3] Renko, S. (editor): *Supply Chain Management - New Perspectives*. In-Tech, Srpen 2011, ISBN 978-953-307-633-1, 784 s.

[4] Batini, C.; Scannapieca, M.: *Data Quality*. Berlin: Springer, 2006, ISBN 978-3-540-33172-8, 262 s.

[5] Kolář, J.: *Teoretická informatika*. Praha: České vysoké učení technické, druhé vydání, Leden 2004, ISBN 80-900853-8-5, 206 s.

[6] Informatica: Data Quality. [online], [cit: 2017-03-03]. Dostupné z: https://www.informatica.com/products/data-quality.html

[7] IBM: Data Quality. [online], [cit: 2017-03-03]. Dostupné z: https://www.ibm.com/analytics/us/en/technology/data-quality/

[8] Ataccama: Ataccama. [online], [cit: 2017-03-03]. Dostupné z: https://www.ataccama.com/

[9] Trillium: Trillium Quality. [online], [cit: 2017-03-03]. Dostupné z: https://www.trilliumsoftware.com/products/tss/data-quality

[10] Jenkins. [online], [cit: 2017-03-03]. Dostupné z: https://jenkins.io/

[11] *Jenkins Documentation*. [cit: 2017-03-07]. Dostupné z: https://jenkins.io/doc/

[12] AbcLinux.cz. [online], Červenec 2011, [cit: 2017-03-07]. Dostupné z: http://abclinuxu.cz/blog/lojzoviny/2011/7/kontinualni-integrace-s-jenkins-ci

[13] Fisher, M.; Partner, J.; Bogoevici, M.; aj.: *Spring Integration in Action*. Shelter Island, NY: Manning, 2012, ISBN 9781935182436, 337 s.

[14] Spring. [online], [cit: 2017-04-23]. Dostupné z: http://spring.io/

[15] Hohpe, G.; Woolf, B.: *Enterprise Integration Patterns*.

[16] 10. Endpoint Quick Reference Table. [online], [cit: 2017-04-23]. Dostupné z: http://docs.spring.io/spring-integration/reference/html/endpoint-summary.html

[17] Fisher, M.; Partner, J.; Bogoevici, M.; aj.: *Spring Integration Reference Manual*. [cit: 2017-04-23]. Dostupné z: http://docs.spring.io/spring-integration/reference/html/

[18] 17. HTTP Support. [online], [cit: 2017-04-23]. Dostupné z: http://docs.spring.io/spring-integration/reference/html/http.html#http-samples

[19] Tsvetinov, N.: *Learning Reactive Programming with Java 8*. Birmingham: Packt Publishing, Červen 2015, ISBN 978-1-78528-872-2, 235 s.

[20] Understanding Reactive types. [online], Duben 2016, [cit: 2017-04-23]. Dostupné z: http://spring.io/blog/2016/04/19/understanding-reactive-types

[21] *org.reactivestreams (reactive-streams 1.0.0 API)*. [cit: 2017-04-23]. Dostupné z: http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/

[22] Maldini, S.; Baslé, S.: *Reactor 3 Reference Guide*. [cit: 2017-04-23]. Dostupné z: http://projectreactor.io/docs/core/release/reference/docs/index.html

[23] Project Reactor. [online], [cit: 2017-04-23]. Dostupné z: https://projectreactor.io/

[24] ReactiveX. [online], [cit: 2017-04-23]. Dostupné z: http://reactivex.io/

[25] *23. WebFlux framework*. [cit: 2017-04-23]. Dostupné z: http://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/html/web-reactive.html

[26] Spring Boot. [online], [cit: 2017-04-23]. Dostupné z: http://projects.spring.io/spring-boot/

[27] GitHub - jenkinsci/jenkins: Jenkins automation server. [online], [cit: 2017-04-23]. Dostupné z: `http://github.com/jenkinsci/jenkins`

[28] *Java Platform, Standard Edition 8 API Specification.* [cit: 2017-05-08]. Dostupné z: `https://docs.oracle.com/javase/8/docs/api/`

Příloha **A**

Seznam použitých pojmů a zkratek

API – **Application Programming Interface** je rozhraní pro programování aplikací.

BI – **Business intelligence** je množina konceptů a metodik, které se používají při práci s firemními daty.

Callback – Volání části programu, který se vykoná v rámci běhu jiného programu.

CI – **Continuous integration** jsou metody a nástroje k urychlení vývoje softwaru.

Deadlock – Uváznutí programu vlivem špatné synchronizace paralelního výpočtu.

DFS – **Depth-first search** je algoritmus prohledávaní grafu do hloubky.

DSL – **Doménově specifický jazyk** je programovací jazyk zaměřen na jeden specifický problém.

DWH – **Data warehouse** – datový sklad, úložiště pro velké množství dat.

ETL – **Extract, transform, load** se nazývá fáze Business intelligence, při které dochází k získání, transformování a nahrání dat.

Framework – Aplikační rámec, jenž poskytuje různé programy a knihovny.

Getter – Metoda objektu, která vrací jeho atribut.

HTTP – **Hypertext Transfer Protocol** je internetový protokol.

I/O – **Input/Output** nebo-li vstup/výstup.

JRE – **Java Runtime Environment** je rozhraní pro spouštění Java aplikací.

JSON – **JavaScript Object Notation** je způsob zápisu dat.

JVM – **Java Virtual Machine** je virtuální stroj spouštějící Java aplikace.

Message-driven architektura – Architektura založena na posílání zpráv (zprávami řízená architektura).

Open source – Počítačový program s otevřeným zdrojovým kódem.

Package – Jmenný prostor sloužící k organizaci Java tříd.

POJO – **Plain Old Java Object** je obyčejný Java objekt.

REST – **Representational State Transfer** je rozhraní postavené na HTTP.

Setter – Metoda objektu, která nastavuje jeho atribut.

Singleton – Návrhový vzor, který zajišťuje aby v aplikaci existovala jen jedna instance dané třídy.

SSE – **Server-sent events** je technologie pro automatické posílání zpráv klientovi ze serveru.

SQL – **Structured Query Language** je jazyk pro práci s daty v relačních databázích.

URI – **Uniform Resource Identifier** nebo-li jednotný identifikátor zdroje.

XML – **Extensible markup language** je značkovací jazyk používající se pro serializaci dat.

WAR – **Web application Archive** je formát souboru pro uchovávání webových aplikacích napsaných v programovacím jazyku Java.

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Maly_Vojtech_2017.pdf` ..... text práce ve formátu PDF