Mendelova univerzita v Brně
Provozně ekonomická fakulta

---

# Návrh a implementace datového úložiště pro projekt SMART faculty

**Diplomová práce**

Vedoucí práce:
Ing. Jan Kolomazník, Ph.D.

Bc. Petr Sadovský

Brno 2016

Chtěl bych poděkovat vedoucímu diplomové práce panu Ing. Janu Kolomazníkovi, Ph.D. za jeho ochotu, cenné připomínky a pomoc ve formě odborných konzultací vedených během dokončování práce. Dále bych rád poděkoval za dohled v průběhu studia mému otci Mgr. Petru Sadovskému a také za podporu celé mé rodině.

**Čestné prohlášení**

Prohlašuji, že jsem tuto práci: **Návrh a implementace datového úložiště pro projekt SMART faculty**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

Brno, 1.5.2017 .....

## Abstract

Sadovský, P. Design and implementation of data storage for the SMART faculty project. Brno, 2017

In this diploma thesis an analysis is carried out and on the basis of it a design of a suitable data repository for the SMART faculty project is made. For the storage acces a REST interface is implemented, which is being tested at the last stage of the work.


**Key words:** REST, NoSQL, data storage, JSON, replication


## Abstrakt

Sadovský, P. Návrh a implementace datového úložiště pro projekt SMART faculty. Brno 2017

V této diplomové práci se provádí analýza a na základě ní návrh vhodného datového úložiště pro projekt SMART faculty. K přístupu do úložiště je implementováno REST rozhraní, u něhož v poslední fázi práce probíhá testování.


**Klíčová slova:** REST, NoSQL, datové úložiště, JSON, replikace

# Obsah

# Seznam obrázků

# Seznam tabulek

# 1 Úvod práce

Účelem projektu SMART faculty (SMART Pef) je zpříjemnit a zjednodušit život osobám kooperujícím s Provozně ekonomickou fakultou. Tohoto cíle se bude dosahovat různými způsoby a každý má jinou metodiku, nicméně některé se prolínají.

Na vývoji projektu se podílí tým složený ze studentů a zaměstnanců fakulty. Každému je přidělena určitá oblast zaměření.

Jedna oblast zahrnuje instalaci zařízení, měřících různé fyzikální veličiny v prostorách fakulty. Generují frekventovaně data narůstající do velkých objemů. Data musí být někde zpracována, persistována a případně distribuována k dalšímu použití.

## 1.1 Cíl práce

Provést analýzu datových úložišť, které by mohly být vhodné pro ukládání dat produkovaných měřícími zařízeními. Z analýzy vytvořit návrh a na základě konzultace s týmem jedno úložiště vybrat. Poté si osvojit práci s ním.

Pro přístup k úložišti implementovat REST rozhraní, jenž data přijme, zpracuje, uloží a umožní je dále distribuovat.

Další fáze zahrnuje otestovaní robustnosti implementovaného řešení s různými konfiguracemi úložiště.

Na závěr vytvořit ekonomické zhodnocení efektivity návrženého řešení.

# 2 Rešeršní část

Níže v této kapitole se nachází přehled technologií, nástrojů a zároveň stručná teorie, která je žádoucí pro pochopení jednotlivých postupů vedoucích k dosažení předem stanoveného cíle práce.

## 2.1 REST

Termín REST byl poprvé zmíněn v roce 2000, v rámci disertační práce Roye Fieldinga[1] s názvem *Architectural Styles and the Design of Network-based Software Architecture*, kde REST popisuje jako architektonický styl pro distribuované hypermediální systémy (Doglio, 2015).

Jednoduše řečeno REST (REpresentational State Transfer) je architektonický styl definovaný za účelem pomoci vytvářet a organizovat distribuované systémy (Doglio, 2015).

Může být zprostředkován různými protokoly (HTTP, FTP atd.), nicméně nejrozšířenější bývá použití HTTP.

### 2.1.1 Zdroje

Hlavními stavebními bloky architektury jsou **zdroje**. Cokoliv, co lze pojmenovat, může být zdroj (webová stránka, obrázek, osoba apod.). Specifikace zdrojů dle (Doglio, 2015):

- reprezentace - způsob reprezentace dat[2] - binárně, JSON, XML atd.,
- identifikátor - URL, která vrací pouze jeden zdroj v daný čas,
- metadata - Content-type, čas poslední změny apod.,
- kontrolní data - cache-control, is-modifiable-since.

Co se týká reprezentace dat, JSON je jako formát velmi rozšířený a dostupný v řadě programovacích jazyků.

Identifikátor zdroje by měl poskytovat unikátní způsob identifikace v daný moment a zároveň úplnou cestu ke zdroji (Doglio, 2015). Tím, jak se REST váže na HTTP, cestu ke zdroji reprezentuje URI (Unique Resource Identifier). Například adresa `GET /api/v1/books`, by mohla vracet z databáze seznam všech knih.

Od RESTu se také očekává, že bude schopen poskytnout CRUD (Create, Read, Update and Delete) operace nad zdroji. Tyto operace mohou být přímo namapovány do *HTTP Verbs* dle (Doglio, 2015):

- **GET** - přistupuje ke zdroji pouze pro čtení (**read** operace),

[1] Americký počítačový vědec narozený roku 1965, mimochodem patří zároveň mezi přední autory HTTP protokolu.

[2] Jeden zdroj může mít více reprezentací

- **POST** - zaslání nového zdroje na server (**create** operace),
- **PUT** - aktualizace daného zdroje (**update** operace),
- **DELETE** - mazání zdroje (**delete** operace),

Kromě CRUD operací, poskytuje další, které však nejsou tak intenzivně používáné (HEAD, OPTIONS). Nicméně většinu služeb jde zajistit pomocí GET a POST.

Další výhodou REST založeného na HTTP protokolu představují HTTP status kódy. Dělí se do pěti množin a každý představuje trojmístné číslo, jenž začíná pořadovým číslem množiny. V nejlepším případě se očekávají kódy z druhé množiny, označující úspěšné provedení operace. Ostatní značí buď neutrální informaci nebo výskyt problému.

Kromě výše zmíněných má REST další specifikace, avšak tyto postačí pro pochopení základní funkce.

## 2.2 Přehled použitých technologií

Výčet obsahuje technologie použité k implementaci a zprovoznění REST API.

### 2.2.1 Java Enterprise Edition

Jak je z názvu možné předpokládat, jedná se o platformu jazyka Java. Poskytuje API a běhové prostředí pro vývoj a běh vysoce škálovatelných, několika stupňových, spolehlivých a zabezpečených síťových aplikací (Oracle, 2012). Základem této platformy je platforma Java SE (Standard Edition), nad ní jsou pak definovany součásti Java EE.

Dříve se pro tuto platformu používalo označení J2EE (Java 2 Enterprise Edition), která byla původně vytvořena firmou Sun Microsystems v letech 1999-2000. Od verze 1.3 je vývoj veden v rámci JCP (Java Community Process). JCP je mechanismus pro vývoj standartních technických specifikací technologie Java (Java Community Process, 2017).

### 2.2.2 Spring Boot

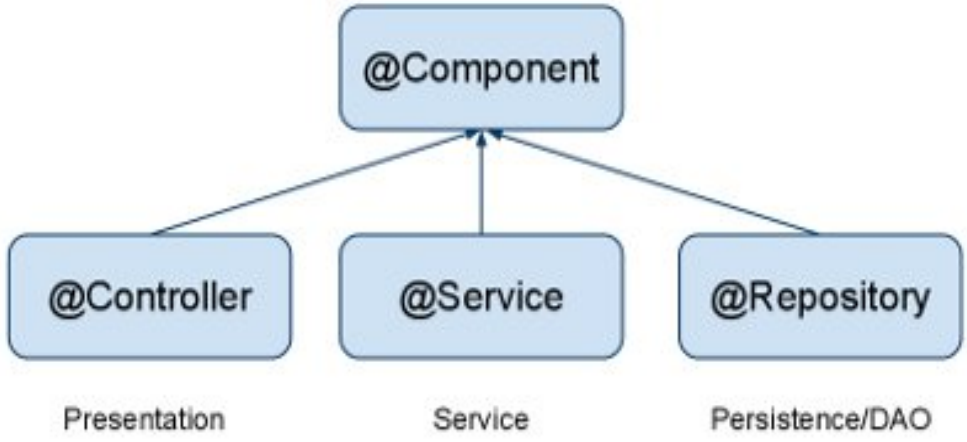
Usnadňuje vytváření samostatných, produkčně klasifikovaných aplikací od Spring[1]. Tvůrci převzali dogmatický pohled z platformy Springu a knihoven třetích stran, takže je možné vyvíjet s minimálním úsilím. Většina Spring Boot aplikací potřebuje jen zlomek konfigurace ze Springu (Webb, 2017).

Mezi primární cíle technologie je v krátkém čase nabýt rozsáhlé prvotní zkušenosti s vývojem ve Springu. Spring Boot poskytuje řadu vlastností, mezi něž patří

[1]Open-source framework pro vývoj Java EE aplikací.

např. embedded servery, bezpečnost, metriky, stavové kontroly a spustitelné konfigurace. Žádné generování kódu a také žádné požadavky na XML konfiguraci (Webb, 2017).

Stále více se z XML konfigurací přechází na tzv. „Java based“ přístup, což je konfigurování přes anotace. Pro přehled uvádím několik základních anotací, které jsou ve Springu implementovány a při implementaci REST budou použity:




Obrázek 1: Spring anotace, (TechFerry, 2015)

Anotace @Component je předek anotací @Controller, @Service a @Repository. Lze z názvu anotací odvodit, že jsou rozděleny podle vrstev aplikace. Jakmile se nakonfiguruje Spring v kořenovém balíčku, pokud nejsou požadavky jiné, od tohoto balíčku se detekují jednotlivé anotace. Zajistí, že anotovaná třída se incializuje Springem a vznikne objekt (Java Bean) existující v rámci něj.

Další důležitá anotace, která však není na obrázku, je @Autowired. Používá se pro vkládání incializovaných objektů na různá místa v aplikaci[1].

Dále anotace @Value s parametrem klíče, načte hodnotu klíče z konfiguračního souboru s příponou .properties (defaultně application.properties) a injektuje do atributu objektu.

### 2.2.3 Gradle

Jedná se o open source buildovací nástroj (licencovaný pod ASL), který nahradil XML skripty za DSL (Domain Specific Language), jež jsou založeny na programovacím jazyce Groovy (Kainulainen, 2017). Spojuje koncepty Apache Ant a Apache Maven. Gradle posunul deklarativní build aplikace na další úroveň, a to poskytnutím deklarativních jazykových prvků, které si můžeme sestavit podle sebe. Tyto prvky

[1]Využívá se přitom principu IoT (Inversion of Control), který přenáší kontrolu nad objekty nebo částí programu do kontejneru frameworku. Dosáhnutí IoT lze pomocí různých návrhových vzorů: Strategy, Factory, Service Locator nebo Dependency Injection (DI).

podporují sestavování dle konvencí pro Java, Groovy, OSGi, Web a Scala projekty. Dokonce je možné provést rozšíření, vytvořením nového jazykového prvku nebo vylepšení již existujícího (Gradle Build Tool 3.3, 2017). Využívá se jak pro sestavování enterprise aplikací, tak i pro mobilní aplikace běžící na operačním systému Android.

### 2.2.4 Apache Tomcat 8

Open source implementace zahrnující technologie Java Servlet, JavaServer Pages, Java Expression Language a Java WebSocket. Specifikace těchto technologií jsou vyvíjeny pod JCP (Apache Tomcat, 2017). Apache Tomcat je tedy webový server, nebo také servlet kontejner, který slouží k běhu javovských webových aplikací.

## 2.3 Přehled použitých nástrojů

Tato část stručně specifikuje použité vývojové prostředí a testovací nástroj.

### 2.3.1 IntelliJ Idea

Tvůrci (JetBrains s.r.o.) prezentují tento produkt jako nejchytřejší vývojové prostředí pro Javu. Existuje ve dvou verzích, kde první je komerční Ultimate a druhé open-source Community[1]. Dá se říct, že Community představuje ořezanou verzi Ultimate.

Má zabudovanou podporu pro nespočet technologií, jako jsou databázové nástroje, aplikační servery, frameworky, verzovací systémy apod. Firma nabízí pro studenty licenci verze Ultimate zdarma.

### 2.3.2 JMeter

Výběr tohoto nástroje vychází ze srovnání s dalšími nástroji Tsung, Gatling a The Grinder. To se nachází na blogu BlazeMeter, kde autor Dmitri Tikhanski na jednom stroji testuje případ s HTTP GET požadavky, pro 20 vláken (uživatelů) a sto tisíc iterací. Ve výsledku JMeter disponuje nejkratší odezvou při nejvyšší průměrné propustnosti (Tikhanski, 2015).

JMeter je open-source aplikace napsaná v jazyce Java, jejíž účel spočívá v testování zatížení a měření výkonu. Původně sloužila pouze pro testování webových aplikací, avšak byla rozšířena i pro další testovací funkce (Apache JMeter, 2017).

Umožňuje testovat výkon statických a dynamických zdrojů (resources). Používá se k simulaci vysokého zatížení v různých prostředích, například na serveru, skupině serverů nebo síti.

Poskytuje širokou škálu testovacích možností. Mezi hlavní patří Test IDE (tvorba testovacích plánů), testování z příkazové řádky, více vláknové zpracování, vysoce rozšiřitelné jádro aplikace, schopnost testovat zatížení a výkon různých typů aplikací, serverů nebo protokolů (v našem případě REST).

[1] https://www.jetbrains.com/idea/

## 2.4 NoSQL databáze

Problém velkého objemu dat (Big Data) a jejich různorodosti, zapříčinil vznik nových databázových systémů, jež se nazývají NoSQL. Všeobecně jsou zařazovány mezi třetí databázovou revoluci (Andlinger, 2013).

### 2.4.1 Big Data

Mají úzkou souvislost s NoSQL databázemi. Jedná se o široký pojem, u něhož probíhá soutěž mezi mnohými definicemi.

Big Data jsou data tak velká, že je není možné zachycovat, spravovat ani zpracovávat pomocí běžně dostupných softwarů tak, abychom s nimi mohli pracovat v rozumném čase. Je to tedy masivní objem nestrukturovaných dat. Za účelem získání vypovídací hodnoty z nich, je nutné, aby je uměl daný objekt vyhodnotit. Představují šanci pro všechny tržní sektory. Mohou pomoci získat více informací o zákaznících, zefektivnit poskytování produktů a služeb, zlepšit komunikaci se zákazníky, vytvořit nové výnosové trhy a pomoci tedy k lepšímu marketingu (Rosenkrancová, 2016).

Nyní lze ukládat a zpracovávat všechny data - strojově generovaná, multimédia, sociální sítě a transakční data - v jejich původním hrubém formátu a udržovat je trvale (Harrison, 2015).

### 2.4.2 Třetí databázová revoluce

Objem ukládaných dat stále narůstá a zároveň s tímto procesem roste i potřeba využívat databázové systémy. Nejvíce známé jsou relační databáze využívající pro zpracování dat tabulkové schémata. Současně s rapidním nárůstem dat se také setkáváme s jevem, kdy data začínají být pouze částečně strukturovaná, například záznamy se mohou lišit v počtu atributů, což může způsobit problém při ukládání. To má za následek, že tradiční správa dat pomocí schémat a relačních referencí přestává být použitelná (Panyko, 2013).

Termín **NoSQL** lze vysvětlit různě, avšak nejpoužívanějším je akronym „Not only SQL“, tedy technologie jdoucí proti SQL. Neznamená to však, že tyto systémy neumožňují SQL jazyk, především jde o práci s velkými objemy dat na místo aplikování klasického RDBMS (relační systém řízení báze dat).

Motivací k užití tohoto přístupu je mnoho, mezi nimi jsou např. jednoduchost designu, horizontální i vertikální škálovatelnost a jemnější kontrola dostupnosti. Obecně je použitelnost daného typu NoSQL databáze dána podle řešeného problému (Andlinger, 2013). V kontextu CAP teorému NoSQL úložiště často potlačují konzistenci ku prospěchu dostupnosti a tolerance k narušení sítě. Mezi bariéry pro rozsáhlejší nasazení do praxe patří nepřítomnost plnohodnotné podpory transakčního modelu ACID, použití nízkoúrovňových dotazovacích jazyků, nedostatečná standardizace rozhraní a vysoké realizované investice podniků do relačních databází v minulosti (Grolinger, Higashino, Tiwari, Capretz, 2013).

### 2.4.3 Transakční model ACID

Jak již bylo zmíněno, relační databáze podporují transakční model ACID, jehož čtyři písmena obsažená v názvu představuji následující vlastnosti dle (Rychlý, Kolář, 2013):

- **Nedělitelnost (Atomicity)** - atomičnost transakcí, transakce proběhne vždy celá,
- **Konzistence (Consistency)** - v DB jsou pouze platná data dle daných pravidel,
- **Separace (Isolation)** - souběžné transakce se neovlivňují,
- **Trvanlivost (Durability)** - uskutečněná transakce nebude ztracena.

### 2.4.4 CAP teorém

Další pojem, který je třeba v rámci sdílených/distribuovaných systémů vysvětlit je Brewerův teorém - CAP teorém. Jeho hlavní princip zahrnuje tři podstatné systémové požadavky pro úspěšný design, implementaci a nasazení aplikace v distribuovaném systému. Vyjmenování požadavků a popis dle (Rychlý, Kolář, 2013):

- **Konzistence (Consistency)** - každý uzel/klient vidí ve stejný čas stejná data, data konzistentní nezávisle na běžících operacích či jejich umístění,
- **Dostupnost (Availability)** - každý požadavek obsloužen, úspěšně nebo neúspěšně, nepřetržitý provoz, vždy možnost zapsat nebo číst data,
- **Odolnost vůči dělení (Partition Tolerance)** - funkční navzdory chybám sítě nebo výpadkům uzlů.

Co se týká sdílených systémů, mohou být zpracováný pouze 2 ze 3 požadavků (Rychlý, Kolář, 2013).

**BASE**

Acronym BASE byl definován stejně jako CAP teorém Ericem Brewerem. Lze říci, že jde o alternativu k ACID modelu, kdy BASE model je vyvinut pro práci s nestrukturovanými daty v NoSQL databázi. Popis BASE modelu:

- **Basically Available** - aplikace funguje v podstatě celou dobu, NoSQL databáze je distribuována skrze více úložišť s vysokým stupněm replikace,
- **Soft-state** - nemusí být konzistentní po celou dobu, konzistence je starost vývojářů,
- **Eventual consistency** - hlavní podmínkou databáze je, že v určitém okamžiku v budoucnu se budou data sbíhat do konzistentního stavu.

### 2.4.5 Srovnání ACID a BASE

Srovnání výše zmíněných modelů:

Tabulka 1: Srovnání, (Rychlý, Kolář, 2013)

| **ACID** | **BASE** |
|---|---|
| silná konzistence | slabá konzistence (stará data) |
| izolovanost | především dostupnost |
| orientace na commit | přibližné odpovědi |
| vnořené transakce | jednodušší, rychlejší |
| teoreticky dostupnost | maximální dodávka dat |
| konzervativní | agresivní |
| složitá evoluce (schématu, ...) | jednodušší evoluce |

## 2.5 Typy NoSQL databází

Databází NoSQL existuje mnoho, avšak je důležité se zaměřit na ty základní, které jsou nejčastěji zmiňovány v různých informačních zdrojích, zabývajících se touto problematikou.

### 2.5.1 Dokumentové databáze

Idea dokumentového úložiště je klasifikace „dokumentu“. V tomto úložišti se ukládají především data, která jsou formátována nebo zakódována v XML, YAML, JSON (Java Script Object Notation), BSON (binární reprezentace formátu JSON s vnořenými sadami klíčů a hodnot). Každý dokument v úložišti je přiřazen k unikátnímu klíči, který slouží jako identifikátor. Datový model pracuje s nějakým API nebo dotazovacím systémem pro přístup k dokumentům (Sharma, S Tim, Gadia, Shandilya, Peddoju, 2014).

Příklad uloženého dokumentu:

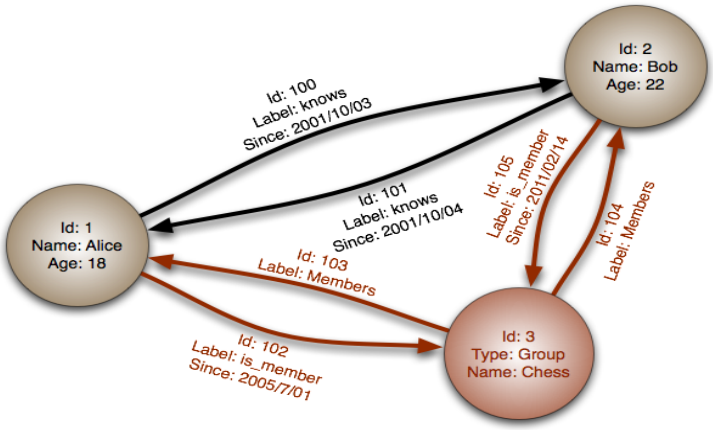
```
{
  "_id" : "58e5631f6a9ae3b028bfaa4e",
  "name" : "Peter"
  "birthDateTimestamp" : "1490960282",
}
```

**Vybraní zástupci**: MongoDB, CouchDB, OrientDB, CouchBase, RavenDB

### 2.5.2 Grafové databáze

Data jsou reprezentována a ukládána podél uzlů, hran a mají vlastnosti grafové struktury. Poskytují spojení mezi jednotlivými elementy pomocí jejich přímých ukazatelů. Tímto způsobem je možné se vyhnout hledání indexů. Používají se pro zná-

zornění sítí a jejich topologií např. sociální, dopravní nebo topologie počítačových sítí (Sharma, S Tim, Gadia, Shandilya, Peddoju, 2014).




Obrázek 2: Reprezentace dat v grafové databázi, (Kumar, 2016)

**Vybraní zástupci**: Neo4J, OrientDB, InfiniteGraph, AllegroGraph, Virtuoso a Stardog

### 2.5.3 Databáze klíč-hodnota

Mezi nejjednodušší datové struktury, které jsou schopny uchovávat klíč a jeho hodnotu, patří asociativní pole nebo mapy. Pro přístup k datům se využívají hash tabulky, což umožňuje přistupovat k datům v konstantním čase. Datový typ pro ukládání do databáze je BLOB (Binary Large OBject), jenž přijme jakákoliv data. Zpracování uložené hodnoty je na aplikaci, databáze ji přijme jako celek (Rychlý, Kolář, 2013).

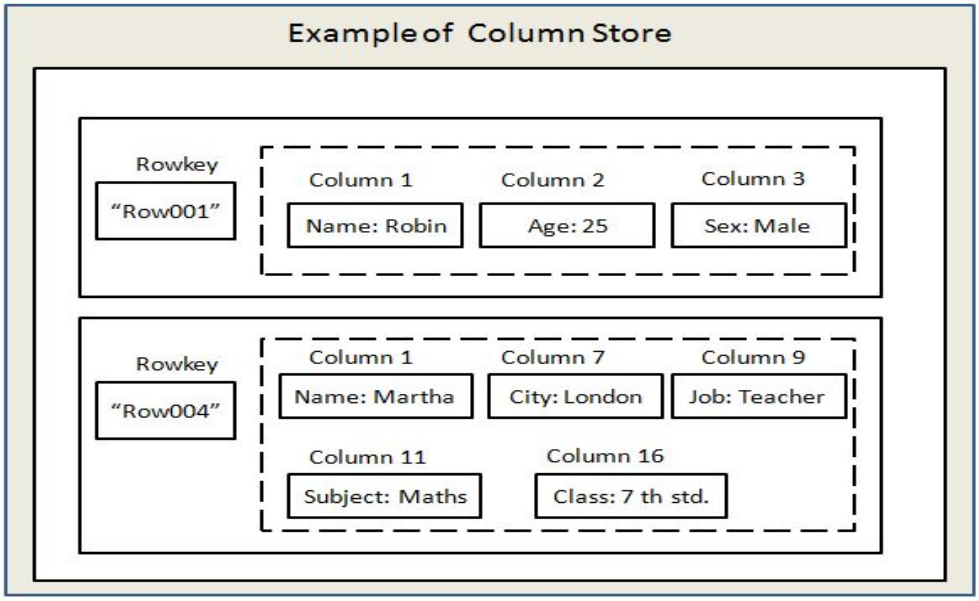
| Key | Value |
|---|---|
| "India" | {"B-25, Sector-58, Noida, India – 201301" |
| "Romania" | {"IMPS Moara Business Center, Buftea No. 1, Cluj-Napoca, 400606",City Business Center, Coriolan Brediceanu No. 10, Building B, Timisoara, 300011"} |
| "US" | {"3975 Fair Ridge Drive. Suite 200 South, Fairfax, VA 22033"} |

Obrázek 3: Reprezentace dat v databázi typu klíč-hodnota, (Kumar, 2016)

**Vybraní zástupci**: Oracle NoSQL, Dynamo, Redis, Voldemort, Riak, Velocity

### 2.5.4 Sloupcové databáze

Sloupcově orientované databáze ukládají data ve sloupcové formě a ne v řádkové, jak tomu je u relačních databází (Panyko, 2013). Abstraktně jsou to úložiště s dvourozměrným klíčem a hodnotou. Klíč představuje název sloupce. Sloupce mohou být pro každý řádek různé. Jsou přičitány k dynamickým záznamům, které ukládají data a můžou pojmout obrovské počty dynamických sloupců. Lze tedy tento typ úložišť považovat za rozšiřitelný (Sharma, S Tim, Gadia, Shandilya, Peddoju, 2014).




Obrázek 4: Struktura dat ukládaných do sloupcové databáze, (Dhaneshwar, 2015)

**Vybraní zástupci**: BigTable, Cassandra, HBase, Vertica, Accumulo, Druid

## 2.6 Navrhovaná úložiště

Jak již bylo zmíněno, výběr vhodného typu úložiště se váže na řešený problém. Data proudící z měřících zařízení jsou ve formátu JSON, čili by se mohl výběr omezit pouze na dokumentové databáze.

Níže uvedená tabulka z velké většiny vychází z webu DB-Engines[1], případně jsou informace doplněny z oficiálních dokumentací.

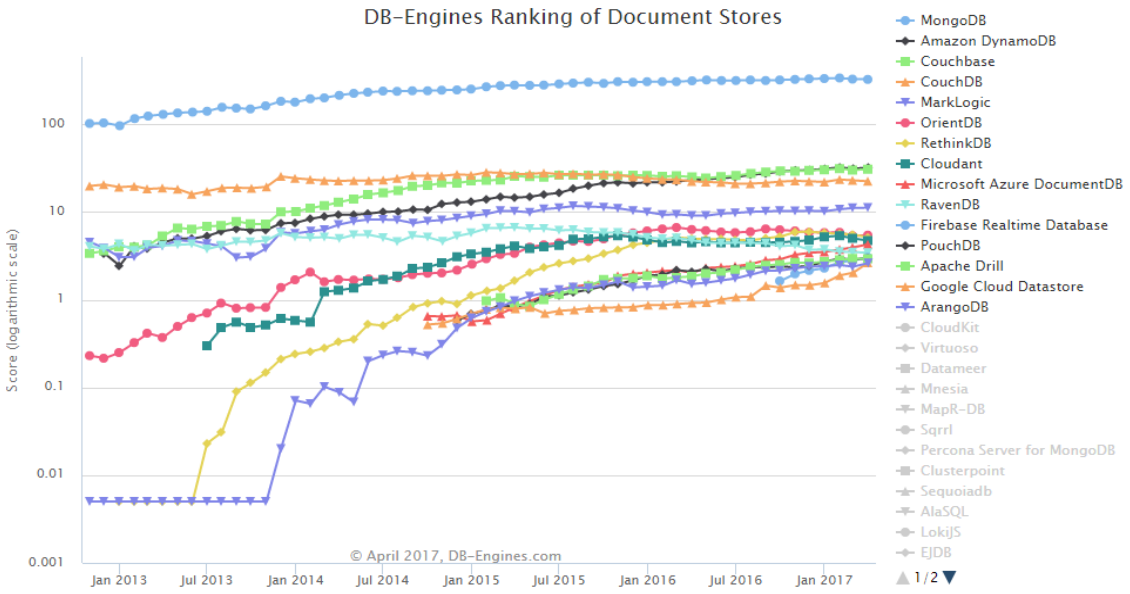
[1]https://db–engines.com/en/system/CouchDB%3BCouchbase%3BMongoDB%3BOrientDB%3BRavenDB

Tabulka 2: Zástupci dokumentových databází

| | **MongoDB** | **CouchDB** | **CouchBase** | **OrientDB** | **RavenDB** |
|---|---|---|---|---|---|
| Implementace | C++ | Erlang | C++, Erlang, C, Go** | Java | .NET |
| Kompatibilita | Java, C, C++, C#, Perl, PHP, Python, Ruby, Scala, Node.js | Jakýkoliv jazyk umožňující HTTP požadavky | C, .NET, Java, Python, Ruby, PHP, LINQ, Node.js | Java | .NET, Java, Python |
| Formát uložení | JSON (BSON) | JSON | JSON | JSON | JSON |
| Replikace | Ano (Master-slave) | Ano (Master-master, Master-slave) | Ano (Master-master, Master-slave) | Ano (Master-master) | Ano (Master-master) |
| Podpora ACID | Ne* | Ne* | Ne* | Ano | Ano |
| Server-side skripty | Javascript (MapReduce) | Javascript (MapReduce) | Javascript (N1QL, MapReduce, Spatial View) | Javascript**** | Neuvedeno v dokumentaci |
| Indexování | B-Tree | B+Tree*** | B+Tree | SB-Tree (B–Tree) | Vlastní datová struktura (Lucene) |
| Podpora OS serveru | Linux<br>OS X<br>Solaris<br>Windows | Android<br>BSD<br>Linux<br>OS X<br>Solaris<br>Windows | Linux<br>OS X<br>Windows | OS s podporou Java JDK >= 6 | Linux<br>Windows |

* Nepodporuje transakce nad více dokumenty, pouze nad jedním, což v určitých případech může stačit.
** CouchBase vzniklo spojením technologií z databází Membase a CouchDB.
*** Mírně pozměněný B-Tree za účelem navýšení rychlosti, což zprostředkovává hospodařením s místem na disku.
**** Defaultně instalován, nicméně podpora všech JVM jazyků.

Z informací o jednotlivých úložištích vyplývá, že disponují převážně stejnými vlastnostmi, proto bude vhodné zahrnout další kritérium pro jednoznačnost výběru. Kritériem je srovnání obliby mezi uživateli, napříč všemi dokumentovými databázemi.




Obrázek 5: Popularita dokumentových databází, (DB-Engines, 2017)

Z grafu lze vyčíst, že MongoDB značně uživatelé upřednostňují před ostatními, čili jeho nasazení na projektu má své opodstatnění.

### 2.6.1 MongoDB

MongoDB je multiplatformní open-source dokumentová databáze, která byla poprvé vydána v roce 2009 společností 10gen. Databáze ukládá dokumenty do kolekcí. Můžeme si je představit jako relační tabulky s tím rozdílem, že tabulky mají povinnost dodržovat schéma relační databáze, ale kolekce v MongoDB nejsou nijak omezeny. Různé dokumenty mohou být spojovány v jednu kolekci, měly by však být podobné za účelem lepší indexace. Ukládaná data v kolekcích musí splňovat předpoklad, že je jde vyjádřit v JSON. Podporuje všechny jeho datové typy, tedy string, integer, boolean, double, null, array a objekt. Díky podpoře BSON formátu lze přidat ještě date, objekt id, regulární výrazy a binární data (Panyko, 2013).

Základní specifikace MongoDB:

Tabulka 3: Specifikace MongoDB, (Kovacs, 2016)

| | |
|---|---|
| **Psáno v**: | C++ |
| **Hlavní pointa:** | ukládání dokumentů v JSON |
| **Licence:** | AGPL (Drivers: Apache) |
| **Protokol:** | vlastní, binární (BSON) |

Nejvhodnější případy užití (Kovacs, 2016):

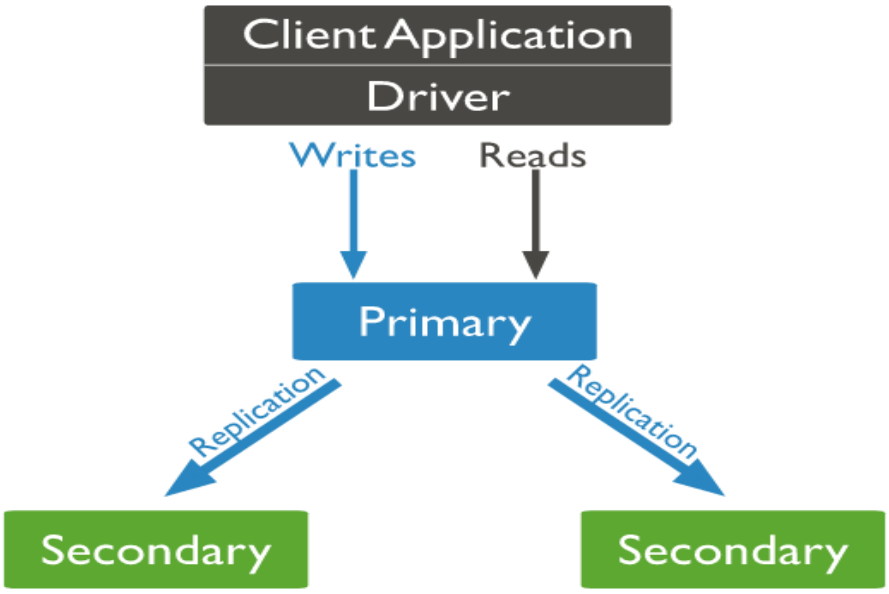
- při používání dynamických dotazů,
- upřednostňování indexů před map/reduce funkcemi,
- potřeba vysokého výkonu nad velkou databází,
- pokud změny dat jsou prováděny příliš rychle, což zabírá hodně paměti.

### 2.6.2 Replikace MongoDB

Replikace zajišťuje redundanci a zvyšuje dostupnost dat. Díky několika kopiím dat vedených skrze více databázových serverů zajišťuje replikace větší odolnost vůči problému s výpadkem nějakého serveru (MongoDB, 2017).

Množina replik jsou mongod (příkaz pro spuštění serveru) instance provádějící údržbu nad stejnou datovou množinou. Mezi instancemi je několik uzlů jako datových nosičů a libovolný jeden z nich rozhoduje, o tom kam data poputují (MongoDB, 2017).

Primární uzel přijímá všechny operace zápisu, bez něj není možné, aby byly přijaty. Vytváří z každé změny na datové množině in-memory kolekci nazvanou operační log (oplog - operation log). Sekundární uzly pak vytváří kopii tohoto logu a zároveň aplikují operace z něj na vlastní datové množiny, tím vzniká odraz primární datové množiny. V případě, že by primární uzel havaroval, je sekundární oprávněn provést volbu a stanovit i sebe sama novým primárním uzlem.




Obrázek 6: Rozdělení replik, (MongoDB, 2017)

Počet replik může být vyšší než 50, avšak počet volících uzlů maximálně 7. Každý další už je bez oprávnění volit primární uzel.

Množina má obvykle 3 uzly při nasazování do produkčního prostředí. Z toho vyplývá další předpoklad, že počet hlasujících uzlů musí být liché číslo. Pokud by tomu tak nebylo, lze přidat **arbitra**, což je další mongod instance. Slouží pouze pro volbu primárního uzlu, nevytváří kopii dat a nemůže se jím stát sám. Odůvodnění lichého počtu uzlů souvisí s chybovou tolerancí, jenž udává počet uzlů, které mohou vypadnout a i přesto jich zůstane dostatek, aby došlo ke zvolení primárního (MongoDB, 2017).

Tabulka 4: Chybová tolerance, (MongoDB, 2017)

| **Number of Members** | **Majority Required to Elect a New Primary** | **Fault Tolerance** |
|---|---|---|
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |

Dále v posloupnosti dle tabulky, by se při počtu 7 uzlů, chybová tolerance zvýšila na 3, ale při 8 by zůstala stejná. Tedy při každém lichém počtu dochází ke zvýšení odolnosti vůči chybám.

# 3 Metodika práce

V kapitole se nachází seznam jednotlivých kroků s jejich stručným popisem. Každý je pak detailně zdokumentován v další kapitole Vlastní práce.

**Registrace projektu data-endpoints**

Celý projekt SMART faculty se skládá z menších projektů (podprojektů) a jedním z nich je i data-endpoints. Multiprojektový build se ukládá do souboru settings.gradle.

**Konfigurace pro build projektu**

V kořenovém adresáři hlavního projektu se nachází konfigurační soubor build.gradle, do něhož se zapisuje konfigurace pro build projektu.

**Konfigurace pro build modulů**

Build konfigurace každého modulu se přizpůsobuje především jeho funkcionalitě.

**Instalace a konfigurace MongoDB**

Bude prováděna instalace Community serveru MongoDB a zároveň jeho konfigurace.

**Architektura REST API**

Pro lepší orientaci, před nadcházející implementací REST rozhraní, bude uveden stručný popis architektury.

**Implementace modulu data-endpoints-core**

Kromě implementace vrstev service a repository, musí být nakonfigurován přístup přes aplikaci do databáze.

**Implementace modulu data-endpoints-api**

Zprostředkovává komunikaci mezi REST rozhraním a klientem. Metody dostupné z modulu data-endpoints-core se namapují na specifické URL adresy pro příjem HTTP požadavků. Na závěr bude vygenerována dokumentace rozhraní.

**Testování navrženého řešení**

Implementované způsoby zpracování dat v rámci REST API budou testovány pomocí nástroje JMeter. V první fázi MongoDB server poběží na jednom stroji a

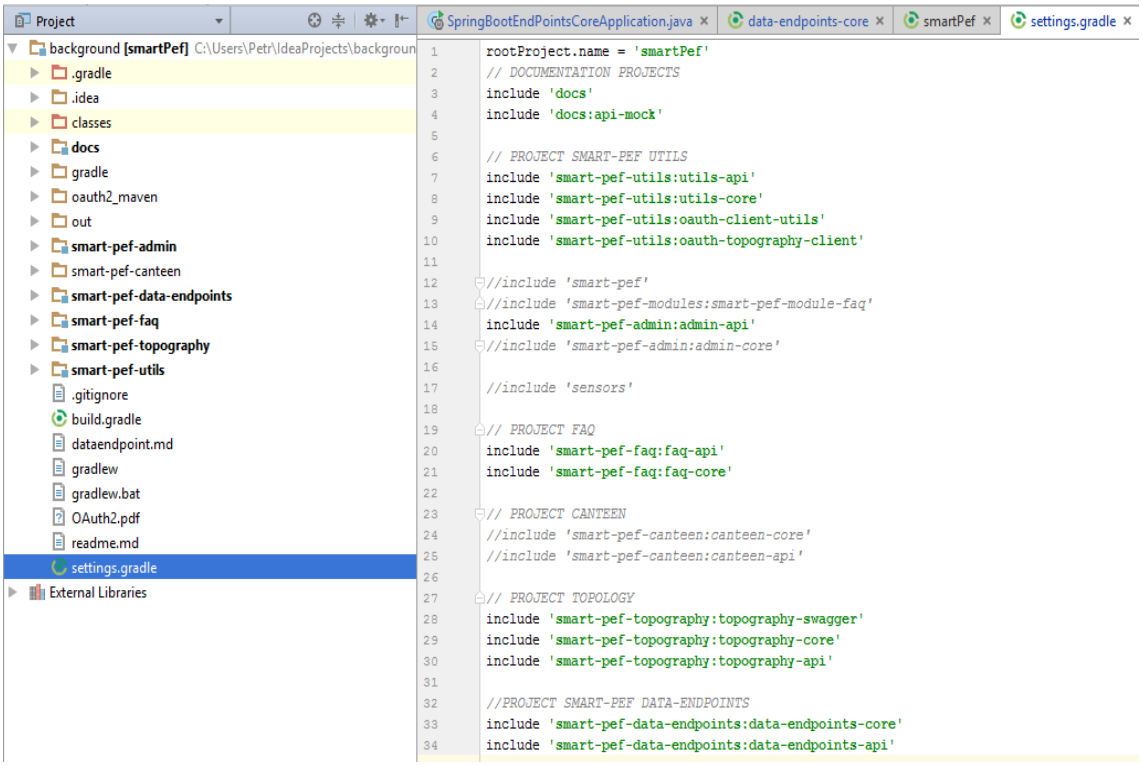
následně proběhne replikace databáze na třech strojích.

# 4 Vlastní práce

V následujících kapitolách jsou detailně zdokumentovány kroky zmíněné v metodice. Veškeré implementace vznikají ve vývojovém prostředí IntelliJ Idea.

## 4.1 Registrace projektu data-endpoints

Dle konvence projektu, se píše před každým názvem podprojektu prefix ”smart-pef-”. Tedy celý název podprojektu zní **smart-pef-data-endpoints**. Registraci zajistí klíčové slovo **include** s názvem modulu vepsaným do jednoduchých uvozovek. Před názvem je uveden název podprojektu oddělený dvojtečkou.



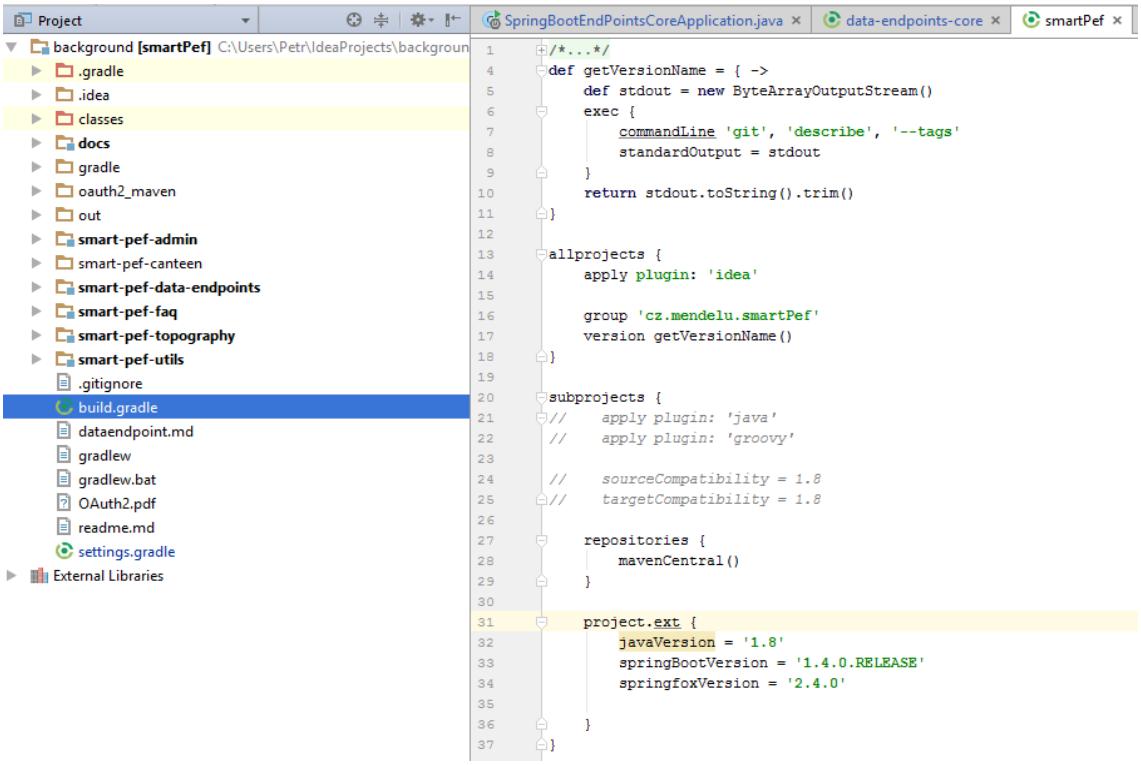

Obrázek 7: Přidání modulů do settings.gradle

Projekt data-endpoints obsahuje dva moduly viz. řádky 33 a 34. První modul je data-endpoints-core a druhý data-endpoints-api.

Gradle oba moduly po vytvoření naindexuje (zahrne do hlavního projektu).

## 4.2 Konfigurace pro build projektu

Kromě settings.gradle, hlavní projekt zahrnuje rodičovskou build konfiguraci, do níž se ukládá nastavení, které mohou podprojekty podědit.




Obrázek 8: Build konfigurace SMART faculty

Jak je možné vidět na obrázku, do proměnné getVersionName se ukládá verze projektu převzatá z verzovacího systému Git. Blok *allprojects* umožňuje sdílet vlastnosti v rámci kořenového projektu i všech podprojektů, oproti tomu blok *subprojects* sdílí vlastnosti pouze pro podprojekty. Z toho lze odvodit, že nad celým projektem SMART faculty se dědí verze, unikátní identifikátor projektu (group 'cz.mendelu.smartPef') a plugin idea. Ten umožňuje otevírat projekty ve vývojovém prostředí Idea a zároveň zahrne další utility poskytované Ideou, které usnadní práci s Gradle.

Pro podprojekty se nastavuje zdroj závislostí (knihoven) v bloku *repositories*. Závislosti se stahují z hlavního repozitáře Mavenu. V *project.ext* se definují vlastnosti ve tvaru **klíč = hodnota**, jako je tomu v souborech s příponou *.properties*. V našem případě jsou zde definovány verze Javy a Springu. Předpokladem pro rozložení vlastností v blocích *allprojects* a *subprojects* je nepřítomnost kódu v kořenovém projektu, tedy zdroj závislostí a verze knihoven nebo programovacího jazyka, stačí sdílet pro podprojekty.

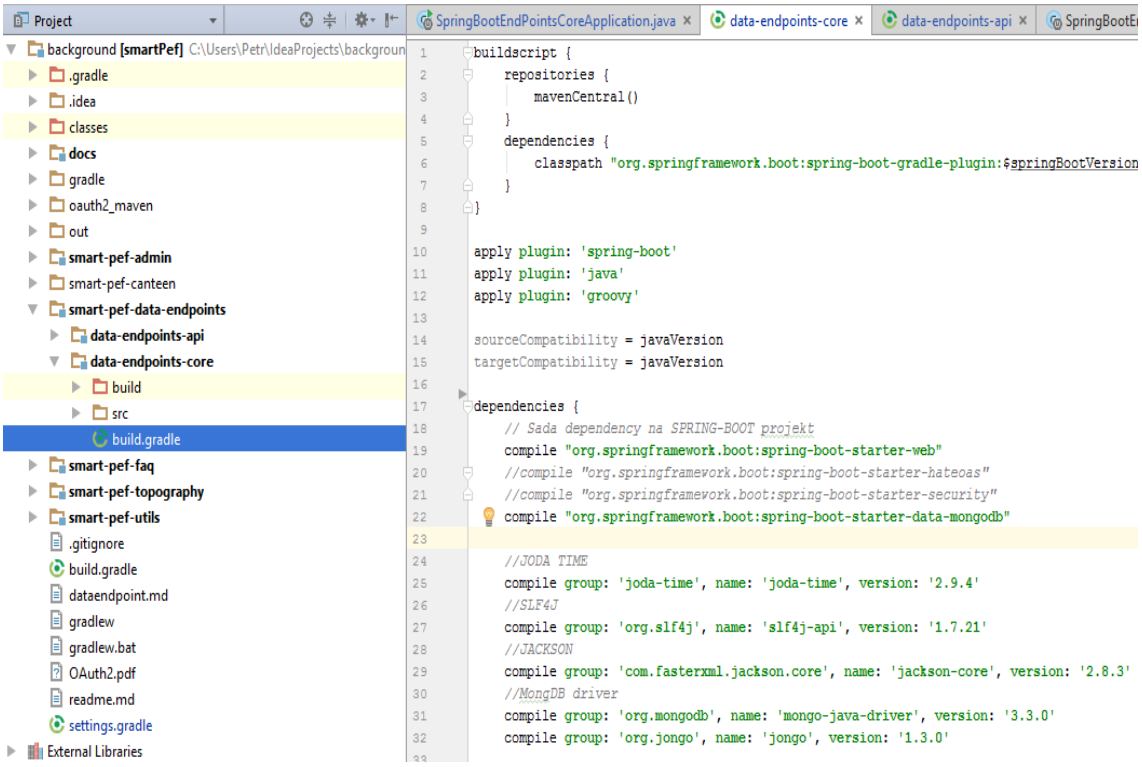
## 4.3 Konfigurace pro build modulů

Rozdělování aplikační logiky do modulů umožňuje vyvíjet jednotlivé moduly nezávisle na sobě a zároveň umožnit jejich znovupoužitelnost. Tím jak se liší jejich

funkcionalita, liší se částečně i build konfigurace.

### 4.3.1 Konfigurace data-endpoints-core

Účel tohoto modulu tkví ve zpracování dat, vytvoření přístupu do databáze a přes vrstvu service vystavit metody pro vstup z dalšího modulu.




Obrázek 9: Konfigurace modulu core

První blok v modulu je buildscript, do něhož se vkládají pluginy, tasky nebo další třídy, které slouží pouze pro vykonání build procesu, nikoliv pro běh programu.

Konkrétně na obrázku jde vidět, že knihovny jsou přidávány z hlavního repozitáře Mavenu (blok repositories) a v dalším bloku je závislost (dependencies) pro spring-boot-gradle-plugin. Ten poskytuje podporu Spring Boot aplikacím v Gradle. Stejně tak se používá tento plugin i pro náš program viz. apply plugin 'spring-boot'. Pod ním následuje plugin Java, který určuje, že bude vyvvíjen javovský projekt. Plugin Groovy umožňuje kombinovat v projektu jazyk Java a Groovy.

Konfigurace *sourceCompatibility* stanovuje verzi Javy a *targetCompatibility* slouží pro build, aby Gradle generoval javovské třídy ve verzi 1.8. Verze *javaVer-*

*sion* je podděděna z hlavní konfigurace. Poslední blok *dependencies* v konfiguraci importuje knihovny[1].

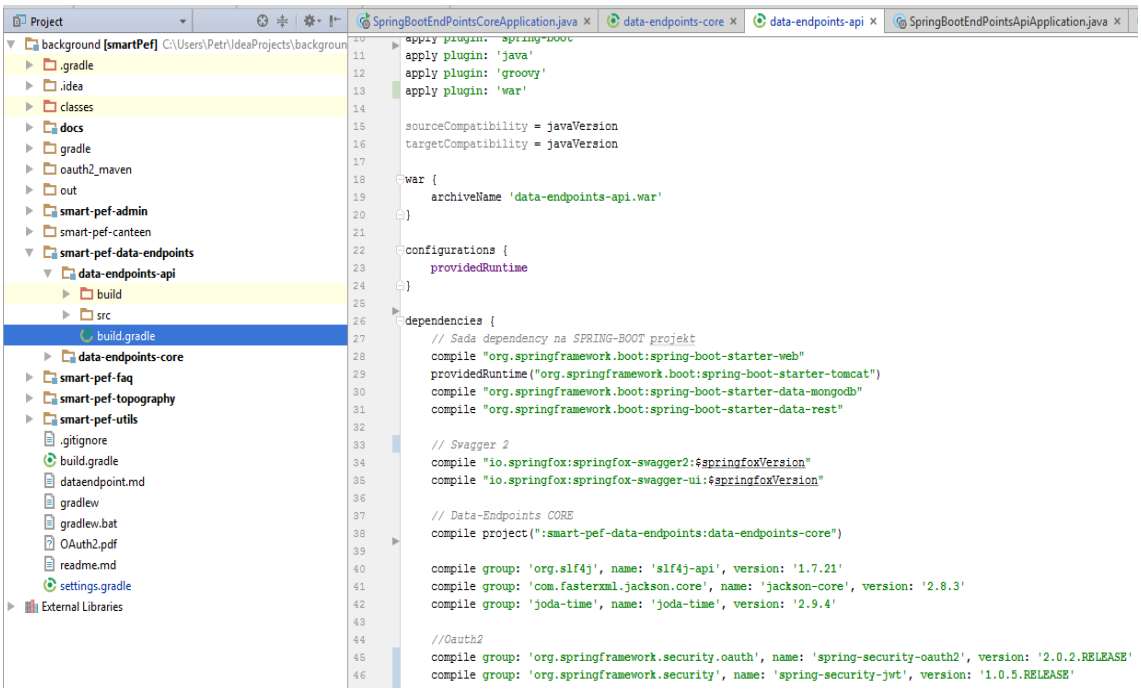
Stručně k jednotlivým knihovnám v modulu:

- ***org.springframework.boot*** - základ pro Spring Boot aplikaci,
  - ***spring-boot-starter-web*** - vývoj webových aplikací pomocí Spring MVC (v rámci REST),
  - ***spring-boot-starter-data-mongodb*** - využívání MongoDB ve Springu.
- ***joda-time*** - práce s časovými údaji (náhrada za defaultní JDK knihovnu pro práci s datumem a časem),
- ***org.slf4j*** - logování,
- ***com.fasterxml.jackson.core*** - serializace/deserializace JSON,
- ***com.mongodb a org.jongo*** - MongoDB drivery.

### 4.3.2 Konfigurace data-endpoints-api

Modul slouží jako rozhraní mezi HTTP požadavky a datovou vrstvou (modul core). Tyto požadavky mohou vznikat za účelem ukládání nebo selectování dat. Příjem a zpracování HTTP požadavků probíhá přes REST controllery, které přijímají požadavky na určitých URL adresách.

[1] Lze je manuálně zadávat po zkopírování z centrálního repozitáře Mavenu - https://mvnrepository.com/




Obrázek 10: Konfigurace modulu api

Na obrázku není znázorněn počátek konfigurace, protože se tam opakují některé bloky jako u modulu core.

Oproti modulu core zde je používán plugin war. Defaultně se aplikace builduje do souboru s příponou .jar (Java ARchive) a tento plugin to přenastaví na soubor .war (Web application ARchive). Blok *war* pak umožňuje dodatečně měnit vlastnosti webového archivu.

S každým pluginem se do projektu vkládá nová předdefinovaná konfigurace. Tu lze využít pomocí bloku *configurations*.

Java plugin obsahuje pro správu závislostí několik konfigurací, mezi něž patří i *compile* a *runtime*. Stručně řečeno, v případě compile jsou všechny artefakty dostupné jak při kompilaci, tak i běhu programu. Runtime dědí od compile a artefakty takto označené, budou dostupné jen při běhu. Plugin war přidává dvě další možnosti *providedCompile* a *providedRuntime*. Obě značí, že se nebudou takto označené závislosti vkládat do webového archivu.

Mezi závislostmi se nachází knihovny již popsané v kapitole Konfigurace data-endpoints-core, tedy popis těch, co jsou unikátní v tomto modulu:

- ***org.springframework.boot***
  - ***spring-boot-starter-tomcat*** - starter verze aplikačního serveru Tomcat[1],

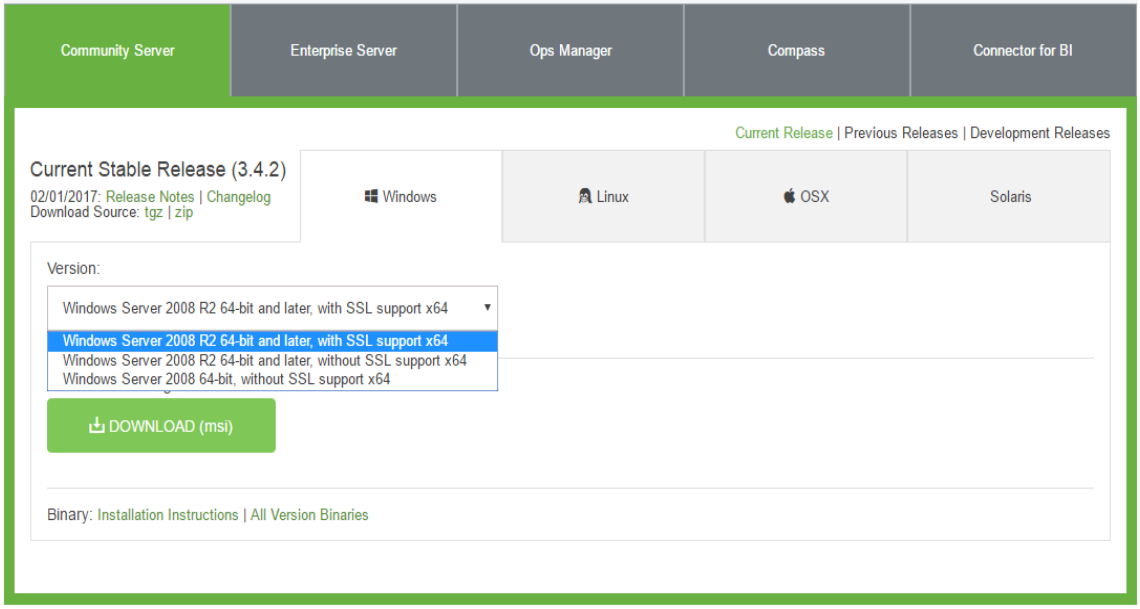
[1]S konfiguraci providedRuntime nebude zahrnut do webového archivu. Je to z důvodu, že se WAR nasazuje na plnou verzi Tomcatu.

  - ***spring-boot-starter-data-rest*** - přístup k repozitářům Spring Data přes REST,
- ***Swagger 2 API*** - generování dokumentace REST rozhraní,
- ***Modul core*** - přidání modulu jako knihovny,
- ***OAuth 2*** - autentizace uživatele.

## 4.4 Instalace a konfigurace MongoDB

MongoDB poskytuje Community a Enterprise verzi na oficiálních stránkách[1]. Enterprise slouží pro komerční účely a zahrnuje některé nastandartní funkce (in-memory úložiště, šifrování apod.).

Pro účely projektu postačí řešení Community. To má 3 verze:




Obrázek 11: Výběr verze instalace

Před výběrem by bylo vhodné se zamyslet nad tím, zda existují nějaká omezení pro nasazení MongoDB na serveru. V případě tohoto projektu nejsou žádné, čili po zhodnocení poskytovaných verzí je podpora SSL a Windows Server 2008 R2 (aktualizovaná verze Windows Server 2008) nejvíce příznivá, proto bude provedena instalace **Windows Server 2008 R2 64-bit and later, with SSL support x64**.

Po úpěšném provedení instalace musí být vytvořen, v kořenovém adresáři na disku, adresář data a v něm db. Absolutní cesta k adresáři *db* na běžném počítači s operačním systémem Windows vypadá pak takto:

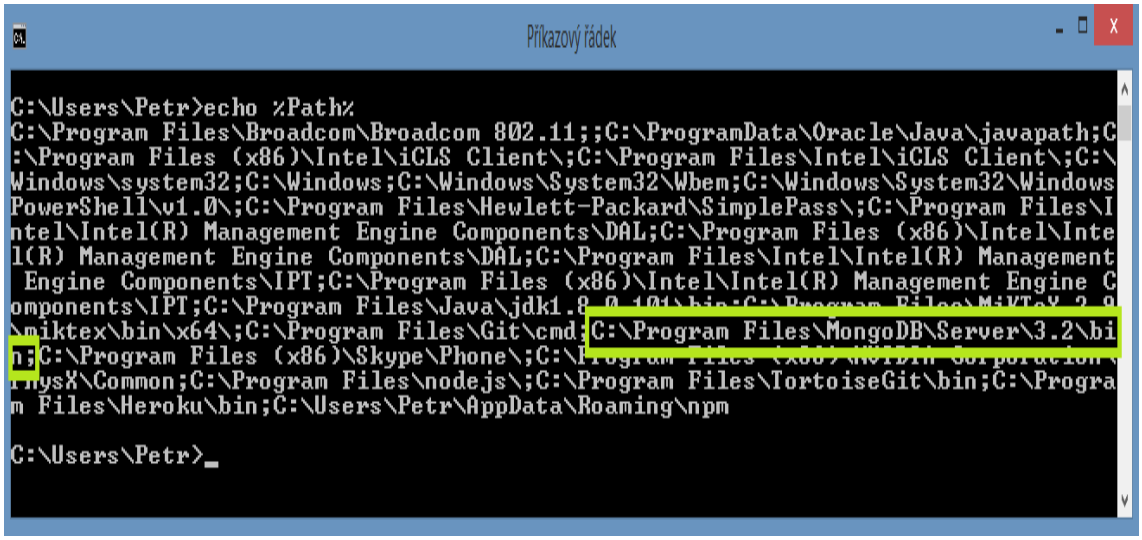
[1] https://www.mongodb.com/download-center#community

```
C:\data\db
```

Do adresáře umísťuje MongoDB veškeré kolekce dokumentů. V případě, že adresář chybí, proces startu databáze havaruje ihned na začátku a vyhodí chybovou hlášku popisující nenalezení zmíněných složek.

Pro usnadnění přístupu k utilitám Monga[1] z příkazové řádky, stačí přidat absolutní cestu k adresáři bin do proměnné prostředí **Path**.

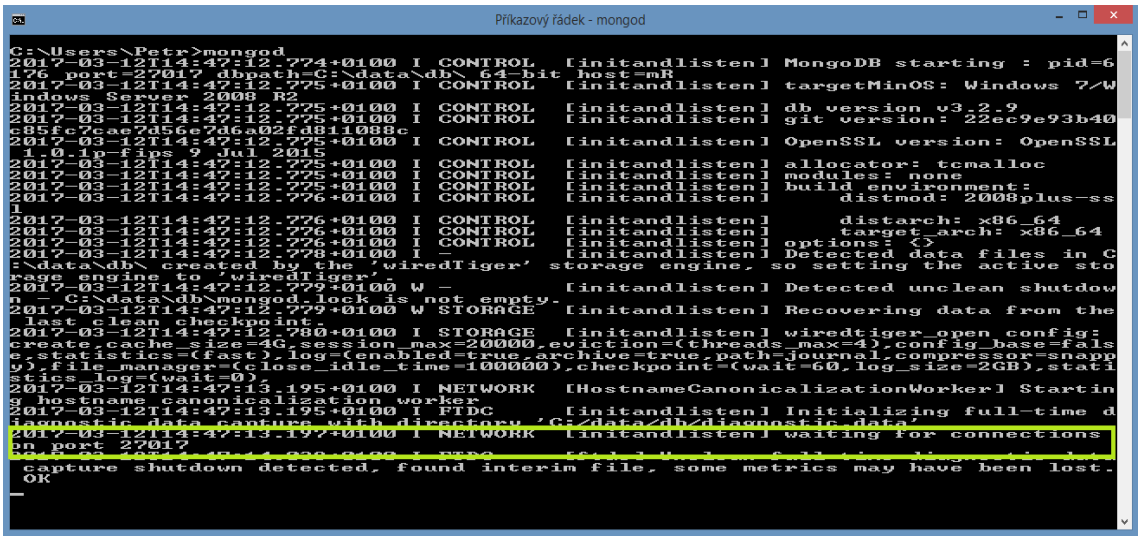
Proměnná prostředí by pak měla vypadat takto:




Obrázek 12: Cesta k adresáři MongoDB bin

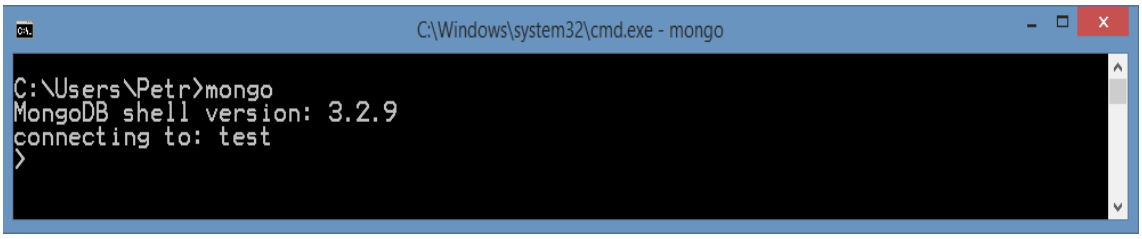
Pokud byly provedeny veškeré zmíněné kroky, lze příkazem **mongod** spustit Mongo na portu 27017.

[1] https://docs.mongodb.com/getting-started/shell/client/




Obrázek 13: Start databáze

Jakmile server běží, po otevření nového okna příkazové řádky (zásahem do původního okna by byl server zastaven), je možné přistoupit k mongo shellu přes příkaz **mongo**.



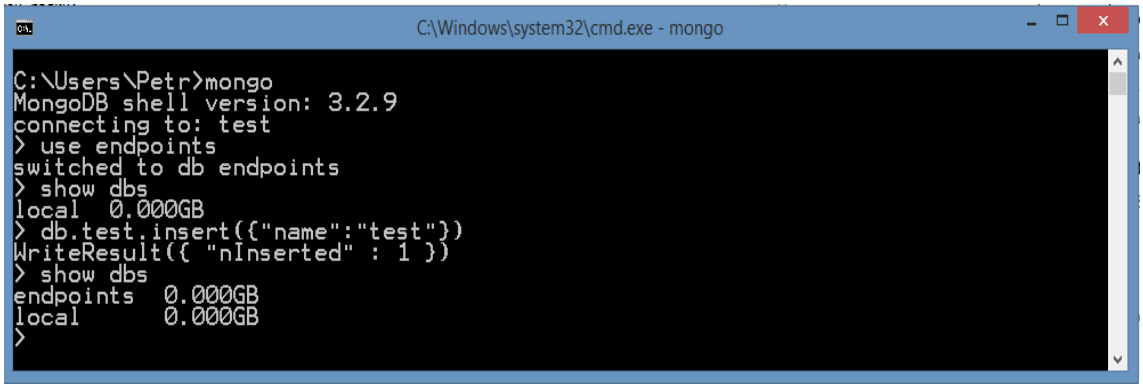

Obrázek 14: Přístup k mongo shellu

Dále se musí vytvořit databáze s názvem „endpoints“, do které budou proudit kolekce dokumentů přes REST API. Mongo shell nemá standartní příkaz, kterým by šlo přímo vytvořit databázi, proto se využívá tento postup:

1. Přepnutí do databáze, i když zatím neexistuje příkazem **use** NÁZEV_DATABÁZE,
2. Kontrola, zda existuje **show dbs**[1],
3. Vložení libovolné kolekce **db.název_kolekce.insert**(JSON),
4. Kontrola, že již existuje **show dbs**.

[1]Pokud nebyla dříve vytvořena, neměla by být vidět.

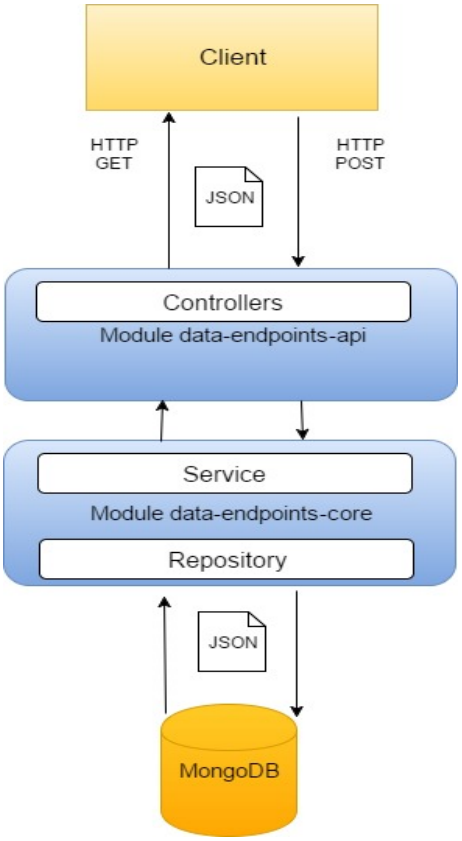
Postup aplikovaný přes příkazovou řádku:




Obrázek 15: Vytvoření databáze endpoints

## 4.5 Architektura REST API

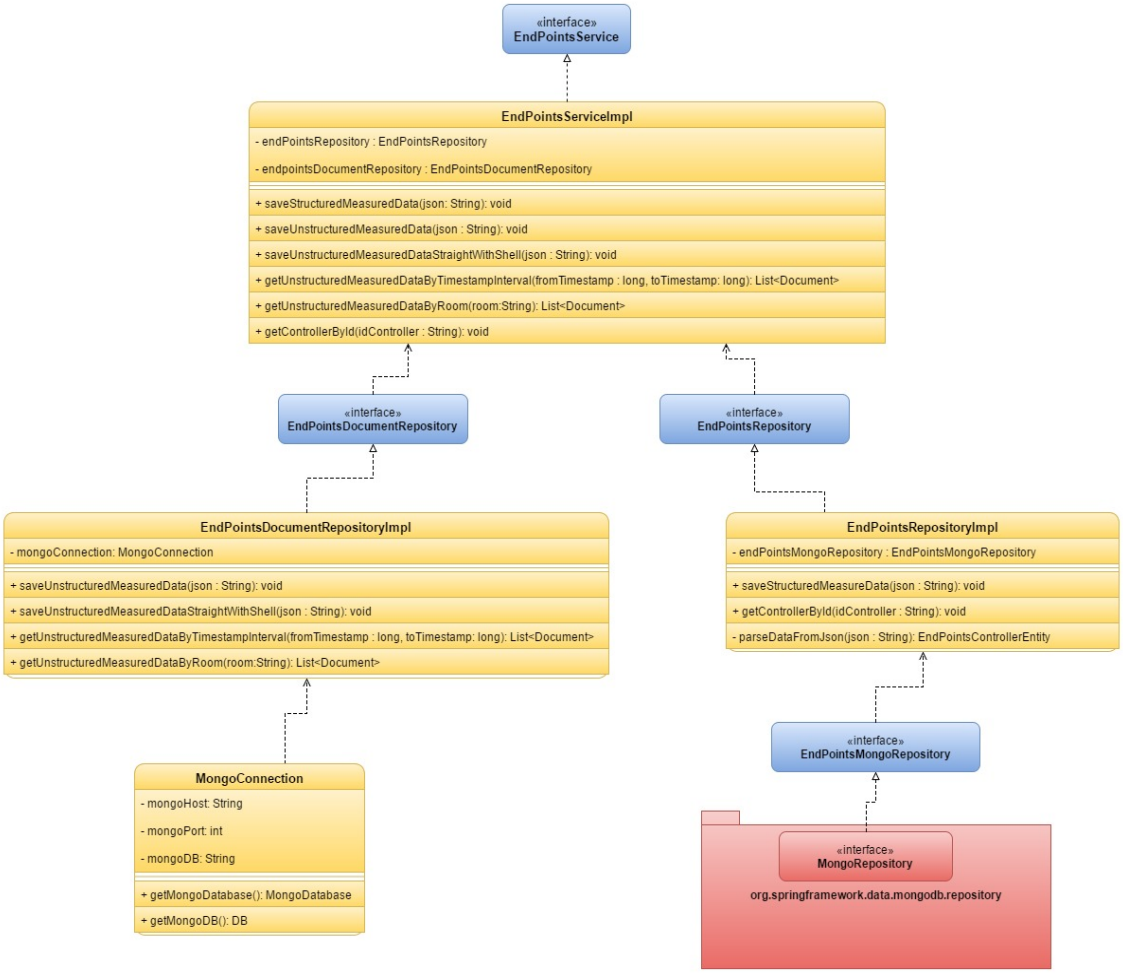
Architektura se skládá ze tří hlavních vrstev. První vrstva, na níž přichází požadavky ke zpracování, je sestavena z controllerů. Controllery implementují metody, které jsou mapovány na URL adresy. Po tom, co přijde požadavek na controller do odpovídající metody, zavolá se v ní metoda z vrstvy service. Standartně v ní bývá implementována business logika, v tomto případě se data pouze ukládají a selektují, takže vrstva data předá do repository. To je poslední vrstva, které zpracovává data a provádí dotazy nad databází.




Obrázek 16: Schéma architektury REST API

## 4.6 Implementace modulu data-endpoints-core

V návaznosti na kapitolu Instalace a konfigurace MongoDB se může přejít ke konfiguraci připojení do databáze na úrovni aplikace. Pak implementovat vrstvu repository a otestovat, zda běží komunikace mezi ní a databází.

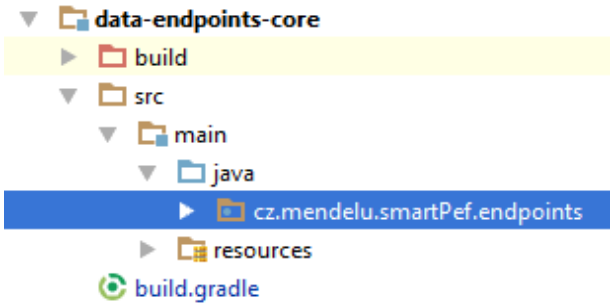



Obrázek 17: Diagram propojení komponent modulu core

Diagram nezahrnuje všechny třídy, pouze demonstruje pohled na hlavní části modulu, které zprostředkovávají datový tok.

### 4.6.1 Hlavní konfigurace Springu

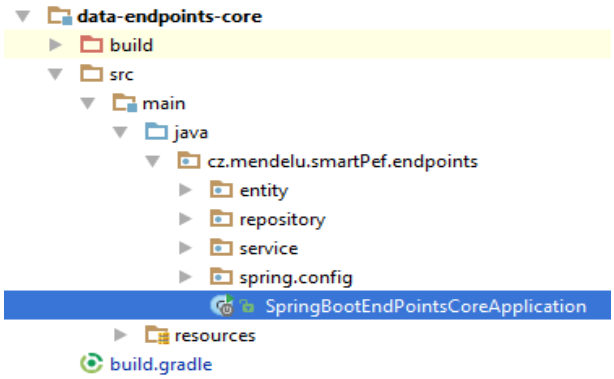
Obecně se zdrojové kódy v javovských projektech umisťují pod kořenový adresář do adresářů `src/main/java`. V rámci celého projektu SMART faculty je konvence umístění zdrojových kódů rozšířena dále do balíčků `cz/mendelu/smartPef/<zkraceny_nazev_podprojektu>`.




Obrázek 18: Umístění zdrojových kódů

Dříve se většina frameworků jako je Spring konfigurovala přes XML, dnes se již přechází na anotace.

Obvykle jako "best practise"[1] vývojáři opakovaně používali anotace @Configuration, @EnableAutoConfiguration a @ComponentScan dohromady za účelem konfigurace Springu. Z toho důvodu se tvůrci Springu rozhodli tyto anotace spojit do jedné - @SpringBootApplication. Ta patří nad deklaraci třídy, do kořenového balíčku (endpoints) a od tohoto balíčku bude Spring skenovat jednotlivé komponenty.




Obrázek 19: Hlavní konfigurace Springu

Třída hlavní konfigurace modulu core má název SpringBootEndPointsCoreApplication. Obsahuje anotaci @EnableMongoRepositories s parametrem *basePackage*, do něhož se zapisují názvy balíčků v podobě pole, v nichž lze nalézt Mongo repozitáře. Nicméně kdyby nebyl tento parametr zadán, Spring hledá repozitáře od kořenového balíčku níže. Přesto z hlediska přehlednosti je vhodné tento parametr zadat.

[1] http://docs.spring.io/autorepo/docs/spring-boot/current/reference/html/using-boot-structuring-your-code.html

```java
package cz.mendelu.smartPef.endpoints;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;

/**
 * Created by Petr on 16. 9. 2016.
 */
@EnableMongoRepositories({"cz.mendelu.smartPef.endpoints.repository"})
@SpringBootApplication
public class SpringBootEndPointsCoreApplication {


    public static void main(String[] args){
        SpringApplication.run(SpringBootEndPointsCoreApplication.class, args);
    }

}
```

Obrázek 20: Zdrojový kód třídy SpringBootEndPointsCoreApplication

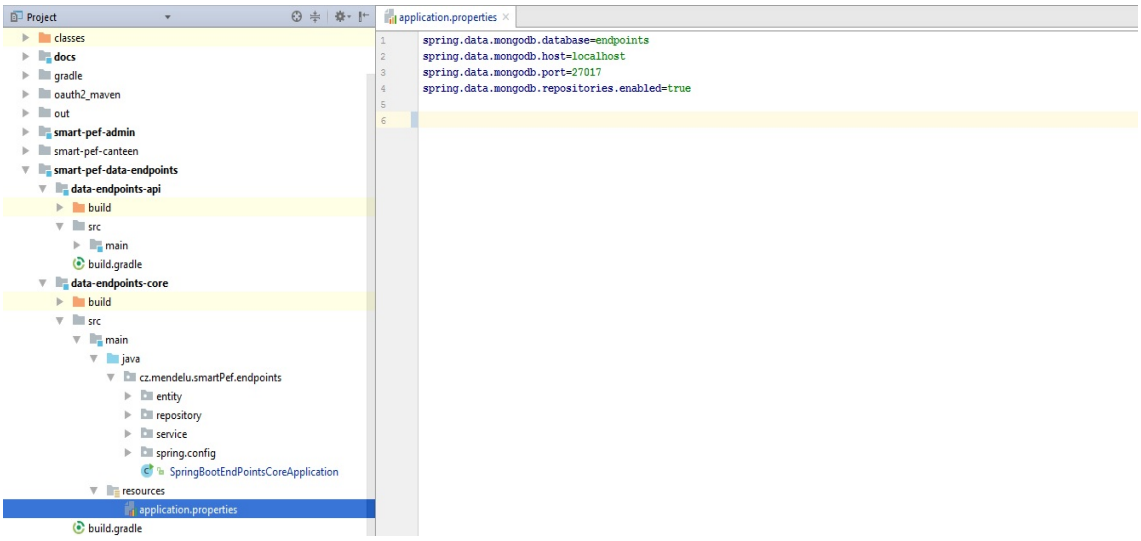
Start aplikace implementuje *main* metoda, v níž se volá přes statickou třídu SpringApplication metoda *run*, která inicializuje Spring framework. Metoda run má dva parametry, první udává hlavní Spring komponentu a druhý slouží pro vložení argumentů z příkazové řádky.

### 4.6.2 Připojení MongoDB

Spring byl nakonfigurován a následně bude připojena databáze. Pro zobecnění konfigurací například databázového připojení, logování nebo rozdělení vývojových prostředí apod.[1] je lze vepisovat do souborů s příponou *.properties*. Tím se docílí, že hodnoty konfigurace budou dostupné na jednom místě, k němuž bude z aplikace přistupováno. Pokud dojde ke změně hodnot klíčů, změny se projeví všude, kde byly hodnoty použity. Tuto úlohu defaultně zajišťuje v adresáři `src/main/resources` soubor *application.properties*. Spring soubor po vytvoření automaticky využívá, jelikož je uveden v jeho konfiguraci. Název i umístění souboru může být změněno[2].

[1] Jedná se o hodnoty, které nejsou příliš proměnlivé a platí většinou nad celým projektem (modulem).

[2] https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html




Obrázek 21: Externí konfigurace modulu core

Struktura zápisu do souboru s příponou *.properties* je ve tvaru **klíč=hodnota**.

K tomu, aby připojení bylo úspěšné, postačí vyplnit jen určité parametry, ačkoliv jich Spring poskytuje více[1]. Konfigurace MongoDB není nijak odlišná od jiných databázových systémů, běžně se musí zadat **host**, **port** a **název databáze**. Ostatní parametry už závisí na nastavení úložiště. Například pokud se z hlediska bezpečnosti požaduje přihlášení uživatele, je nutné přidat do souboru heslo a jméno.

Jestliže není v konfiguraci MongoDB specifikován port, Spring nastaví defaultní 27017. Údaje o připojení není nutné zapisovat jako na obrázku, ale mohou být vepsány do parametru *spring.data.mongodb.uri* jako URI. Hodnota klíče by podle standardu vypadala takto:

```
mongodb://<uzivatelske_jmeno>:<heslo>@<host>:<port>/<nazev_databaze>
```

Blíže k naší konfiguraci viz. obrázek, první je název databáze endpoints, která byla vytvořena v kapitole Instalace a konfigurace MongoDB. Dále server běží na lokálním počítači, tedy host má hodnotu **localhost** a port 27017. Poslední klíč, není pro připojení důležitý, ale udává povolení k použití Spring Mongo repository.

### 4.6.3 Implementace komponenty MongoConnection

Dále dojde k první implementaci, a to komponenty nazvané **MongoConnection**, v níž se využívá právě hodnot z konfiguračního souboru. Je to běžná javovská třída anotovaná @Component, má tři atributy host, port a název databáze. Atributy se anotují @Value, což zajistí, že při incializaci objektu dojde k načtení hodnot z

[1] https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

konfiguračního souboru podle klíče v parametru anotace. Klíč je vkládán jako řetězec ve tvaru "${nazev.klice}".

Jednotlivé atributy se využijí k vytvoření instance databáze. Tu vrací metody *getMongoDatabase()* a *getMongoDB()*. Obě metody provádí totéž, nicméně vrací odlišné objekty. Třída DB má označení deprecated[1], avšak stále se používá pro inicializaci Jongo driveru. MongoDatabase nahrazuje třídu DB od verze driveru 3.0.

```java
package cz.mendelu.smartPef.endpoints.spring.config;

import com.mongodb.*;
import com.mongodb.client.MongoDatabase;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

/**
 * Created by xsadovsk on 11.11.2016.
 */

@Component
public class MongoConnection {

    @Value("${spring.data.mongodb.host}")
    private String mongoHost;

    @Value("${spring.data.mongodb.port}")
    private int mongoPort;

    @Value("${spring.data.mongodb.database}")
    private String mongoDB;

    public MongoDatabase getMongoDatabase() {
        MongoClient client = new MongoClient(mongoHost, mongoPort);
        return  client.getDatabase(mongoDB);
    }

    //Jongo driver still use deprecated DB.class
    public DB getMongoDB() {
        MongoClient mongoClient = new MongoClient(mongoHost, mongoPort);
        return  mongoClient.getDB(mongoDB);
    }
}
```

Obrázek 22: Komponenta MongoConnection

Tuto komponenta inicializuje Spring a přes anotaci @Autowired ji bude možné vkládat na různá místa v aplikaci.

### 4.6.4 Zpracování dat

Aplikace má již přístup do databáze, nicméně chybí implementace zajišťující zpracovávání dat. Způsobů jak je zpracovat, může být hned několik a diplomová práce se zabývá třemi vybranými.

Struktura příchozích dat je předem dohodnuta, a tedy není kladen důraz na bezpečnost, ale především na výkon při zpracování oběma směry z/do databáze.

[1] V následujících verzích driveru se nepoužívá.

### Zpracování dat pomocí entit

Tento způsob je z hlediska implementace nejnáročnější, avšak na druhou stranu poskytuje větší kontrolu nad daty než ostatní[1]. K parsování se využívá knihovny Jackson - FasterXML[2].

```
{
      "idController": "Q_12_2P",
      "timeMeasurement": "1490994851",
      "room": "Q_28",
      "typeJsonPackage": "1",
      "sensors": [{
            "idSensor": "41",
            "type": "temperature XY",
            "temperature": "35",
            "pressure": "57"
      }, {
            "idSensor": "42",
            "type": "temperature XH",
            "temperature": "24",
            "pressure": "48"
      }]
}
```

Obrázek 23: Formát ukládaného dokumentu

Hodnoty atributů JSON objektu viz. obrázek jsou pouze orientační. Především tento formát slouží jako předloha k implementaci jednotlivých tříd (entit), do nichž se data parsují. Objekt představuje fyzické zařízení nazývané controller, na němž jsou umístěny senzory měřící různé fyzikální veličiny.

Kromě identifikátoru controlleru, se do objektu přidávají popisné parametry, a to čas měření (timeMeasurement), místnost (room) a typ JSON (typeJsonPackage). Objekt dále obsahuje JSON pole s jednotlivými objekty reprezentujícími senzory, v jejichž atributech jsou hodnoty naměřených veličin.

Ze struktury objektu lze odvodit, že bude nutné implementovat dvě entity, jednu pro controller a druhou pro senzor. Obě se umístí do balíčku *entity*.

[1] Všeobecně MongoDB uloží jakýkoliv JSON dokument a jediná možnost jak ošetřit vstup je přes aplikaci.

[2] Označována za nejrychlejší knihovnu pro serializaci a deserializaci JSON - https://github.com/FasterXML/jackson-core/wiki

```java
package cz.mendelu.smartPef.endpoints.entity;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import java.util.List;

/**
 * Created by Petr on 17. 9. 2016.
 */

@Document(collection = "measuredDataController")
public class EndPointsDataControllerEntity {

    //@Id
    private String idController;
    private String timeMeasurement;
    private String room;
    private String typeJsonPackage;
    private List<EndPointsSensorEntity> sensors;
```

Obrázek 24: Entita controlleru

Pomocí anotace @Document se namapuje třída na entitu. Zároveň s ní lze explicitně přejmenovat název kolekce dokumentů. MongoDB totiž ukládá dokumenty do kolekcí, kde každému dokumentu kolekce vygeneruje identifikátor ve tvaru:


Obrázek 25: Vzorec pro generování identifikátoru MongoDB, (Chodúr, 2016)

Jednotlivé části identifikátoru v pořádí dle obrázku znamenají:

1. timestamp v sekundách,
2. identifikátor stroje,
3. identifikátor procesu,
4. počítadlo s náhodnou počáteční hodnotou.

Tento identifikátor může být přepsán hodnotou atributu pomocí anotace @Id, umístěnou nad atribut třídy. V entitě controlleru se nachází jako atribut seznam entit senzorů, což reprezentuje zmíněné JSON pole.

```
package cz.mendelu.smartPef.endpoints.entity;

/**
 * Created by Petr on 17. 9. 2016.
 */
public class EndPointsSensorEntity {

    private String idSensor;
    private String type;
    private String temperature;
    private String pressure;
```

Obrázek 26: Entita senzoru

Ke každému atributu u obou entit, jsou vygenerovány **get** a **set** metody.

Je nutné podotknout, že v okamžiku ukládání dat, pokud nemá vstupní JSON tvar odpovídající struktuře objektů, nedojde k uložení do databáze a zaloguje se výjimka popisující první neznámý parametr v JSON objektu.

### Zpracování dat přes Java driver

Pro práci s Mongem byl zároveň vyvinut Java driver[1]. V jeho implementaci se nachází balíček *org.bson* s objektem Document, jenž slouží jako entita. Chová se jako datová struktura Javy - Mapa, tedy záznamy mají tvar - **(klíč,hodnota)**. Vkládaná data lze metodou *parse()* deserializovat do objektu. Tato metoda má jeden vstupní parametr, a to JSON řetězec, který může mít libovolnou strukturu.

Oproti předešlému způsobu mapování, tady nedochází k žádné kontrole vstupních dat při ukládání a může být uložen jakýkoliv JSON.

### Zpracování dat přes Jongo driver

Třetí způsob využívá Jongo driver[2]. Jongo driver lze označit jako nádstavbu MongoDB Java driveru, jelikož spojuje funkcionalitu Java driveru a MongoDB shellu. Při zpracování dat stačí využít instanci třídy Jongo a nádstavby DBCollection (pochází z Java driveru) - MongoCollection. Není přímo volána metoda na parsování, nicméně implicitně uvnitř Jonga se data serializují/deserializují pomocí knihovny Jackson. Stejně jako u předešlého způsobu ukládá libovolný JSON.

## 4.6.5 Implementace vrstvy repository

Obecně se tato vrstva nachází na hranici mezi vrstvou service a přístupem do databáze. Účelem je převzít hrubá data a uložit do databáze, nebo naopak směrem ven z databáze data distribuovat.

[1] https://mongodb.github.io/mongo-java-driver/
[2] http://jongo.org

Byl vytvořen balíček *repository*, do něhož se umisťují jednotlivé rozhraní. Kromě rozhraní se v něm nachází balíček *impl*, jenž obsahuje implementace těchto rozhraní.

### Zpracování dat pomocí entit

Entity jsou připraveny, dále dojde k vytvoření rozhraní a jeho implementaci.

#### *Rozhraní*

Rozhraní se jmenuje *EndPointsRepository*, kde EndPoints je odvozeno od názvu podprojektu a Repository souvisí s konvencí vývoje na vrstvě repository.

```
package cz.mendelu.smartPef.endpoints.repository;

import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;
/**
 * Created by Petr on 14. 9. 2016.
 */
public interface EndPointsRepository {

    void saveStructuredMeasuredData(String json);

    EndPointsDataControllerEntity getControllerById(String idController);
}
```

Obrázek 27: Rozhraní EndPointsRepository

V rozhraní před signatury metod nemusí být vepsán modifikátor viditelnosti, jelikož všechny jsou **public**[1]. Metoda pro ukládání dat se nazývá *saveStructuredMeasuredData()*. Nevrací žádnou hodnotu (návratový typ void), ale má jeden vstupní parametr JSON řetězec. Druhá metoda nazvaná *getControllerById()* selectuje data podle identifikátoru a vrací objekt - entitu controlleru.

Implementace tohoto rozhraní používá pro přístup do databáze springovské *MongoRepository*. Jeho vlastnosti lze jednoduše podědit. Potomek třídy pak získá přístup k metodám poskytujícím základní operace nad databází. Pokud však jsou požadavky na dotazy specifičtější, mohou se přidat nové metody anotované @Query[2].

[1] Idea automaticky modifikátory podtrhává z důvodu nadbytečnosti.

[2] http://docs.spring.io/spring-data/mongodb/docs/1.2.0.RELEASE/reference/html/mongo.repositories.html

```java
package cz.mendelu.smartPef.endpoints.repository;

import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

/**
 * Created by Petr on 14. 9. 2016.
 */
@Repository
public interface EndPointsMongoRepository extends MongoRepository<EndPointsDataControllerEntity, String> {
}
```

Obrázek 28: Rozhraní EndPointsMongoRepository

Při dědění musí být vyplněny generické typy, první je entita a druhý odpovídá datovému typu identifikátoru entity. Nemá implementaci, tudíž se nad rozhraní umisťuje anotace @Repository, i přesto že obvykle bývá anotace až nad implementující třídou. Jak bylo zmíněno, musí být přidána anotace @EnableMongoRepositories do hlavní konfigurace, aby mohl Spring detekovat tuto komponentu.

### *Implementace rozhraní*

Třída implementující rozhraní má název stejný jako rozhraní (pokud není implementací více), ale z hlediska konvence a odlišení je na konec názvu připisována přípona *Impl*. Nad deklaraci třídy patří anotace @Repository. Kvůli abstrahování přístupu k objektům vkladaným jako závislosti, se injektují rozhraní, nikoliv jejich implementace. Rozhraní totiž může mít implementací hned několik a Spring je schopen přes proxy[1] dosadit odpovídající implementaci.

[1] https://spring.io/blog/2012/05/23/transactions-caching-and-aop-understanding-proxy-usage-in-spring

```java
package cz.mendelu.smartPef.endpoints.repository.impl;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;

import cz.mendelu.smartPef.endpoints.repository.EndPointsMongoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import cz.mendelu.smartPef.endpoints.repository.EndPointsRepository;

import java.io.IOException;
import java.sql.Timestamp;

/**
 * Created by Petr on 14. 9. 2016.
 */
@Repository
public class EndPointsRepositoryImpl implements EndPointsRepository{

    private static final Logger LOGGER = LoggerFactory.getLogger(EndPointsRepositoryImpl.class);

    @Autowired
    private EndPointsMongoRepository endPointsMongoRepository;
```

Obrázek 29: Deklarace třídy EndPointsRepositoryImpl

Z hlediska údržby, ladění, odstraňování chyb apod. je považováno za nesmírně důležité logování určitých klíčových stavů aplikace, proto se nejen zde vkládá jako statický atribut třídy *LOGGER* z knihovny slf4j[1]. Dalším atributem je zmiňované EndPointsMongoRepository.

Při implementaci rozhraní se musí implementovat všechny jeho metody. Většina dnešních vývojových prostředí, ihned po zapsání slova **implements** a názvu rozhraní, na to upozorňuje.

```java
    @Override
    public void saveStructuredMeasuredData(String json){
        LOGGER.info("Data saved to database by first way: \n" + json);
        EndPointsDataControllerEntity controller = parseDataFromJson(json);
        endPointsMongoRepository.save(controller);
    }
```

Obrázek 30: Implementace metody saveStructuredMeasuredData

V implementaci metody *saveStructuredMeasuredData* se pouze zalogují ukládaná data a pomocí privátní metody *parseDataFromJson()* jsou deserializována do objektu entity. Po provedení deserializace je nad instancí EndPointsMongoRepository volána metoda *save()* pro uložení.

[1] https://www.slf4j.org/

```
private EndPointsDataControllerEntity parseDataFromJson(String json) {

    ObjectMapper mapper = new ObjectMapper();
    EndPointsDataControllerEntity controller = null;
    try {
        controller = mapper.readValue(json, EndPointsDataControllerEntity.class);
    } catch(JsonParseException ex){
        LOGGER.warn(" JsonParseException with message :" + ex.getMessage());
    } catch(JsonMappingException ex){
        LOGGER.warn("JsonMappingException with message :" + ex.getMessage());
    } catch(IOException ex){
        LOGGER.warn("IOException with message :" + ex.getMessage());
    }
    LOGGER.info("Id controller: " + controller.getIdController());
    return controller;
}
```

Obrázek 31: Implementace privátní metody parseDataFromJson

Vytvoří se instance objektu *ObjectMapper*, jenž pochází ze zmiňované knihovny Jackson a zajistí deserializaci. Pro kontrolu může být zalogována hodnota identifikátoru z právě vytvořené instance entity.

Tento proces by se měl obalit blokem **try-catch**, jelikož metoda *readValue()* vyvolává tři různé výjimky[1].

```
@Override
public EndPointsDataControllerEntity getControllerById(String idController) {
    EndPointsDataControllerEntity controller = endPointsMongoRepository.findOne(idController);
    LOGGER.info("Controller with id " + controller.getIdController() + " was returned.");
    return controller;
}
```

Obrázek 32: Implementace metody getControllerById

Druhá metoda rozhraní *getControllerById* má vstupní parametr identifikátor controlleru, který se vloží do metody *findOne*. Navrací data z databáze v podobě entity, s hodnotami odpovídajícího dokumentu. Na závěr by se měla zalogovat hodnota nějakého atributu entity, aby byla jistota, že data existují.

### Zpracování dat pomocí Java driveru

Metody tohoto i následujícího zpracování jsou umístěny v jednom rozhraní. I když každé má svoji metodu k ukládání, pro selectování využívájí stejné.

#### *Rozhraní*

Rozhraní má název *EndPointsDocumentRepository*, jenž aplikuje stejnou konvenci

[1]Kvůli přehlednosti a lepšímu ladění, by se měla každá odchytit zvlášť.

jako u předešlého způsobu, avšak pro rozlišení bylo doplněno do názvu slovo Document, jelikož se toto rozhraní při implementaci opakovaně používá.

```
package cz.mendelu.smartPef.endpoints.repository;

import org.bson.Document;

import java.util.List;

/**
 * Created by Petr on 30. 9. 2016.
 */
public interface EndPointsDocumentRepository {

    void saveUnstructuredMeasuredData(String json);

    List<Document> getUnstructuredMeasuredDataByRoom(String room);

    List<Document> getUnstructuredMeasuredDataByTimestampInterval(long fromTimestamp, long toTimestamp);

}
```

Obrázek 33: Rozhraní EndPointsDocumentRepository

První metoda *saveUnstructuredMeasuredData* ukládá data a zbylé dvě slouží pro selectování.

Metoda *getUnstructuredMeasuredDataByRoom* má jediný parametr, a to řetězec představující označení místnosti. Jedná se o místnost, z které pochází naměřená data (označení místností na fakultě např. Q_28).

Druhá metoda vyžaduje zadat dva parametry, oba jsou typu **long**, čili přijímají číselné hodnoty. Tyto hodnoty mají tvar Unix Timestamp v sekundách, takže účel metody představuje získání dat z určitého časového intervalu. První parametr udává spodní a druhý horní hranici. Obě metody získávají data jako seznam objektů.

### *Implementace rozhraní*

Jedná se tedy o třídu *EndPointsDocumentRepositoryImpl*, v níž musí být implementovány zmíněné tři metody.

```java
package cz.mendelu.smartPef.endpoints.repository.impl;

import com.google.common.collect.Lists;
import com.mongodb.*;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import cz.mendelu.smartPef.endpoints.repository.EndPointsDocumentRepository;
import cz.mendelu.smartPef.endpoints.spring.config.MongoConnection;
import org.bson.Document;
import org.jongo.Jongo;
import org.jongo.MongoCollection;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import java.sql.Timestamp;
import java.util.List;

import static com.mongodb.client.model.Filters.and;
import static com.mongodb.client.model.Filters.gt;
import static com.mongodb.client.model.Filters.lte;

/**
 * Created by Petr on 30. 9. 2016.
 */

@Repository
public class EndPointsDocumentRepositoryImpl implements EndPointsDocumentRepository {

    private static final Logger LOGGER = (Logger) LoggerFactory.getLogger(EndPointsDocumentRepositoryImpl.class);
    private static final String UNSTRUCTURED_DATA = "unstructuredData";
    private static final String MAINTENANCE_DATA = "maintenanceData";
    private static final String MEASURED_DATA = "measuredData";
    private static final String TIMESTAMP_MEASURE = "timeMeasurement";

    @Autowired
    private MongoConnection mongoConnection;
```

Obrázek 34: Implementace rozhraní EndPointsDocumentRepository

Mezi atributy třídy se nachází konstanty loggeru, názvy databázových kolekcí a název parametru času měření. Jsou zavedeny z důvodu opakovaných výskytů řetězců jež zastupují. Pomocí nich dochází k lepší čitelnosti kódu a zároveň, pokud dojde ke změně, lze ji provést na jednom místě.

Znovu musí být implementace anotována @Repository. Kromě konstant, třída také obsahuje jako atribut komponentu MongoConnection, která byla nainjektována za účelem vytvoření instance databáze.

```java
    @Override
    public void saveUnstructuredMeasuredData(String json) {
        LOGGER.info("Data saved to database by second way: \n" + json);
        com.mongodb.client.MongoCollection<Document> collection = getMongoClient().getCollection(UNSTRUCTURED_DATA);
        Document objectFromJson = Document.parse(json);

        objectFromJson.put(TIMESTAMP_MEASURE, createTimeMeasureAsTimestamp(objectFromJson));
        collection.insertOne(objectFromJson);
    }
```

Obrázek 35: Implementace metody saveUnstructuredMeasuredData

Pro přehled o proudících datech se zaloguje ukládáný řetězec. Kvůli opakovanému volání instance databáze, zajistí lepší čitelnost kódu, když bude vytváření

umístěno do privátních metod.

```
private DB getMongoDB(){
    return mongoConnection.getMongoDB();
}

private MongoDatabase getMongoClient(){
    return mongoConnection.getMongoDatabase();
}
```

Obrázek 36: Metody pro přístup do databáze

Přes instanci lze přistoupit k libovolné kolekci dokumentů, tedy probíhá inicializace objektu *MongoCollection*. Pokud kolekci při ukládání nenajde, bude vytvořena. Na obrázku Implementace metody saveUnstructuredMeasuredData je před názvem třídy prefix **com.mongodb.client**[1], z důvodu duplicitních názvů tříd. Jongo i Java driver implementují vlastní třídu s názvem MongoCollection.

Vstupní JSON je deserializován do objektu Document metodou *parse()*, přesto ještě před uložením, musí dojít k modifikaci parametru s časem měření. V tomto stavu by se uložil jako řetězec nebo číslo a nebylo by možné zprostředkovat dotazy, k získání dat naměřených v zadaném časovém intervalu.

K tomu poslouží privátní metoda *createTimeMeasureAsTimestamp()*, která ze vstupní hodnoty vytvoří instanci třídy Timestamp, s níž dokáže databáze pracovat.

```
private Timestamp createTimeMeasureAsTimestamp(Document document){

    String timeMeasure;
    long timeMeasureLong;
    int timeMeasureInteger;
    try {
        timeMeasure = (String) document.get(TIMESTAMP_MEASURE);
        String timeMeasureConvertToMilliseconds = timeMeasure + "000";
        timeMeasureLong = convertStringToLong(timeMeasureConvertToMilliseconds);
    }catch(ClassCastException ex){
        LOGGER.warn("Field " + TIMESTAMP_MEASURE + " is not of type String.");
        timeMeasureInteger = (int) document.get(TIMESTAMP_MEASURE);
        timeMeasureLong = (long) timeMeasureInteger*1000;
    }
    return new Timestamp(timeMeasureLong);
}
```

Obrázek 37: Metoda konvertující řetězce a čísla na Timestamp

Logika metody musí předvídat existenci hodnoty času měření jako čísla nebo řetězce. V první fázi testování se čas zpracovával pouze jako řetězec, nicméně, jakmile byl nahrazen číslem, došlo k vyhození výjimky **ClassCastException**. Výjimka znamená, že byl očekáván jiný datový typ hodnoty, než ten co byl vložen, tudíž nedošlo k přetypování.

[1]Balíčky, do nichž je zanořena třída.

Řešení zahrnuje vnoření kódu do try/catch bloku. Poté v bloku try zpracovat řetězec a pokud dojde k vyhození výjimky, lze ji odchytit a předpokládat, že příchozí hodnota je číslo.

V prvním kroku zpracování se získá čas měření podle názvu parametru a uloží do proměnné.

Konstruktor Timestampu očekává čas v řádech milisekund a datový typ long, avšak controller posílá čas v sekundách. Z toho důvodu musí být přidány do řetězce na konec tři nuly, nebo po přetypování na long číslo vynásobit tisícem.

V metodě je použitá jednoduchá privátní metoda *convertStringToLong()*, jak název napovídá, provádí konverzi mezi String a long. Tělo metody obsahuje pouze jeden řádek kódu(return Long.parseLong(value);), kde se navrací již přetypovaný řetězec.

Následně stačí dokument uložit metodou *insertOne()*.

Zbývá implementace metod přenášejících data opačným směrem - ven z databáze. Jsou univerzální a mohou vracet data uložená oběma drivery.

První z implementovaných metod navrací uložená data podle parametru s názvem místnosti, kde se nachází controller.

```
@Override
public List<Document> getUnstructuredMeasuredDataByRoom(String room){

    com.mongodb.client.MongoCollection<Document> collection = getMongoClient().getCollection(UNSTRUCTURED_DATA);
    FindIterable<Document> findCursor = collection.find(new Document("room", room));
    List<Document> documents = Lists.newArrayList(findCursor);
    return documents;
}
```

Obrázek 38: Metoda selectující data dle označení místnosti

Nad kolekcí je volána metoda *find()* s parametrem konstruktoru třídy Document. Do něj se uvádí klíč a hodnota parametru, čili z jaké místnosti data pochází. Metoda vrací dokumenty v rozhraní **FindIterable**[1]. Kvůli odpovědi na požadavek z controlleru (v modulu api) jsou dokumenty převedeny na obyčejný seznam.

Tento proces explicitně zajistí knihovna Guava[2], z níchž pochází objekt **Lists**.

Způsob jak získat naměřená data v určitém časovém intervalu implementuje již zmíněná metoda *getUnstructuredMeasuredDataByTimestampInterval()*.

[1]Dědí od MongoIterable, které dědí od Iterable. Iterable umožňuje záznamy rovnou procházet ve for cyklu, aniž by byl vytvořen Iterator.

[2]https://mvnrepository.com/artifact/com.google.guava/guava

```
@Override
public List<Document> getUnstructuredMeasuredDataByTimestampInterval(long fromTimestamp, long toTimestamp) {
    com.mongodb.client.MongoCollection collection = getMongoClient().getCollection(UNSTRUCTURED_DATA);
    FindIterable<Document> findCursor = collection.find(and(
            gt(TIMESTAMP_MEASURE, new Timestamp(fromTimestamp*1000)),
            lte(TIMESTAMP_MEASURE, new Timestamp(toTimestamp*1000))));
    List<Document> documents = Lists.newArrayList(findCursor);
    return documents;
}
```

Obrázek 39: Metoda selectující data dle časového intervalu

Tělo metody se liší od předešlé pouze selectující podmínkou. K efektivnímu a přehlednému sestavení podmínky jsou volány statické metody z třídy *com.mongodb.client.model.Filters gt()* a *lte()*, jejichž parametr přijímá údaj o měření ve tvaru klíč,hodnota. Klíč reprezentuje název parametru (timeMeasurement) a hodnota je nová instance třídy Timestamp s časem v milisekundách. Poslední metoda *and()* zastupuje vlastnost logické spojky **and**.

### Zpracování dat přes Jongo driver

Implementace posledního způsobu zpracování navazuje na předešlé tím, že vychází ze stejného rozhraní - EndPointsDocumentRepository.

#### *Rozhraní*

Dojde k jednoduchému rozšíření rozhraní, a to pouze přidáním jedné metody *saveUnstructuredMeasuredDataStraightWithShell()*.

```
public interface EndPointsDocumentRepository {

    void saveUnstructuredMeasuredData(String json);

    void saveUnstructuredMeasuredDataStraightWithShell(String json);

    List<Document> getUnstructuredMeasuredDataByRoom(String room);

    List<Document> getUnstructuredMeasuredDataByTimestampInterval(long fromTimestamp, long toTimestamp);

}
```

Obrázek 40: Přidání metody do rozhraní EndPointsDocumentRepository

Kompletní rozhraní se pak bude injektovat do service.

#### *Implementace rozhraní*

Způsob by mohl mít vlastní implementační třídu, avšak některé části kódu (konstanty, privátní metody) by se zbytečně opakovaly v obou třídách, což není příliš efektivní. Tedy metoda bude přidána do EndPointsDocumentRepositoryImpl.

```
@Override
public void saveUnstructuredMeasuredDataStraightWithShell(String json) {

    LOGGER.info("Data saved to database by third way: \n" + json);
    Jongo jongo = new Jongo(getMongoDB());
    MongoCollection measuredData = jongo.getCollection(MEASURED_DATA);
    measuredData.insert(json);
}
```

Obrázek 41: Metoda pro ukládání přes Jongo

K inicializaci objektu Jongo se využívá instance databáze, v tomto případě objekt DB vložený do parametru do konstruktoru. Jongo poskytuje přístup ke kolekci dokumentů standartně voláním *getCollection("název kolekce")*. Kolekce se namapuje do objektu MongoCollection, pak stačí zavolat metodu *insert()* a do jejího parametru vložit ukládáný řetězec.

Na první pohled to vypadá, že nedochází k žádné deserializaci, avšak Jongo se o to implicitně postará. Uvnitř driveru mapuje objekty defaultně knihovna Jackson, prozatím však provádí deserializaci do zastaralého objektu DBObject (ve vyšší verzi nahrazen objektem Document).

### 4.6.6 Implementace vrstvy service

Vrstva představuje rozhraní mezi modulem api a core.

Obecně, dle konvencí vývoje, bývá na této vrstvě implementována veškerá business logika, avšak v tomto případě pouze zprostředkovává výměnu dat mezi controllery a repozitáři.

#### Rozhraní

Rozhraní je prosté, obsahuje všechny metody, co byly implementovány v rámci vrstvy repository.

```java
package cz.mendelu.smartPef.endpoints.service;

import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;
import org.bson.Document;

import java.util.List;

/**
 * Created by Petr on 14. 9. 2016.
 */
public interface EndPointsService {

    void saveMeasuredStructuredData(String json);

    void saveMeasuredUnstructuredData(String json);

    void saveMeasuredUnstructuredDataWithShell(String json);

    List<Document> getUnstructuredMeasuredDataByRoom(String room);

    List<Document> getUnstructuredMeasuredDataByTimestampInterval(long fromTimestamp, long toTimestamp);

    EndPointsDataControllerEntity getControllerById(String idController);
}
```

Obrázek 42: Rozhraní service

Obdobně jako u vrstvy repository, i zde platí konvence pojmenování viz. End-Points**Service**.

### Implementace rozhraní

Nad deklarací třídy EndPointsServiceImpl se uvádí anotace @Service. Pro přístup do repository postačí injektovat obě rozhraní.

```java
package cz.mendelu.smartPef.endpoints.service.impl;

import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;
import cz.mendelu.smartPef.endpoints.entity.EndPointsSensorEntity;
import cz.mendelu.smartPef.endpoints.repository.EndPointsDocumentRepository;
import cz.mendelu.smartPef.endpoints.repository.EndPointsRepository;
import org.bson.Document;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import cz.mendelu.smartPef.endpoints.service.EndPointsService;

import java.util.List;

/**
 * Created by Petr on 14. 9. 2016.
 */

@Service
public class EndPointsServiceImpl implements EndPointsService{

    @Autowired
    private EndPointsRepository endPointsRepository;

    @Autowired
    private EndPointsDocumentRepository endPointsDocumentRepository;
```

Obrázek 43: Deklarace třídy a atributů EndPointsServiceImpl

Jak napovídají názvy metod, lze snadno odvodit, kterou metodu z repository zavolat.

```
@Override
public void saveStructuredMeasuredData(String json) {
    endPointsRepository.saveStructuredMeasuredData(json);
}

@Override
public void saveUnstructuredMeasuredData(String json) {
    endPointsDocumentRepository.saveUnstructuredMeasuredData(json);
}

@Override
public void saveUnstructuredMeasuredDataStraightWithShell(String json) {
    endPointsDocumentRepository.saveUnstructuredMeasuredDataStraightWithShell(json);
}

@Override
public EndPointsDataControllerEntity getControllerById(String idController) {
    return endPointsRepository.getControllerById(idController);
}

@Override
public List<Document> getUnstructuredMeasuredDataByRoom(String room) {
    return endPointsDocumentRepository.getUnstructuredMeasuredDataByRoom(room);
}

@Override
public List<Document> getUnstructuredMeasuredDataByTimestampInterval(long fromTimestamp, long toTimestamp) {
    return endPointsDocumentRepository.getUnstructuredMeasuredDataByTimestampInterval(fromTimestamp, toTimestamp);
}
```

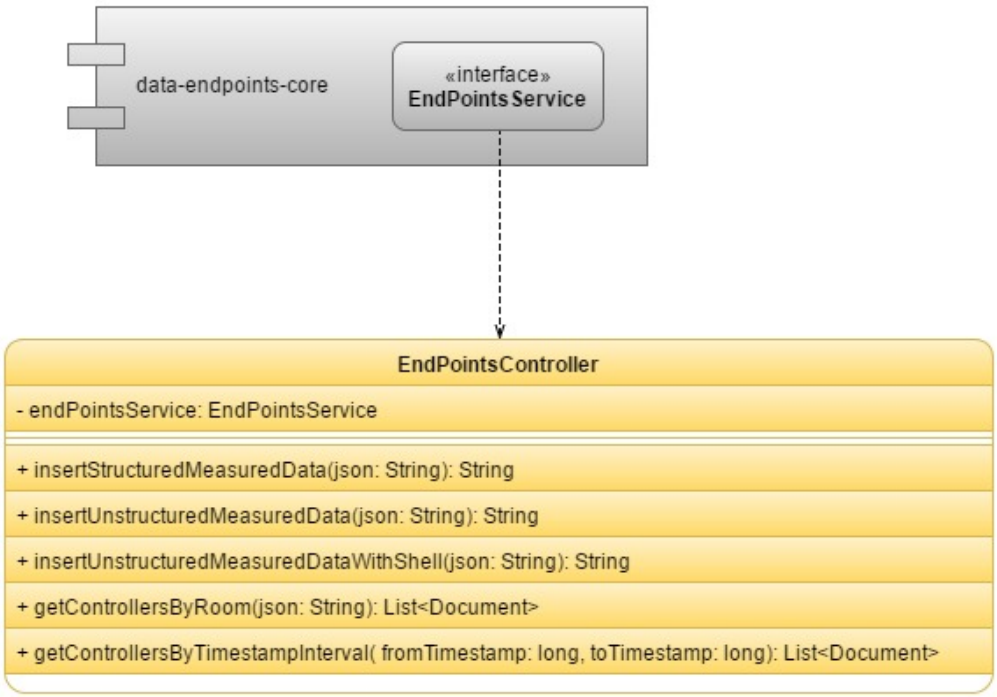
Obrázek 44: Implementace metod vrstvy service

## 4.7 Implementace modulu data-endpoints-api

Modul data-endpoints-core jde připojit stejně jako když se vkládá nová knihovna, tedy vložením následujícího kódu do build.gradle konfigurace:

```
compile project(":smart-pef-data-endpoints:data-endpoints-core")
```

Po tomto kroku bude možné jednoduše injektovat EndPointsService do controlleru, což zobrazuje následující diagram propojení komponent.




Obrázek 45: Diagram propojení komponent modulu core a api

Základ implementace tvoří konfigurační třída v kořenovém balíčku (cz.mendelu.smartPef.endpoints) anotovaná @SpringBootApplication. Oproti modulu core navíc dědí od SpringBootServletInitializer, který registruje jednotlivé servlety (v tomto případě EndPointsController) do servlet kontejneru (aplikační server).

```
package cz.mendelu.smartPef.endpoints;

import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

/**
 * Created by Petr on 18. 9. 2016.
 */
@SpringBootApplication
public class SpringBootEndPointsApiApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootEndPointsApiApplication.class);
    }
}
```

Obrázek 46: Základní konfigurační třída modulu

Následující postup zahrnuje dva dílčí kroky, které jsou na sobě závislé. Nejdříve je nutno implementovat controller, jenž zároveň využívá komponenty z technologie

Swagger 2[1], přičemž v druhém kroku má být tato technologie správně nakonfigurována.

### 4.7.1 Implementace EndPointsController

Controller je komponenta Springu, která nemá vlastní rozhraní, ale pouze implementační třídu. Vystavuje metody REST API na specifických URL adresách dostupných klientovi.

Základ deklarace třídy tvoří anotace @RestController. K tomu se připojí @RequestMapping, v jejímž parametru value, na úrovni deklarace třídy určuje kořen URL adres (dataendpoint) všech metod v tomto controlleru.

```
package cz.mendelu.smartPef.endpoints.controllers;

import cz.mendelu.smartPef.endpoints.entity.EndPointsDataControllerEntity;
import cz.mendelu.smartPef.endpoints.entity.EndPointsSensorEntity;
import cz.mendelu.smartPef.endpoints.service.EndPointsService;
import org.bson.Document;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

/**
 * Created by Petr on 14. 9. 2016.
 */
@RestController
@RequestMapping(value = "/dataendpoint")
public class DataEndPointsController {

    private static final Logger LOGGER = LoggerFactory.getLogger(DataEndPointsController.class);

    @Autowired
    private EndPointsService endPointsService;
```

Obrázek 47: Deklarace EndPointsControlleru

Třída EndPointsController má dva atributy, první je Logger a druhý rozhraní EndPointsService z modulu core. Dále se v controlleru implementují jednotlivé metody, z nichž každá bude anotována @RequestMapping a @ResponseBody. Anotace @RequestMapping poskytuje několik parametrů, z nichž nutné je vyplnit **path**, **method**, **consumes** nebo **produces**.

- path - mapovaná URL,
- method - HTTP verbs,
- consumes - MIME typ těla požadavku[2],
- produces - MIME typ těla odpovědi[2].

[1] http://swagger.io/

[2] V našem případě vždy application/json.

URL adresy nemají udány přesnou strukturu, nicméně je vhodné si stanovit určitou hierarchii, aby bylo možné se v tom lépe orientovat.

@ResponseBody vkládá do těla odpovědi hodnotu, kterou daná metoda vrací.

Zmíněné specifikace byly společné pro všechny. Dále se deklarace metod liší dle jejich účelu. Ty co ukládají data, přijímají požadavky zaslané metodou POST a parametr anotuje @RequestBody, tím je uložena hodnota těla požadavku do něj. Naopak ty co selektují data, vrací je metodou GET a incializují parametr metody hodnotou parametru v URL pomocí @RequestParam. Pokud se do parametru anotace nevloží název, který má v URL hledat, bude defaultně předpokládat, že je stejný jako název proměnné.

Výsledný controller se všemi namapovanými metodami:

```
@RequestMapping(value = "/controller/measurement", method = RequestMethod.POST, consumes = { "application/json" })
@ResponseBody
public String saveMeasuredStructuredData(@RequestBody String json){
    endPointsService.saveMeasuredStructuredData(json);
    return "Data byla vložena do databáze";
}

@RequestMapping(value = "/controller/measurement/unstructured", method = RequestMethod.POST,
        consumes = { "application/json" })
@ResponseBody
public String saveMeasuredUnstructuredData(@RequestBody String json){
    endPointsService.saveMeasuredUnstructuredData(json);
    return "Data byla vložena do databáze";
}

@RequestMapping(value = "/controller/measurement/unstructured/nonparse", method = RequestMethod.POST,
        consumes = { "application/json" })
@ResponseBody
public String saveMeasuredUnstructuredDataWithShell(@RequestBody String json){
    endPointsService.saveMeasuredUnstructuredDataWithShell(json);
    return "Data byla vložena do databáze";
}

@RequestMapping(value = "/controllers/measurement/room", method = RequestMethod.GET,
        produces = {"application/json"})
@ResponseBody
public List<Document> getUnstructuredMeasuredDataByRoom(@RequestParam("room") String room){
    List<Document> measuredData = endPointsService.getUnstructuredMeasuredDataByRoom(room);
    return measuredData;
}

@RequestMapping(value = "/controllers/measurement/interval", method = RequestMethod.GET,
        produces = {"application/json"})
@ResponseBody
public List<Document> getControllersByTimestampInterval(@RequestParam("fromTimestamp") long fromTimestamp,
                                                         @RequestParam("toTimestamp") long toTimestamp){
    List<Document> measuredData =
            endPointsService.getUnstructuredMeasuredDataByTimestampInterval(fromTimestamp, toTimestamp);
    return measuredData;
}

@RequestMapping(value = "/controllers", method = RequestMethod.GET)
@ResponseBody
public String getControllerById(@RequestParam("idController") String idController) {
    LOGGER.info("ID CONTROLLERU je: " + idController);
    EndPointsDataControllerEntity controllerEntity = endPointsService.getControllerById(idController);
    return controllerEntity.toString();
}
```

Obrázek 48: Namapování všech metod

Pokud by byl v tomto stavu zdrojový kód zbuildován do webového archivu a nasazen na aplikační server, mohl by již klient zasílat požadavky na REST API. Implementace celého rozhraní je téměř hotová. V dalším kroku bude vygenerována dokumentace Swaggerem.

### 4.7.2 Dokumentace Swagger 2

Krok přidání knihovny do build.gradle lze přeskočit, jelikož byl proveden v rámci konfigurace build modulů.

Prvním krokem je vložit anotaci @EnableSwagger2 nad deklaraci třídy hlavní konfigurace (nad @SpringBootApplication). Dále musí být vytvořena třída, kterou se Swagger konfiguruje.

```
package cz.mendelu.smartPef.endpoints.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;

/**
 * Created by psadovsky on 10. 10. 2016.
 */
@Configuration
public class SwaggerDocumentationConfig {

    ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("SmartPEF API Data endpoints")
                .description("API to access measured data from sensors.")
                .license("")
                .licenseUrl("")
                .termsOfServiceUrl("")
                .version("1.0.0")
                .contact(new Contact( name: "Petr Sadovský", url: "",  email: "xsadovsk@node.mendelu.cz"))
                .build();
    }

    @Bean
    public Docket customImplementation(){
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage("cz.mendelu.smartPef.endpoints"))
                .build()
                .apiInfo(apiInfo());
    }

}
```

Obrázek 49: Konfigurace Swagger 2

Anotace @Configuration definuje předpoklad, že v této třídě bude jedna nebo více metod anotovaných @Bean a Spring si je přidá do kontejneru. Do metody *apiInfo()* lze specifikovat základní informace o rozhraní. Vrací vytvořenou instanci třídy ApiInfoBuilder, která poslouží k aktivování Swaggeru. Inicializuje Docket, v němž musí být specifikován základní balíček, od něhož budou skenovány prvky (anotace) frameworku, jimiž vzniká dokumentace. Anotovány bývají především controllery, čili by mohlo být dostatečné skenovat pouze balíček *controllers*.

Kromě REST controllerů se přidává ještě controller zajišťující domovskou stránku.

```java
package cz.mendelu.smartPef.endpoints.configuration;

import io.swagger.annotations.Api;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * Created by psadovsky on 10. 10. 2016.
 */
@Controller
@Api(hidden = true)
public class HomeController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String index() {
        System.out.println("swagger-ui.html");
        return "redirect:swagger-ui.html";
    }

}
```

Obrázek 50: Domovská stránka dokumentace

Deklarace třídy zahrnuje anotaci Springu @Controller. Pod ní anotace @Api již pochází ze Swagger a udává, že jde již o konkrétní stránku dokumentace. Controller obsahuje jedinou metodu třídy *index()*[1] standartně namapovanou @RequestMapping, kde v parametru value je vložen kořen URL adresy "/".

V této fázi, by po nasazení webového archivu bylo možné vidět webové rozhraní se všemi controllery, i přesto že zatím některé nebyly namapovány Swagger anotacemi. Swagger detekuje controllery přes anotace Springu.

Framework poskytuje spoustu popisných prvků, avšak ne všechny musí být povinně použity. Z čehož plyne, že záleží na každém vývojáři, jak detailně chce svou práci zdokumentovat.

Základem dokumentace je namapovat controller anotací @Api, což stanoví kořen od něhož se budou dále řadit jednotlivé metody anotované @ApiOperation. Pokud není z názvu parametru metody jasné co představuje, lze popis explicitně definovat v @ApiParam. Swagger také poskytuje například doplnění popisu HTTP odpovědí pomocí @ApiResponse, pro jednotlivé status kódy a další specifikace, které tvůrci udávají v oficiální dokumentaci[2].

Po doplnění anotací do EndPointsControlleru, nelze optimálně zachytit celý kód do jednoho obrázku. Pro demonstraci principu dostačuje jedna metoda.

[1] Běžná konvence webových stránek, psát domovskou stránku jako např. index.php, index.html apod.

[2] http://swagger.io/specification/

```
@RestController
@RequestMapping(value = "/dataendpoint")
@Api(value = "/dataendpoint", description = "accessing and saving measured data",
        consumes = "application/json", produces = "application/json")
public class DataEndPointsController {

    private static final Logger LOGGER = LoggerFactory.getLogger(DataEndPointsController.class);

    @Autowired
    private EndPointsService endPointsService;

    @ApiOperation(value = "Return measured data by timestamp interval.")
    @RequestMapping(value = "/controllers/measurement/interval", method = RequestMethod.GET,
            produces = {"application/json"})
    @ResponseBody
    public List<Document> getControllersByTimestampInterval(@ApiParam(value = "Lower limit timestamp in seconds.")
                                                             @RequestParam("fromTimestamp") long fromTimestamp,
                                                             @ApiParam(value = "Upper limit timestamp in seconds.")
                                                             @RequestParam("toTimestamp") long toTimestamp){
        List<Document> measuredData =
                endPointsService.getUnstructuredMeasuredDataByTimestampInterval(fromTimestamp, toTimestamp);
        return measuredData;
    }
}
```

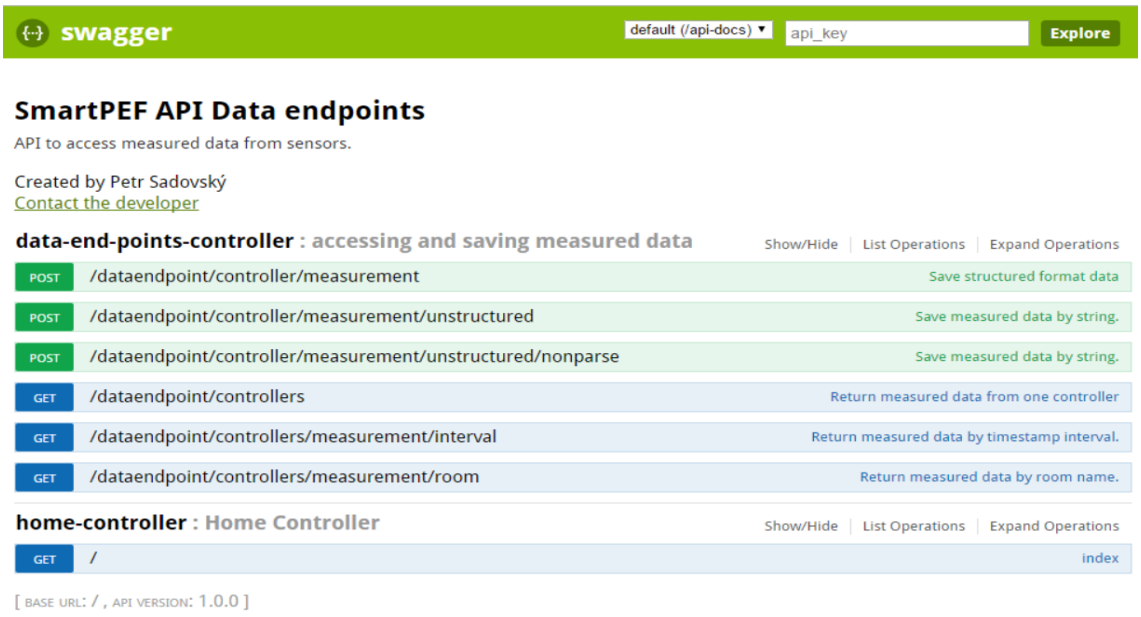
Obrázek 51: Namapovaný EndPointsController pomocí Swagger 2

Implementace rozhraní je hotová a může být testována. K vyzkoušení aplikace stačí sestavit z výsledného kódu webový archiv a nasadit archiv na aplikační server (Tomcat 8).

Zbuildit kód jde přes terminál/příkazovou řádku příkazem **gradle war**, ale musí ukazovat do adresáře modulu, kde se nachází soubor **build.gradle**. Případně většina vývojových prostředí umožňuje proces provést přes uživatelské rozhraní, což bude popsáno v dokumentaci.

Výsledný archiv se manuálně umísťuje v aplikačním serveru do adresáře webapps a po startu archiv server rozbalí. Nicméně i tento postup lze zautomatizovat ve vývojovém prostředí.

Výsledná dokumentace ve webovém prohlížeci:




Obrázek 52: Webové rozhraní dokumentace REST API

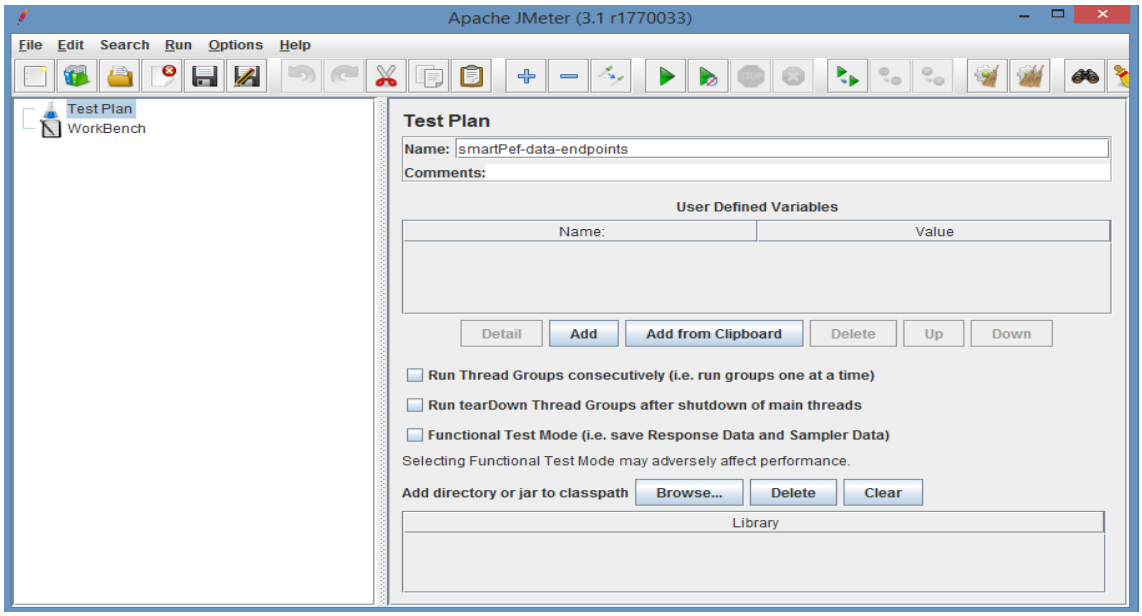
## 4.8 Testování navrženého řešení

Výsledkem celého testování má být předpoklad k výběru nejefektivnější technologie na zpracování dat ze tří implementovaných.

REST API s připojením do databáze je kompletní. Dále se musí vytvořit testovací plán v JMeteru.

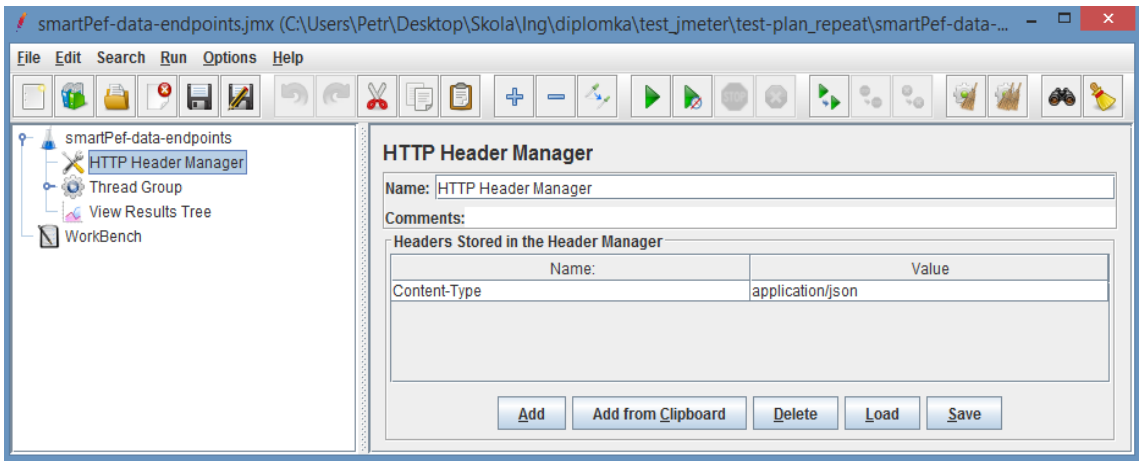
Tvůrci JMeteru doporučují vytvářet testovací script v GUI módu (je to o mnoho jednodušší) a vlastní proces testování pak provádět přes příkazovou řádku.

Spuštění GUI jde v příkazové řádce, jestliže ukazuje do adresáře bin JMeteru scriptem jmeter.bat nebo vložením absolutní cesty k bin do Path odkudkoliv. Po spuštění se zadává název testovacího plánu.




Obrázek 53: Pojmenování testovacího plánu

O úroveň níž pod názvem testovacího plánu je specifikován formát dat přenášených v HTTP požadavku pomocí **HTTP Header Manager**. Na stejné úrovni bývá vložen objekt **View Results Tree** sloužící k prohlížení požadavků za běhu testu. Nicméně při testování bez GUI objekt nejde optimálně využít.




Obrázek 54: Vytvoření HTTP Header Managera

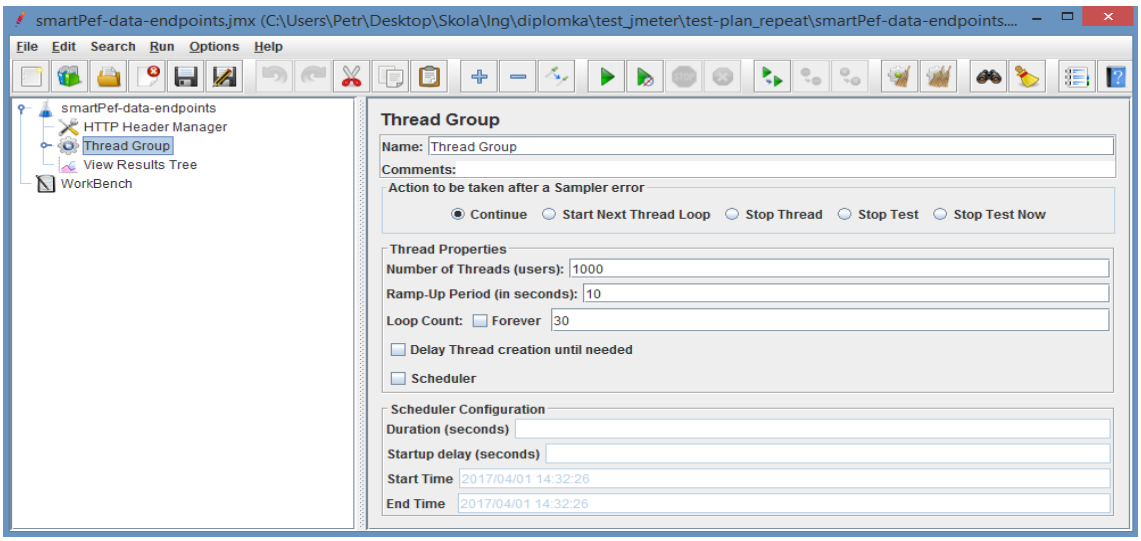
Klíčový prvek, pod který spadají další komponenty, je **Thread Group**. Jsou v něm specifikovány hodnoty: maximální počet spouštěných vláken (Number of

Threads (users)), zpoždění synchronního běhu všech vláken (Ramp-up period) a počet iterací procesu (Loop count).

Testovací případ spustí 100, 250, 500, 1 000 vláken s periodou 10 sekund a počtem iterací 30. Pro představu při 1 000 vláknech, každou sekundu přibyde 100 vláken a to opakovaně 30-krát, tudíž na rozhraní přijde celkem 30 000 požadavků. Pokud nastane při testu libovolná chyba, což může být například zahození požadavku, lze na to reagovat několika způsoby. Bud zastavení testu ihned (Stop Test Now), zastavení testu po dokončení operace (Stop test), pokračovat od dalšího cyklu (Start Next Thread Loop), zastavit vlákno (Stop Thread) nebo test kompletně dokončit (Continue).

Test by měl proběhnout bez ohledu na výskyt chyb.

Dalšími specifikacemi mohou být nastaveny časové intervaly, ale to pro tento test není žádoucí.

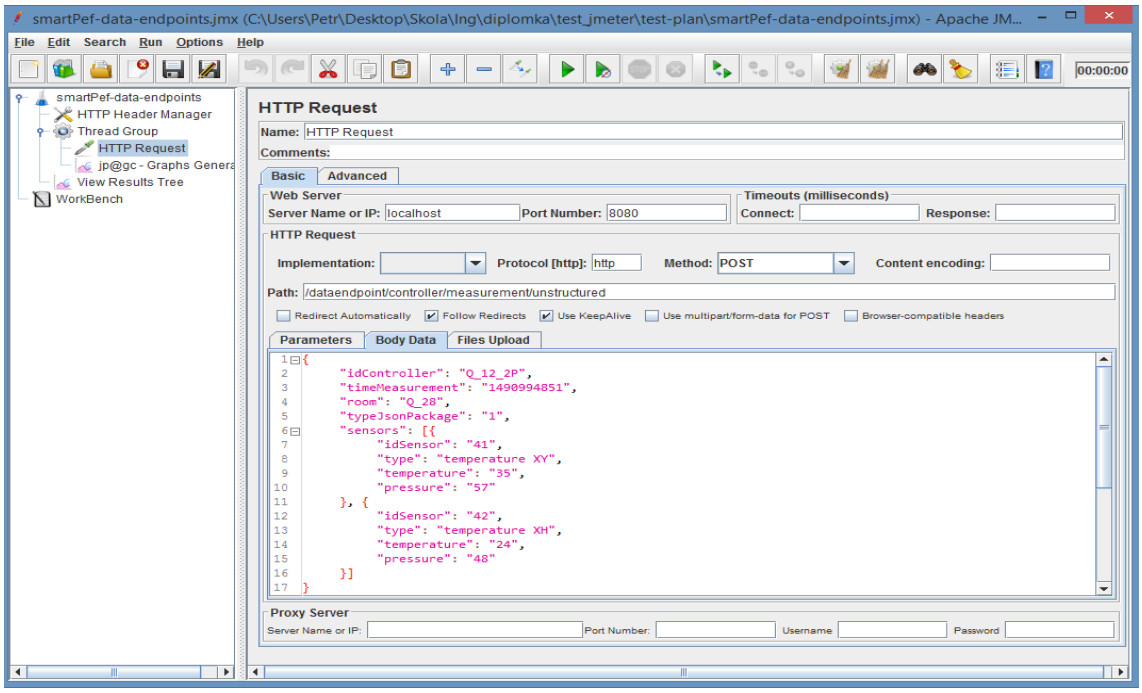



Obrázek 55: Konfigurace vláken

O další úroveň níž, pod Thread Group se vytváří HTTP požadavek (**HTTP Request**). Povinně musí být vyplněny tyto parametry:

- název serveru (Server Name or IP),
- číslo portu (Port Number),
- protokol (Protocol [http], http je zároveň default),
- metoda (Method),
- data v těle požadavku (Body Data).

Ostatní parametry mohou mít defaultní nastavení.

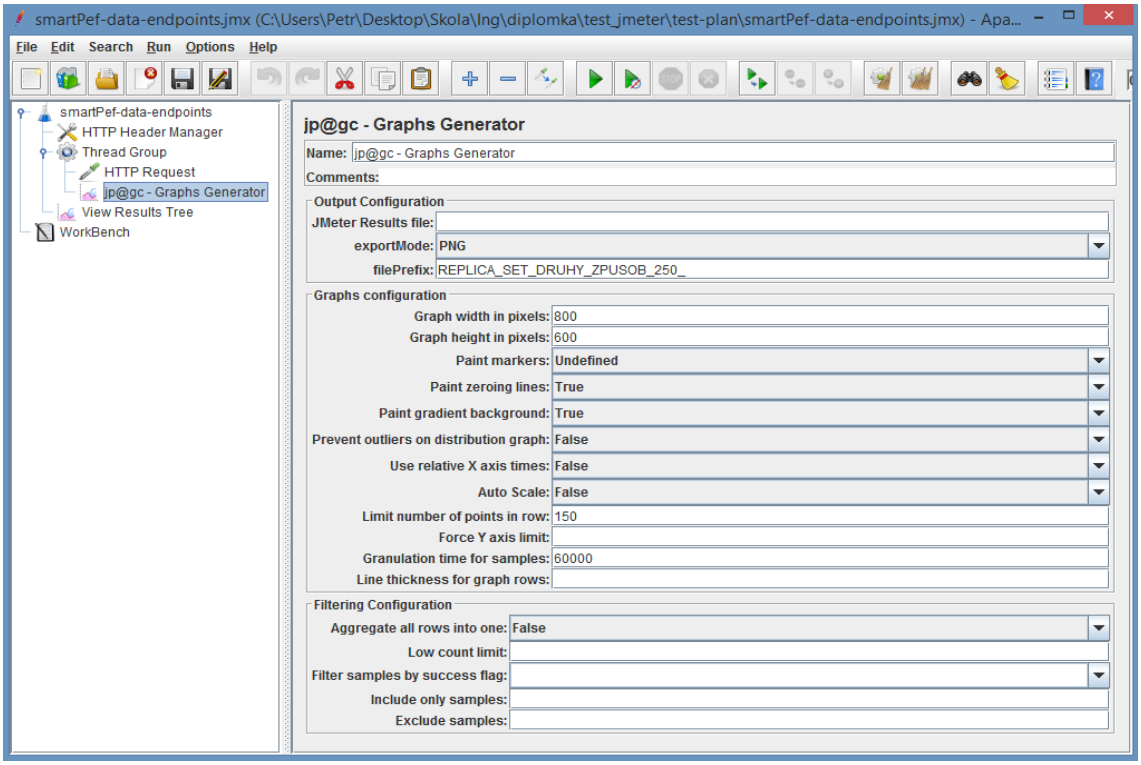



Obrázek 56: Konfigurace HTTP požadavku

Průběh testování lze zobrazit v grafu. Přes uživatelské rozhraní jde monitorovat průběh testu i za běhu, nicméně v příkazové řádce se vygenerují grafy defaultně do adresáře odkud se spouští test.

JMeter má několik defaultních komponent pro generování grafů, avšak je možné pomocí **JMeter Plugins Manageru**[1] přidat další. Například Graphs Generator Listener[2], který na konci testování vygeneruje 11 grafů popisujících různé metriky. Nutno podotknout, jak je psáno v dokumentaci, že vždy o úroveň výš před listenerem, by měl být vytvořen View Results Tree.

[1] https://jmeter-plugins.org/wiki/PluginsManager/
[2] https://jmeter-plugins.org/wiki/GraphsGeneratorListener/



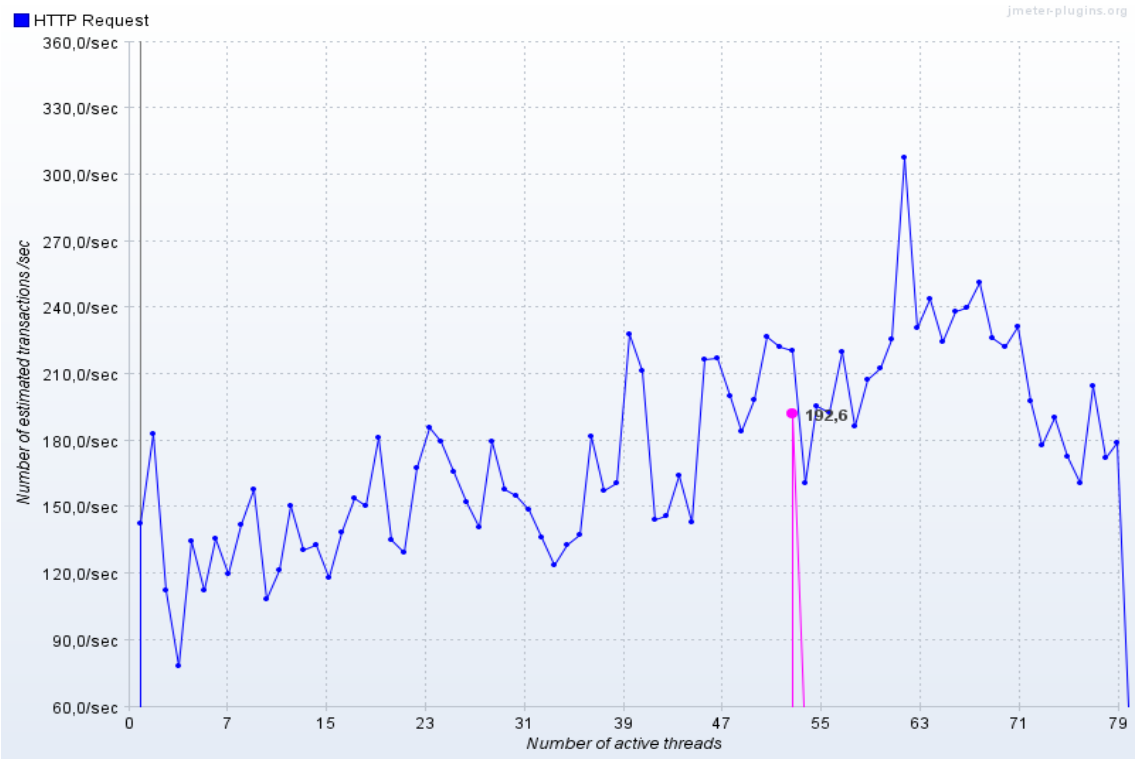

Obrázek 57: Konfigurace Graphs Generator Listener

Data generovaná během testování jsou uložena buď do CSV nebo PNG souboru (exportMode). Během toho, jak se mění vstupní parametry testu a grafů přibývá, užitečným identifikátorem je prefix souboru (filePrefix), jenž se na konci testu přidá před název grafu. Ostatní parametry mohou zůstat v defaultním nastavení.

Test se spouští z příkazové řádky příkazem:

```
jmeter -t "název_testovacího_scriptu" -n
```

Parametr ”-n”znamená režim v příkazové řádce. Výsledný graf pak může vypadat takto:




Obrázek 58: Graf poměru počtu požadavků k počtu aktivních vláken

### 4.8.1 Testování na jednom stroji

Použitý stroj byl běžný uživatelský počítač (notebook), čili kdyby testování proběhlo na výkonném serveru (kde v budoucnu bude vše běžet), výsledné hodnoty by byly určitě jiné.

Tabulka 5: Technické údaje počítače

| | |
|---|---|
| **Značka** | HP Pavilion |
| **Operační systém** | Windows 8.1 |
| **Procesor** | Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz |
| **Operační paměť** | 8GB Samsung 1600MHz |
| **Grafická karta** | NVIDIA GeForce 840M - 3 840 MB |
| **Disk** | SSD |

Nicméně výsledek by to neovlivnilo, jelikož všechny testované případy probíhaly za stejných podmínek.

Během testu došlo k chybě[1] v okamžiku, kdy se objevil první nezpracovaný požadavek. Vyvolává ji objekt ErrorPageFilter, který má přesměrovat chybové hlášky

[1]ErrorPageFilter: Cannot forward to error page for request ...

na základě implementovaného zpracování (do logů jako výjimku, nebo na chybovou stránku apod.). Faktem je, že jakýkoliv výpis by zbytečně zatěžoval server v průběhu testu, proto v tomto případě mohou být logy[1] i filtr zastaveny. Filtr lze zastavit následujícím kódem, umístěným do hlavní konfigurace api modulu (SpringBootEndPointsApiApplication):

```
@Bean
public ErrorPageFilter errorPageFilter() { return new ErrorPageFilter(); }

@Bean
public FilterRegistrationBean disableSpringBootErrorFilter(ErrorPageFilter filter) {
    FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean();
    filterRegistrationBean.setFilter(filter);
    filterRegistrationBean.setEnabled(false);
    return filterRegistrationBean;
}
```

Obrázek 59: Zastavení ErrorPageFilteru

Výsledná tabulka udává konečný počet požadavků, zpracovaných za sekundu, v poměru k procentuálnímu počtu nezpracovaných.

Tabulka 6: Testování na jednom stroji

| **počet vláken** | **1.způsob** | **2.způsob** | **3.způsob** |
|---|---|---|---|
| 100 | 228,1/0 | 153,7/0 | 53,4/0 |
| 250 | 309,0/0 | 107,0/7,53 | 74,0/14,72 |
| 500 | 407,7/0 | 107,2/7,89 | 84,0/9,89 |
| 1000 | 560,6/0 | 53,8/7,84* | 47,2/9,59* |

* Předčasné ukončení testu z důvodu rapidního poklesu rychlosti zpracování. Procentuální počet nezpracovaných požadavků stále narůstal a schopnost zpracovávat nově příchozí požadavky se snižovala.

Z výsledku testu vyplývá, že první způsob je nejefektivnější.

### 4.8.2 Testování pomocí replikace

V testování na jednom stroji postačila standartní konfigurace MongoDB, ale zde proběhně úprava v podobě replikace databáze na třech strojích. Ta může být prováděna i na jednom stroji[2].K předešlém stroji se přidají další dva, tedy dojde k využití jejich systémových prostředků ke zvýšení celkového výkonu databáze.

Stručný popis technických parametrů obou strojů:

[1]Logování jde zakázat buď v konfiguraci Tomcat, ve vlastním application.properties nebo implementací.

[2]https://docs.mongodb.com/manual/tutorial/deploy-replica-set-for-testing/

Tabulka 7: Technické údaje druhého počítače

| | |
|---|---|
| **Značka** | HP Elite 8570w |
| **Operační systém** | Windows 8.1 |
| **Procesor** | Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz |
| **Operační paměť** | 16GB 800MHz |
| **Grafická karta** | NVIDIA Quadro K1000M |
| **Disk** | SSD |

Tabulka 8: Technické údaje třetího počítače

| | |
|---|---|
| **Značka** | MSi GE60 2PL Apache |
| **Operační systém** | Windows 10 |
| **Procesor** | Intel(R) Core(TM) i5-4210H CPU @ 2.90GHz |
| **Operační paměť** | 8GB DDR3L 1600MHz |
| **Grafická karta** | NVIDIA GeForce GTX 850M 2GB VRAM DDR3 |
| **Disk** | SSD |

Znovu lze podotknout, že oba stroje jsou běžné uživatelské počítače (notebooky), čili na serverech by výkon rapidně vzrostl a s tím i výsledné hodnoty testu.

Celý následující postup vychází z postupu uvedeného v oficiální dokumentaci MongoDB[1].

Každý počítač byl připojen do jedné sítě přes router pomocí UTP kabelu, jelikož kabel má větší propustnost oproti přenosu bezdrátovou sítí.

Na každém stroji musí proběhnout standartní instalace MongoDB. Předtím než dojde ke konfiguraci replikační množiny, je důležité zkontrolovat, zda se každý uzel může připojit ke všem sousedícím[2]. Po spuštění všech tří instancí mongod, lze test provést příkazem:

```
mongo --host "hostname" --port 27017
```

Jedná se o standartní příkaz pro spuštění mongo shellu, avšak specifikováním hostname, lze ustavit připojení do vzdálené instance Monga, pokud tedy bylo úspěšně navázáno síťové spojení. Může nastat problém například s firewallem, která se řeší vložením výjimky opravňující přístup.

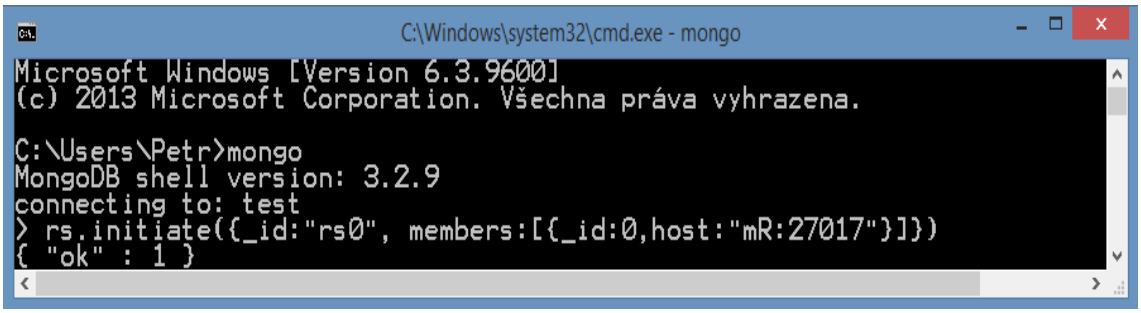
Všechny instance jsou restartovány s parametrem ”replSet”, za nímž následuje název množiny.

```
mongod --replSet "rs0"
```

[1] https://docs.mongodb.com/manual/tutorial/deploy-replica-set/

[2] https://docs.mongodb.com/manual/tutorial/troubleshoot-replica-sets/#replica-set-troubleshooting-check-connection

Možnost inicializovat konfiguraci má libovolný počítač vstupem do mongo shellu. V rámci shellu existuje funkce *rs.initiate()*, do níž se vkládá JSON objekt s počáteční konfigurací obsahující aktuální instanci.




Obrázek 60: Počáteční konfigurace replikační množiny

Zadáním *rs.conf()* je vypsána celá dosavadní konfigurace pro kontrolu. Přidání zbylých dvou uzlů může udělat pouze primární uzel přes *rs.add("hostname:port")*[1]. Stav všech instancí i s popisem označení uzlů udává funkce rs.status().

Pokud by nevyhovovala volba uzlů, jde explicitně primárního zvolit následujícím postupem:

1. Na sekundárním uzlu, který se nemá stát primárním, je spuštěna funkce rs.freeze(120)[2],
2. Na primárním uzlu se spustí funkce rs.stepDown(120)[2],
3. Automaticky bude zvolen požadovaný uzel za primární.

Replikace byla nakonfigurována, ale ještě chybí modifikace připojení REST rozhraní ke všem třem replikám.

Modifikace zahrnuje nahrazení klíčů spring.data.mongodb.database, spring.data.mongodb.host a spring.data.mongodb.port v souboru application.properties za jeden klíč, a to za URI spring.data.mongodb.uri. Celá URI pro všechny repliky má tvar:

```
mongodb://hostname1:27017,hostname2:27017,hostname3:
27017/endpoints?replicaSet=rs0
```

Spring s replikami implicitně naváže spojení, ale drivery používají explicitně komponentu MongoConnection, čili k ostatním atributům komponenty se musí přidat ještě atribut **mongoURI** anotovaný @Value("${spring.data.mongodb.uri}"). V metodách *getMongoDatabase* a *getMongoDB* je zaměněn řádek s incializací objektu MongoClient za:

[1]Defaultně pokud není zadán port, nastaví 27017.
[2] Zabrání, aby byl uzel zvolen primárním na dobu 120 sekund.

```
MongoClient client = new MongoClient(new MongoClientURI(mongoURI));
```

Modifikovaný zdrojový kód musí být znovu zbuilděn do webového archivu a nasazen na server.Pak může proběhnout další testování.

Stejně jako u prvního testování, i zde hodnoty v tabulce udávají poměr počtu zpracovaných požadavků za sekundu vzhledem k procentuálnímu počtu nezpracovaných.

Tabulka 9: Testování na třech strojích

| **počet vláken** | **1.způsob** | **2.způsob** | **3.způsob** |
|---|---|---|---|
| 100 | 210,0/0 | 87,3/0 | 74,9/0 |
| 250 | 470,1/0 | 70,5/0 | 94,0/0 |
| 500 | 1032,7/0 | 15,8/1,10 | 20,2/0,30 |
| 1000 | 1532,0/0 | 23,2/2,81* | 29,2/10,06* |

* Předčasné ukončení testu z důvodu rapidního poklesu rychlosti zpracování. Procentuální počet nezpracovaných požadavků stále narůstal a schopnost zpracovávat nově příchozí požadavky se snižovala.

Stejně jako v prvním testu, i zde nejlépe obstál první způsob. Nicméně lze si všimnout, že u druhého a třetího způsobu výrazně poklesl procentuální počet nezpracovaných požadavků, což zajistila právě replikace.

# 5 Ekonomické zhodnocení

V této kapitole bude popsáno ekonomické zhodnocení navrženého a implementovaného řešení.

Mezi hlavní aspekty týkající se ekonomické efektivnosti tohoto řešení, lze zahrnout vhodný výběr datového úložiště a nasazení celého řešení.

## 5.1 Výkon NoSQL vs. komerční RDBMS

Pokud pro daný případ užití neexistují okolnosti kladoucí důraz na přítomnost transakčního zpracování (ACID transakce) a zároveň je vyžadován vysoký výkon, je sáhnout po NoSQL úložišti jedno z efektivních řešení.

Například relační databáze, která by byla použitá k implementaci sociální sítě, může na testovacím prostředí být dostatečná, avšak enormní nepravidelný narůst dat na produkčním prostředí již zapříčiní problémy s výkonem. Tento problém řeší koupě nového výkonějšího hardwaru, nebo u NoSQL škálovací model umožní přidání serveru do clusteru. Tím dojde k postupnému růstu bez náhrady stávajících investic do hardwaru (Kim, 2016).

Kromě existence open-source relačních databází (PostgreSQL, MySQL apod.), existují také komerční RDBMS. Pořizování jejich licencí může značně zatěžovat rozpočet. Nejznámější a nejpoužívanější komerční RDBMS v dnešní době je Oracle[1]. Poskytuje několik druhů licencí, z nichž pro představu uvedu Standart Edition 2, jejíž cena činí 17 500 dolarů za jednotku[2].

Výpočet nákladů na nasazení 3 serverů pod licencí Standart Edition 2, kde každý obsahuje 2 procesory Intel Xeon X7560 (8 jader)[3] :

$$3 \text{ servery} * 2 \text{ sokety/server} = 6 \text{ jednotek} \tag{1}$$

$$6 \text{ jednotek} * 17500 = 105000 \text{ dolarů} \tag{2}$$

Ve srovnání s open-source NoSQL úložišti, komerční RDBMS poskytuje sice také vysoký výkon, ale nese sebou značné náklady.

## 5.2 Nasazení implementovaného řešení

Projekt SMART PEF disponuje výhodou, že má k dispozici univerzitní server, tedy není potřeba zajišťovat další hardware za účelem nasazení.

Nicméně pokud by nebyl k dispozici, je vhodné mít přehled o aktuální nabídce serverů na trhu.

Výběr vychází ze složení strojů uvedeného v rámci testování, čili jsou dvě možnosti:

[1] https://db-engines.com/en/ranking

[2] http://www.oracle.com/us/corporate/pricing/technology–price–list–070617.pdf

[3] http://www.dba–oracle.com/t_licensing_pricing.htm

1. 3 servery s nižším výkonem (cluster),
2. 1 výkonný server ( + 1 záložní server).

Jeden výkonný server má odpovídat stejnému výkonu, jaký zajistí 3 slabší servery propojené do clusteru.

Níže uvedené stroje pochází z nabídky na serveru CZC.cz[1]. Nutno podotknout, že jejich technické vybavení v nabídce odpovídá pouze základní konfiguraci, tudíž se předpokládá, že v reálném nasazení by byla podle potřeby rozšířena o další komponenty (procesory, operační paměť, disky apod.).

### Kalkulace nákladů na pořízení 3 stejně výkonných serverů

Všechny tři servery jsou stejného typu[2].

Tabulka 10: Fujitsu Primergy TX1320M1

| | |
|---|---|
| **Procesor** | Intel Xeon (Haswell/4. generace)<br>E3-1220 v3(3.1/3.5GHz 4jádra/4vlákna) |
| **Operační paměť** | 8 GB DDR3 |
| **Interní paměť** | 1000 GB (7200RPM) |
| **Cena** | **26245 Kč s DPH** |

Výsledná kalkulace:

$$26245 * 3 = \underline{\underline{\mathbf{78735}}} \text{ Kč} \tag{3}$$

### Kalkulace nákladů na pořízení 1 výkonného a 1 záložního serveru

Při obecných základních konfiguracích nelze plně docílit, aby výkon daného serveru[3] odpovídal přesně výkonu tří, muselo by se jednat o řešení stavěné na míru, čili byl učiněn přibližný výběr z nabídky.

Úlohu záložního serveru zajistí stejný typ jako v kalkulaci výše - Fujitsu Primergy TX1320M1.

Tabulka 11: Lenovo System x TS x3650 M6

| | |
|---|---|
| **Procesor** | Intel Xeon (Broadwell/5. generace)<br>E5-2650 v4(2.2/2.9GHz 12jader/24vláken) |
| **Operační paměť** | 16 GB DDR4 |
| **Interní paměť** | chybí disk (volné sloty) |
| **Cena** | **92238 Kč s DPH** |

[1] https://www.czc.cz/server/servery/hledat

[2] https://www.czc.cz/fujitsu–primergy–tx1320m1–e3–1220v3–8gb–2x500gb/158941/produkt

[3] https://www.czc.cz/lenovo-system-x-ts-x3650-m6-e5-2650v4-16gb-bez-hdd-1x750w-rack/190846/produkt

Výsledná kalkulace:

$$92238 + 26245 = \underline{\underline{\mathbf{118483}}} \text{ Kč} \tag{4}$$

Clustery jsou obvykle nasazovány za účelem zvýšení výkonu a dostupnosti oproti jednomu počítači. Kromě toho bývají mnohem levnější než samostatný počítač se srovnatelnou rychlostí nebo dostupností (Bader, Pennington, 2001). Stejný závěr lze vyvodit i z výsledku kalkulace, tedy že výhodnější by bylo nakoupit 3 slabší servery a z nich sestavit cluster.

# 6 Závěr

Na základě analýzy datových úložišť a konzultace návrhu s týmem podílejícím se na projektu SMART faculty vyplynulo, že dokumentová databáze MongoDB bude pro případ užití projektu vhodná. Na toto rozhodnutí navázala praktická zkušenost s tímto produktem. Ta zahrnovala v první řadě instalaci, konfiguraci a otestování základních operací nad databází. Vše proběhlo úspěšně a dalším krokem byla implementace REST rozhraní.

Některé zvolené technologie k implementaci byly pro mne nové, ale s většinou jsem se již setkal. Díky dřívější zkušenosti nemusela každé implementaci předcházet detailní studie dané technologie. Naopak tyto zkušenosti dopomohly k tomu, aby řešení bylo správně navržené a implementované.

Z hlediska testování byl očekáván opačný výsledek, tedy že drivery budou efektivnější než objekty z knihovny Spring. Z toho plyne, že testování splnilo svůj účel a na základě výsledku mohla být zvolena primární technologie pro zpracování dat.

Dále pak testování ukázalo, jak velký vliv má replikace databáze na výkon a stabilitu, což by se určitě mělo zohlednit, při jejím nasazování na serveru. Z ekonomického zhodnocení zároveň vyplývá, že nasazovat na cluster je z hlediska pořizovacích nákladů efektivnější.

V následujících odstavcích jsou popsána možná vylepšení REST rozhraní.

Podle výsledku testování nejlépe zpracovává data technologie Springu, avšak s přibývajícími typy měřících zařízení, jenž mohou generovat mírně odlišnou strukturu dat, by mohl nastat problém. Ten se týká různorodosti generovaných dat. Každá vlastní entita má specifickou strukturu a při mapování musí i tomu odpovídat data. Jedním z řešení je vytvořit entitu, která bude předkem, od níž podědí ostatní společné atributy vyskytující se v datech. Nicméně i tak přibydou nové entity.

Vzhledem k tomu, že přes rozhraní neputují citlivá data, zatím není řešena bezpečnost. Její vylepšení souvisí s přenášením objektů mezi vrstvami namísto obyčejných řetězců. Data přijatá v controlleru incializují objekt odpovídající struktuře dat a pak za pomoci konverze se předává mezi vrstvami.

Poslední vylepšení se týká zpracování výjimek. Ve vlastním projektu je optimální implementovat pro obecné problémy například s databází, autentizací, strukturou dat apod. Výjimky jsou vyhazovány na vrstvě repository, z níž putují do service, kde jsou předávány dál do controllerů. V controllerech se odchytávají a zpracovávají většinou zalogováním, případně přídáním do odpovědi na požadavek.

# 7 Reference

[Andlinger, 2013] Andlinger, Paul.*RDBMS dominate the database market, but NoSQL systems are catching up.*DB-engines [online]. 2013, , 1 [cit. 2016-07-22].Dostupné z: http://db-engines.com/en/blog_post/23.

[Apache JMeter, 2017] *Apache JMeter* [online]. 2017 [cit. 2017-04-22]. Dostupné z:http://jmeter.apache.org/.

[Apache Tomcat, 2017] *Apache Tomcat* [online]. 2017 [cit. 2017-02-19]. Dostupné z:http://tomcat.apache.org/.

[Bader, Pennington, 2001] Bader, David A., Pennington, Robert.*Cluster Computing: Applications*. The International Journal of High Performance Computing Applications [online]. 2001 [cit. 2017-05-03]. Dostupné z: http://www.cc.gatech.edu/~bader/papers/ijhpca.pdf.

[Chodúr, 2016] Chodúr, Martin.*MongoDB*[online]. In: . 2016 [cit. 2017-03-22]..

[DB-Engines, 2017] *DB–Engines: Knowledge Base of Relational and NoSQL Database Management Systems* [online]. 2017 [cit. 2017-04-30]. Dostupné z: https://db-engines.com.

[Dhaneshwar, 2015] Dhaneshwar, Shashank.*NoSQL: A Silver Bullet for handling Big Data?*. In: LinkedIn [online]. 2015 [cit. 2017-04-22]. Dostupné z: https://www.linkedin.com/pulse/nosql-silver-bullet-handling-big-data-shashank-dhaneshwar.

[Doglio, 2015] Doglio, Fernando.*Pro REST API Development with Node.js*[online]. Apress, 2015 [cit. 2017-04-16]. ISBN 978-1-4842-0917-2.

[Gradle Build Tool 3.3, 2017] *Features.*Gradle Build Tool 3.3 [online]. 2017 [cit. 2017-02-19]. Dostupné z: https://docs.gradle.org/3.3/userguide/overview.html.

[Grolinger, Higashino, Tiwari, Capretz, 2013] Grolinger, Katarina, Higashino, Wilson A, Tiwari, Abhinav, Capretz, Miriam AM.*Data management in cloud environments: NoSQL and NewSQL data stores.*Journal of Cloud Computing a SpringerOpen Journal [online]. 2013, , 24 [cit. 2016-07-22]. DOI: 10.1186/2192113X222.Dostupné z:http://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-2-22.

[Harrison, 2015] Harrison, Guy. *Next generation databases: NoSQL, NewSQL, and Big Data* [online]. Apress, 2015 [cit. 2017-02-19]. Expert's voice in Oracle. ISBN 978-1-4842-1329-2.

[Java Community Process, 2017] Java Community Process.*Community Development of Java Technology Specifications*. Oracle [online]. 2017 [cit. 2017-02-12]. Dostupné z: https://www.jcp.org/en/home/index.

[Kainulainen, 2017] Kainulainen, Petri.*Getting Started With Gradle* [online]. 2017 [cit. 2017-02-19]. Dostupné z: https://www.petrikainulainen.net/getting-started-with-gradle/.

[Kim, 2016] Kim, Dale.*NoSQL vs. SQL: It's About the Performance and Scale*.Dataversity: Data Education for Business and IT Professionals [online]. 2016 [cit. 2017-05-01]. Dostupné z: http://www.dataversity.net/nosql-vs-sql-its-about-the-performance-and-scale/.

[Kovacs, 2016] Kovacs, Kristof.nazevMongoDB vs HBase comparison.In: Kristof Kovacs [online]. 2016 [cit. 2016-07-24]. Dostupné z: https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis.

[Kumar, 2016] Kumar, Girish.*Exploring the different types of NoSQL*.In: 3pillar global [online]. [cit. 2016-07-23]. Dostupné z: http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases.

[MongoDB, 2017] MongoDB.*MongoDB: documentation.*[online]. MongoDB, Inc, 2016 [cit. 2016-10-06]. Dostupné z: https://docs.mongodb.com/.

[Oracle, 2012] Oracle.*Your first cup: Differences between Java EE and Java SE*. Oracle [online]. 2012 [cit. 2017-02-12]. Dostupné z: http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html. .

[Panyko, 2013] Panyko, Tomáš.*NoSQL databáze*. České Budějovice, 2013. Bakalářská. Jihočeská univerzita v Českých Budějovicích. Vedoucí práce Hana Havelková. Dostupné z:http://theses.cz/id/1xkybg/BP.pdf.

[Rosenkrancová, 2016] Rosenkrancová, Martina.*Co jsou big data a k čemu jsou dobrá?* Objevit.cz [online]. 2016 [cit. 2017-02-20]. Dostupné z: http://objevit.cz/co-jsou-big-data-a-k-cemu-jsou-dobra-t157688.

[Rychlý, Kolář, 2013] Rychlý, Marek, Kolář, Dušan.*NoSQL databáze*.In: Vysoké učení technické v Brně [online]. Brno, 2013 [cit. 2016-07-23]. Dostupné z: http://www.fit.vutbr.cz/ rychly/public/docs/slides-nosql-databases/slides-nosql-databases.print.pdf.

[Sharma, S Tim, Gadia, Shandilya, Peddoju, 2014] Sharma, Sugam, S Tim, Udoyara, Gadia, Shashi, Shandilya, Ritu, Peddoju, Sateesh.*Classification and Comparison of NoSQL Big Data Models*.[online].

2014, , 27 [cit. 2016-07-23]. Dostupné z: http://web.cs.iastate.edu/ sugamsha/articles/Classification.

[TechFerry, 2015] *Part 1: Spring Annotations.*TechFerry [online]. [cit. 2017-03-18]. Dostupné z:http://www.techferry.com/articles/spring-annotations.html.

[Tikhanski, 2015] Tikhanski, Dmitri.*Open Source Load Testing Tools: Which One Should You Use?* BlazeMeter [online]. 2015 [cit. 2017-04-30]. Dostupné z: https://www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use.

[Webb, 2017] Webb, Phillip et. al.*Spring Boot Reference Guide*. Spring [online]. 2017 [cit. 2017-02-12]. Dostupné z: http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle.