Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

VYUŽITÍ NODE.JS A SPRING BOOT PŘI TVORBĚ REST API

Bakalářská práce

Autor: Vítek Vaníček
Studijní obor: Informační management

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Odborný konzultant: Ing. Rostislav Podmanický, vývojář informačních systémů

Hradec Králové duben 2022

Prohlášení:

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

V Hradci Králové dne 28.4.2022 Vítek Vaníček

Poděkování:

Rád bych vyjádřil poděkování vedoucímu bakalářské práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce, doporučení a cenné poznámky při zpracování práce. Děkuji také Ing. Rostislavu Podmanickému za odborné konzultace k vývoji, množství cenných rad, podnětů a doporučení.

Anotace

Cílem této práce je představit a nastínit možnosti využití aktuálních technologií při tvorbě API. Především se bude jednat o framework Spring Boot a Node.js. Každá z těchto technologií bude v práci představena, a nakonec porovnána. Porovnání budou zaměřena například na jednoduchost napojení API na nejpoužívanější databázové systémy (dostupnost ovladačů a podobně), náročnost instalace/spuštění a provozu API, a na samotnou implementaci – způsob a rychlost vývoje.

API bude sloužit pro obsluhu a zpracování požadavků z mobilních aplikací informačního systém pro malé a střední firmy, závěr práce obsahuje zdůvodnění volby dané technologie, která je pro vývoj použita.

Annotation

Title: Use of Node.js and Spring Boot for creating REST API

The aim of this thesis is to present and outline the possibilities of using current technologies for API development. In particular, the Spring Boot framework and Node.js will be discussed. Each of these technologies will be introduced in the thesis, and finally compared. The comparisons will focus, for example, on the ease of connecting the API to the most commonly used database systems (availability of drivers and so on), the difficulty of installing/running and operating the API, and the implementation itself – the method and speed of development.

The API will be used to serve and process requests from mobile applications for the information-economy system, the conclusion of the thesis contains a justification of the choice of the technology used for the development.

Obsah

Úvod 1

1 Cíl práce 2

2 Úvod do REST API 3

2.1 JSON 3

3 Charakteristika Spring Boot 7

3.1 Automatická konfigurace 7

3.2 Názorný přístup ke konfiguraci 7

3.3 Možnost vytvářet samostatné aplikace 8

3.4 Programovací jazyky 8

3.4.1 Java vs Kotlin 9

3.5 Sestavovací prostředí 9

3.6 JAR nebo WAR 11

3.7 Požadavky na spuštění & provoz 12

3.8 Anotace 13

3.8.1 @Controller nebo @RestController 13

3.8.2 @Service 13

3.8.3 @Repository 14

3.8.4 @Bean, @Autowired, @Component 14

3.8.5 Další anotace použitelné pro REST a MVC 16

3.9 Rozšíření 17

3.9.1 Generování dokumentů 18

3.9.2 Monitorování instancí 21

4 Charakteristika Node.js 25

4.1 Nedostatky 25

4.1.1 Request body mapping 25

4.1.2 Validace request body ... 26
5 Popis implementace a vývoje REST API ... 28
5.1 REST API ... 28
5.1.1 Stavové kódy ... 28
6 Napojení na databáze ... 29
6.1 Firebird databáze ... 29
6.2 Spring Boot ... 30
6.2.1 JDBC – Java Database Connectivity ... 30
6.2.2 Pooling ... 31
6.2.3 ORM – Hibernate – JPA (Java persistence API) ... 31
6.3 Node.js ... 34
7 Porovnání Spring Boot a Node.js ... 35
7.1 Prostředí vývoje ... 35
7.1.1 Spring Boot ... 35
7.1.2 Node.js ... 35
7.2 Prostředí pro běh API ... 36
7.3 Ukázka základní implementace endpointu ve Spring Boot ... 36
Závěr ... 38
Použité zdroje ... 39
Přílohy ... 41

Seznam obrázků

Obrázek 1 – Schéma tvorby JSON objektu (2) 4
Obrázek 2 – Schéma tvorby JSON pole (2) 4
Obrázek 3 – Srovnání vyhledávání klíčových slov JSON, XML, REST a SOAP na platformě Stack Overflow (3) 5
Obrázek 4 – Srovnání vyhledávání klíčových slov JSON a XML na platformě Google (4) 6
Obrázek 5 – Ukázka startovacích balíčků (zdroj: autor) 8
Obrázek 6 – Graf srovnání Maven vs Gradle (7) 10
Obrázek 7 – Ukázka Maven konfigurace v souboru pom.xml (zdroj: autor) 11
Obrázek 8 – Ukázka konfiguračního souboru Windows Service Wrapper (zdroj: autor) 12
Obrázek 9 – Ukázka anotace @RestController (zdroj: autor) 13
Obrázek 10 – Ukázka anotace @Service (zdroj: autor) 13
Obrázek 11 – Ukázka anotace @Repository (zdroj: autor) 14
Obrázek 12 – Ukázka anotace @Bean (zdroj: autor) 15
Obrázek 13 – Ukázka anotace @Autowired (zdroj: autor) 16
Obrázek 14 – Ukázka anotace @GetMapping (zdroj: autor) 16
Obrázek 15 – Ukázka anotace @RequestBody (zdroj: autor) 17
Obrázek 16 – Ukázka anotace @ResponseBody (zdroj: autor) 17
Obrázek 17 – Ukázka anotace @RequestParam (zdroj: autor) 17
Obrázek 18 – Ukázka anotace @PathVariable (zdroj: autor) 17
Obrázek 19 – Ukázka řešení generování PDF pomocí knihovny Thymeleaf (zdroj: autor) 19
Obrázek 20 – Ukázka použití knihovny Jaspersoft pro generování PDF (zdroj: autor) 20
Obrázek 21 – Ukázka definice fontů pro Jaspersoft (zdroj: autor) 21

Obrázek 22 Ukázka vynucené komunikace i bez validního SSL certifikátu (zdroj: autor) ... 22
Obrázek 23 – Ukázka přehledu informací ze Spring Boot Admin dashboardu (zdroj: autor) ... 23
Obrázek 24 – Ukázka informací o databázi ze SBA dashboardu (zdroj: autor) ... 24
Obrázek 25 – Ukázka validace request body Node.js (13) ... 26
Obrázek 26 – Ukázka anotace field pro validaci request body (zdroj: autor) ... 27
Obrázek 27 – Ukázka rozhraní JDBC (16) ... 30
Obrázek 28 – Ukázka definování ovladače pro JDBC a parametrů pro připojení k Firebird databázi (zdroj: autor) ... 31
Obrázek 29 – Hibernate – ukázka @Entity (zdroj: autor) ... 33
Obrázek 30 – Hibernate – ukázka @OneToMany (zdroj: autor) ... 34
Obrázek 31 – Ukázka implementace endpointu metody GET (zdroj: autor) ... 37

Seznam tabulek

Tabulka 1 – Vybrané parametry pro porovnání jazyků Java a Kotlin (6) ... 9
Tabulka 2 – Popis vybraných anotací pro REST a MVC (8) ... 16

Úvod

API (Application Programming Interface) je rozhraní, které umožňuje vzájemnou komunikace dvou aplikací/platforem za účelem výměny a zpracování dat. Je možné provozovat jak jen jednosměrnou, tak i obousměrnou komunikaci. Obecně se jedná o balík metod, tříd, funkcí a protokolů, na základě kterých probíhá následná komunikace = výměna uživatelských nebo systémových dat v předem známém formátu (nejčastěji JSON nebo XML).

API je téměř nepostradatelným prvkem, díky kterému funguje většina mobilních aplikací. Nalezneme jej však i na úrovni operačního systému nebo u webových služeb (například e-shop, který využívá API fakturačního/skladového systému a díky tomu může přijímat objednávky nebo zjišťovat stav daného produktu).

Správný návrh API a volba technologií je naprosto základní pro úspěšný vývoj, především u veřejných API pak není možné vzít nesprávně implementované rozhraní zpět. API by mělo být intuitivní (i bez dokumentace), ošetřené proti nesprávnému používání uživateli (API by mělo validovat vstupní data před jejich samotným zpracováním) a co nejvíce modularizované pro opakované použití. Také je důležitá zpětná kompatibilita a možnost dalšího rozšíření (přidání nových metod).

Práce se zaměřuje na dvě konkrétní technologie pro vývoj API – framework Spring Boot a Node.js. Obě technologie jsou založeny na trochu odlišném principu, vyžadují jiné přístupy při vývoji a následně různé podmínky pro samotný běh API, které budou v práci prozkoumány a vysvětleny.

Frameworky, ať už výše zmíněné nebo i jiné, se dnes používají velice často, jelikož vývojářům umožňují výrazně zkrátit dobu vývoje aplikace – mimo jiné totiž redukují nutnost psaní standardního kódu pro základní operace a místo toho lze využít a konfigurovat předpřipravené moduly, které jsou víceméně součástí frameworku.

1 Cíl práce

Cílem práce je představit, porovnat a zhodnotit využití technologií Spring Boot a Node.js při vývoji API pro mobilní aplikaci informačního systému určeného pro malé a střední firmy – API bude tedy muset splňovat bezpečnostní záležitosti vzhledem k povaze systému a samotných dat, dále bude muset být velice výkonné, jelikož předpokládaný počet současně připojených klientů je v desítkách – např. obchodní zástupci v terénu nebo pracovníci ve skladu. Především je třeba docílit co nejvyšší rychlosti vývoje společně s co nejmenší chybovostí v kódu samotného API – vývoj by se tedy měl převážně soustředit na problematiku samotného zpracování dat v obchodní logice a neměl by se naopak zaobírat implementací například parsování dat, zpracováním samotných requestů protokolem HTTP a podobně.

Dále by tato práce měla odpovědět na otázky týkajících se problematiky samotného vývoje (výběr programovacího jazyka, prostředí pro vývoj API), včetně otázek napojení na databázový systém Firebird (případně další databáze) a dostupnosti dalších externích knihoven/balíčků pro nejrůznější rozšíření API – například knihovny určené na generování PDF souborů dle předem připravených šablon (použitelné pro vygenerování faktury a podobně), knihovny pro odesílání e-mailů zákazníkům a tak dále.

Zásadním kritériem při výběru technologie pro API by také měla být schopnost spuštění API na cílovém zařízení – tedy zda bude API provozováno v cloudových službách typu Amazon Web Services, nebo naopak na on-premise infrastruktuře dané firmy. S tím se následně váže i parametr, zda je možné dané API provozovat na jakémkoliv systému, ať už se bude jednat o Windows Server nebo Linux, případně macOS.

Vzhledem k zamýšlenému použití a napojení na informační systém je zapotřebí mít možnost spuštění více instancí zároveň na jednom zařízení (serveru). Každou instanci je následně potřeba napojit na jinou cílovou databázi, například na testovací. Z tohoto důvodu bude kritériem i náročnost na samotné zdroje hostitelského zařízení, tedy zatížení CPU, RAM a využití disku.

2 Úvod do REST API

Pro komunikaci klientských aplikací (včetně prohlížeče) se serverem je možné využít REST API – jedná se o aplikační programové rozhraní, které splňuje specifická architektonická omezení, jako je bezstavová komunikace a data ukládaná do mezipaměti. Nejedná se o protokol ani standard. Ačkoli k rozhraní REST API lze přistupovat prostřednictvím řady komunikačních protokolů, nejčastěji se volá přes protokol HTTPS. (1)

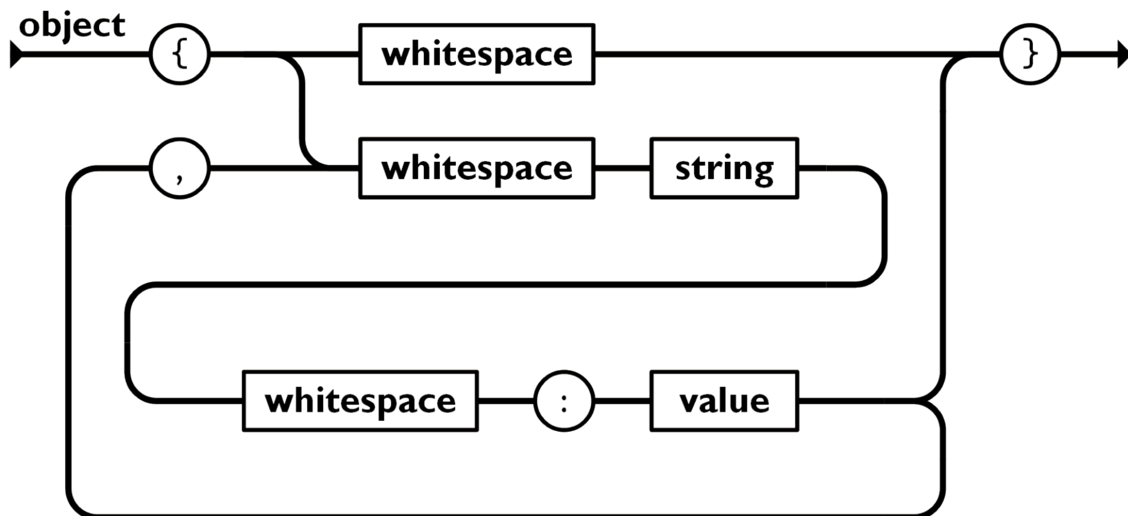
2.1 JSON

„JSON je odlehčený formát pro výměnu dat. Je jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově. Je založen na podmnožině programovacího jazyka JavaScript, Standard ECMA-262 3rd Edition – December 1999. JSON je textový, na jazyce zcela nezávislý formát, využívající však konvence dobře známé programátorům jazyků rodiny C (C, C++, C#, Java, JavaScript, Perl, Python a dalších).“ (2)

Vzhledem k široké podpoře standardu JSON (JavaScript Object Notation) pro přenos dat ve většině frameworků (vestavěné metody pro kódování a dekódování JSONu) začíná tento standard postupně převažovat nad XML, které se používá například v SOAP API (většinou je požíváno pro aplikace staršího data vývoje, které ještě nebyly nebo nemohou být z konkrétních důvodů převedeny pod REST API).

Aby si klientská aplikace rozuměna s API serverem a správně interpretovala obdržená data, je důležité, aby hlavička HTTP obsahovala informaci o typu obsahu – například *„Content-Type: application/json“*, a to jak při odesílání požadavku z klientské aplikace, tak i API při odpovědi na právě zaslaný požadavek. Dle této informace v HTTP hlavičce se aplikace pokusí rozkódovat data a případně dále provést automatické parsování do konkrétního objektu dle implementace.

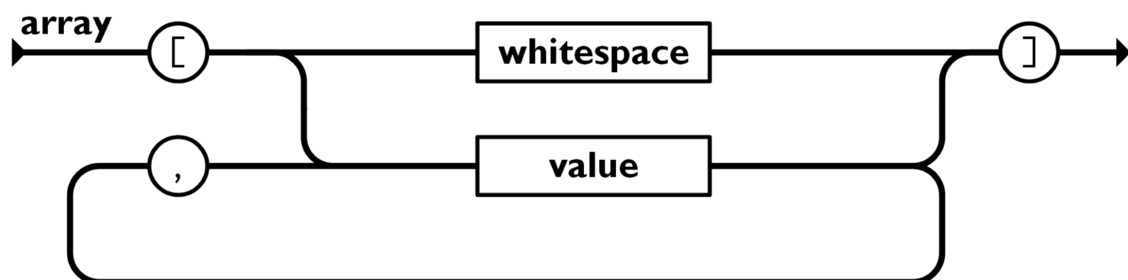
Struktura JSON objektu je pevně daná, napříč programovacími jazyky a prostředími – díky tomu je ideální pro komunikaci mezi odlišnými aplikacemi a skládá se podle následujících schémat, která jsou doplněna ukázkami.



Obrázek 1 – Schéma tvorby JSON objektu (2)

Jednoduchý výsledný JSON by mohl poté vypadat například takto:

```
{
  "version": 131,
  "verze": 131,
  "build": "dev",
  "build_number": 1310,
  "check_api_version": false,
}
```

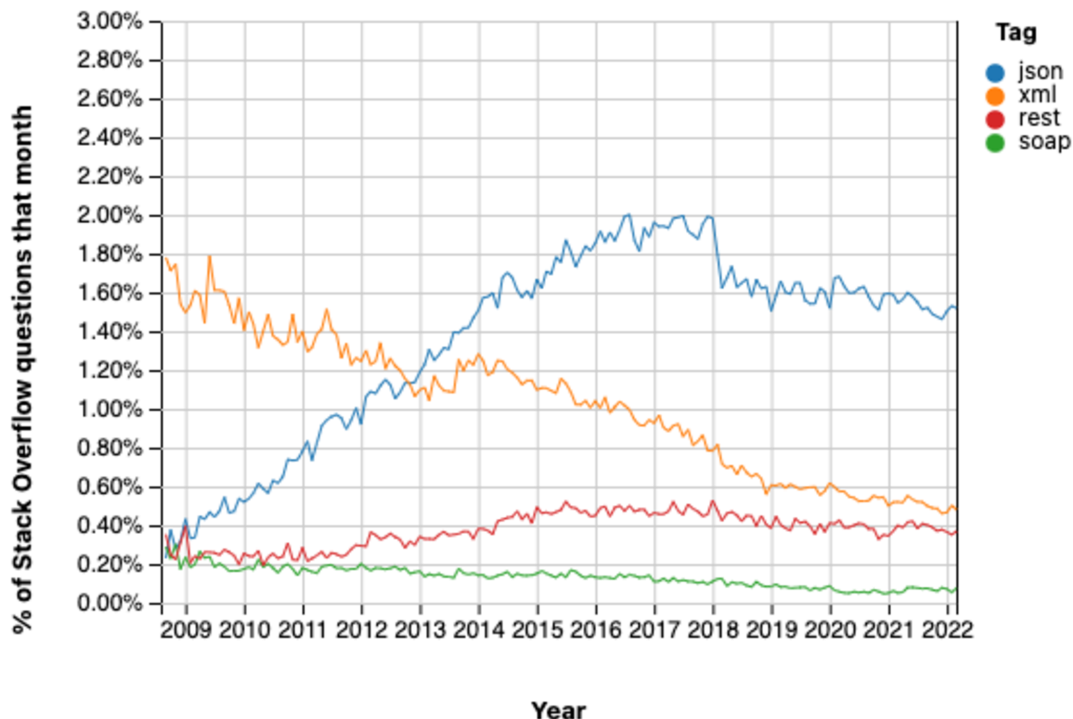


Obrázek 2 – Schéma tvorby JSON pole (2)

Příklad JSONu s daty ve formátu pole:

```
[
        {
          "oid": "00001454546",
          "nazev": "Artikl č. 1",
          "cena": 999.8
        },
        {
          "oid": "00001454547",
          "nazev": "Artikl č. 2",
          "cena": 249.9
        }

]
```

Níže je graf trendu vyhledávání klíčových slov „json“, „xml“ a „soap“ od roku 2009 na platformě Stack Overflow, na kterém je vidět jasný nárust formátu JSON a naopak pokles XML. Podobný trend je vidět i v porovnání REST vs SOAP API, kde taktéž dochází k poklesu ve prospěch REST API.



Obrázek 3 – Srovnání vyhledávání klíčových slov JSON, XML, REST a SOAP na platformě Stack Overflow (3)

Dále lze uvést porovnání trendu vyhledávání klíčových slov „json“ a „xml“ na platformě Google od roku 2004 (celosvětově). Z těchto dat se dá následně usoudit, že formát JSON začíná postupně převažovat nad XML formátem (využívaným spíše pro SOAP API).

Obrázek 4 – Srovnání vyhledávání klíčových slov JSON a XML na platformě Google (4)

3 Charakteristika Spring Boot

Open source framework Spring Boot založený na Javě je určen pro vytváření samostatných aplikací, které jsou následně provozovány na JVM (Java Virtual Machine).

Tento framework urychluje a usnadňuje vývoj webových aplikací a mikroslužeb s rozhraním Spring Framework prostřednictvím tří základních funkcí, díky kterým je možné vytvořit aplikaci s minimální konfigurací a nutnosti implementace základních funkcí (5):

3.1 Automatická konfigurace

Komponenty – balíčky a závislosti jsou automaticky ve výchozím stavu nakonfigurovány pro nejběžnější a nejpoužívanější způsoby použití. Zabraňuje to tedy možnosti zanesení chyb z neznalosti – opomenutí nutnosti nastavení konkrétních parametrů, a snižuje tedy možnost zavlečení chyby, třeba i bezpečnostní (pro cílové použití by to mohlo být kritické, vzhledem k širokému nasazení).

3.2 Názorný přístup ke konfiguraci

Základní balík závislostí je určen tzv. „startovacími balíčky", které obsahují běžné minimum závislostí potřebné pro danou oblast. Jako příklad lze uvést *„spring-boot-starter-security"* potřebný pro zprovoznění základního zabezpečení aplikace, nebo *„spring-boot-starter-mail"* pro zpracování e-mailů (například odesílání přes protokol SMTP).

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Obrázek 5 – Ukázka startovacích balíčků (zdroj: autor)

3.3 Možnost vytvářet samostatné aplikace

Finální package je možné okamžitě spustit na jakémkoliv JVM (dle minimální verze) a není nutné instalovat další podpůrná běhová prostředí a podobně.

V obecné rovině je Spring Boot víceméně rozšířením frameworku Spring o nejrůznější závislosti/balíčky a poskytuje velké množství předem připravených konfigurací a nástrojů, který by jinak musel vývojář programovat a jedná se tak o jeden z nejjednodušších způsobů vývoje aplikace (API). Díky vestavěnému webovému serveru (ve výchozím nastavení Apache Tomcat) je Spring Boot ideálním pro vývoj REST API, ať už jde o veřejné API nebo určené pouze pro konkrétní použití a okruh uživatelů.

3.4 Programovací jazyky

Pro vývoj ve Spring Boot lze standardně využít programovací jazyky Java, Kotlin nebo Groovy. Například Kotlin má plnou podporu Spring Boot, a navíc má kompaktnější a úspornější syntaxi než Java s výrazně menší šancí na chybovost v kódu způsobenou samotným autorem kódu. Vývojář má tak skvělou možnost výběru jazyka dle svého uvážení, preferencí a zkušeností.

3.4.1 Java vs Kotlin

Tabulka 1 – Vybrané parametry pro porovnání jazyků Java a Kotlin (6)

	Java	Kotlin
Null Safe	Je potřeba ošetřit veškeré možné null hodnoty, jinak v případě přístupu k neočekávané null hodnotě dojde k vyvolání výjimky „NullPointerException“.	Vyřešeno – každá proměnná určuje, zda je možné vůbec přiřadit null hodnotu, díky tomu je možné provést kontrolu již při kompilaci.
Smart Casts	Vždy je potřeba explicitně určit datový typ proměnné již při deklaraci.	Na základě podmínek se kompilátor pokusí automaticky určit datový typ a umožnit tak přístup k proměnné.
Data Classes	Za účelem uchovávání dat a práce s nimi musíme definovat konstuktory, metody get, set a další funkce.	Při označení třídy jako „data class“ kompilátor automaticky vygeneruje příslušné funkce a konstruktor.
Getters & Setters	Je nutné ručně implementovat pro každou proměnnou.	Kompilátor automaticky interně tyto metody definuje (pro proměnné typu „val“ jen getter).

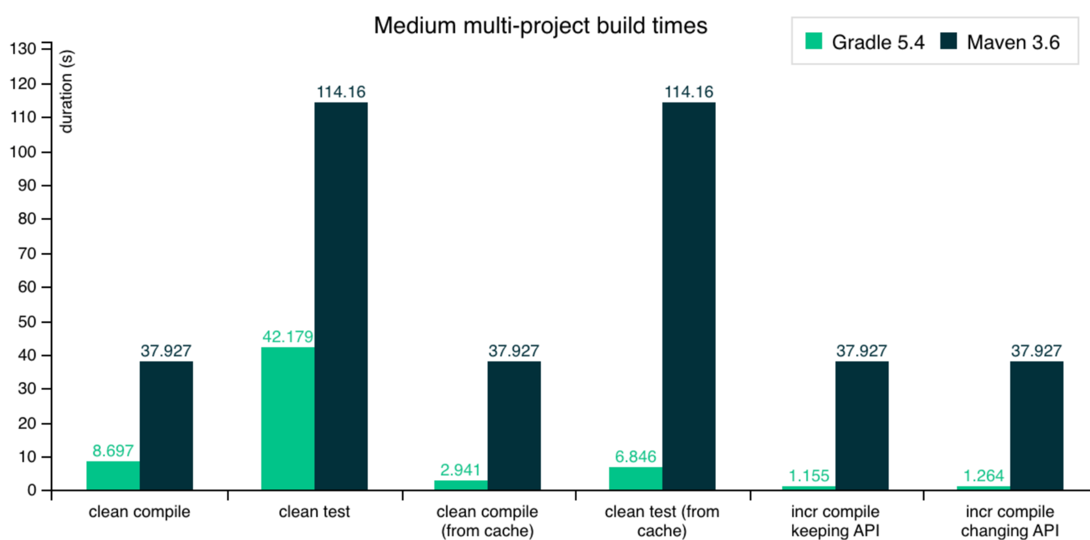
3.5 Sestavovací prostředí

Vývojář má možnost si vybrat, zda využije pro automatizaci sestavování (kompilaci) jeho aplikace, software Gradle nebo Maven. Ty zajišťují kompletní sestavení (build) aplikace se všemi potřebnými závislostmi (balíčky) do finálního souboru dle využití do JAR nebo WAR formátu.

Kromě jiného se výše uvedený software liší v zápisu konfiguračního souboru projektu – Maven využívá zápis do formátu XML (soubor „pom.xml“), a naopak Gradle (více zaměřený na flexibilitu a výkon) může být psát například v Javě, Kotlinu nebo v Gradle, jelikož je založen na DSL (Domain Specific Language), výsledný soubor může být pojmenován například „build.gradle.kts“.

Rychlejšího sestavení dosahuje software Gradle, a to hned z několika důvodů. Využívá inkrementální kompilace, které zajišťují že jsou zkompilovány jen soubory, které vývojář od poslední kompilace upravil. Dále využívá tzv. „Gradle Build Cache", díky které není nutné kompilovat stejné části například při změně větve – Gradle využívá výstupy z předchozích sestavení, aby zajistil rychlejší kompilaci. (7)

Z níže uvedeného grafu je vidět, že je Gradle oproti Mavenu 4x-5x rychlejší pro kompletní sestavení, což v rámci denního vývoje je velice zásadní časová a finanční úspora, která se může projevit ve finálních nákladech na vývoj.



Obrázek 6 – Graf srovnání Maven vs Gradle (7)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.6</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.signys</groupId>
    <artifactId>SignysLogisticsAPI</artifactId>
    <version>1.3.2</version>
    <packaging>jar</packaging>
    <name>SignysLogisticsAPI</name>
    <description>API pro mobilní aplikaci Signys Logistics</description>

    <properties>
        <java.version>1.8</java.version>
        <kotlin.version>1.6.20</kotlin.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <skipTests>true</skipTests>
        <start-class>com.signys.logisticsAPI.SignysLogisticsApiKt</start-class>
        <mainClass>com.signys.logisticsAPI.SignysLogisticsApiKt</mainClass>
    </properties>

    <repositories>
        <repository>
            <id>maven-repository</id>
            <name>Maven Repository</name>
            <url>https://repo1.maven.org/maven2</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
        </dependency>
```

Obrázek 7 – Ukázka Maven konfigurace v souboru pom.xml (zdroj: autor)

3.6 JAR nebo WAR

Spring Boot a sestavovací prostředí dokáže sestavit jak samostatně spustitelnou kompletní aplikaci, tzv. JAR formát, který obsahuje všechny potřebné závislosti včetně aplikačního kontejneru (ve výchozím nastavení se jedná o Tomcat).

Naopak sestavení do tzv. WAR formátu se použijte tehdy, když daná aplikace bude spouštěna na již existujícím aplikačním serveru a není tedy třeba aby obsahovala vlastní aplikační kontejner. Z toho plyne, že sestavení do WAR bude mít násobně menší velikost a náročnost na sestavení.

3.7 Požadavky na spuštění & provoz

Výslednou aplikaci (.jar) je možné spustit v jakémkoliv JVM (Java Virtual Machine), ať už bude hostitelský systém založen na platformě Windows, Linux nebo macOS.

Verze JVM, potažmo JRE (Java Runtime Environment), musí být vždy minimálně stejná jako je definováno v samotné aplikaci, respektive v sestavovacím prostředí, avšak může být i vyšší. Aplikaci sestavenou pro Java verze 8 můžeme spustit jak na verzi 8, tak i vyšší – 11, 14, 16 a podobně.

Vzhledem nejen k licenční politice společnosti Oracle produktu Java Development Kit (pro komerční využití JDK) lze využít nejrůznější open source varianty OpenJRE a OpenJDK – například od skupiny Adoptium (odkaz: https://adoptium.net/).

Pro spuštění API na serverech s OS Windows byl použit *Windows Service Wrapper* (odkaz: https://github.com/winsw/winsw). Samozřejmě není problém spustit výsledný JAR soubor přímo s pomocí JRE, ale díky tomuto podpůrnému wrapperu je možné docílit spouštění API na serveru jako služby, a tedy například využít automatického startu, možnosti doplnění některých konfiguračních parametrů nutných pro start API, anebo využít automatické ověření, a případné stažení nové verze z HTTP serveru před každým startem API. Jedním z nastavení Windows Service Wrapperu je i určení způsobu ukládání logů ze samotného API do souborů.

```xml
<service>
  <id>_SignysLogisticsAPI</id>
  <name>_Signys Logistics API</name>
  <description>API server pro aplikaci Signys Logistics</description>
  <download from="https://testovaciserver.cz/SignysLogisticsAPI.jar" to="%BASE%\SignysLogisticsAPI.jar"
                auth="basic" user="xxx" password="xxx"/>
  <executable>java</executable>
  <arguments>
    -DSGA_PORT="443"
    -DSGA_FILES_PATH="D:/dokumenty"
    -DSGA_REPORTS_PATH="D:/reporty"
    -DTZ="Europe/Prague"
    -jar "%BASE%\SignysLogisticsAPI.jar"
  </arguments>
  <logpath>%BASE%\logs-wrapper</logpath>
  <log mode="roll-by-time">
       <autoRollAtTime>00:00:00</autoRollAtTime>
       <zipOlderThanNumDays>7</zipOlderThanNumDays>
       <pattern>yyyy-MM-dd</pattern>
  </log>
</service>
```

Obrázek 8 – Ukázka konfiguračního souboru Windows Service Wrapper (zdroj: autor)

3.8 Anotace

Spring Boot využívá rozsáhlý soubor anotací, uplatňujících se na třídy, jejich metody a podobně. Anotace můžeme využít také například pro automatické generování dokumentace, a to pomocí knihovny OpenAPI 3.0, kdy u každého endpointu můžeme definovat tagy, názvy parametrů, a v případě data class pro request se zobrazí i tyto informace. Za účelem zprovoznění základního API s běžnými metodami se setkáme nejčastěji s následujícími anotacemi:

3.8.1 @Controller nebo @RestController

Označuje třídu, která se stará o obsluhu webových požadavků, většinou je uvedena společně s anotací *@RequestMapping*, která určuje relativní URL adresu vůči rootu aplikace, na které jsou přijímány požadavky a kde dané endpointy v třídě naslouchají.

```
Vítek Vaníček
@Tag(name = "FILES", description = "Endpoint pro práci se soubory")
@RestController
@RequestMapping(value = ["v1/files", "v1/download"])
class FilesController: AppContext() {
```

Obrázek 9 – Ukázka anotace @RestController (zdroj: autor)

3.8.2 @Service

Označuje třídu, která se stará o business logiku API a jeho funkcí. Tato třída následně pracuje s datovou vrstvou. Vzhledem k uvedenému použití se jedná o naprosto základní anotaci používanou pro veškerou obchodní logiku.

```
@Service
class ArtiklService: AppContext() {

    Vítek Vaníček
    fun getArtiklDet(oidcenktgm01: String, tryExists: Boolean = true): ArtiklDetail {
        if(tryExists) {
            if(!artiklDetail.existsArtikl(oidcenktgm01)) {
                throw ValidationException("Artikl $oidcenktgm01 neexistuje")
            }
        }
```

Obrázek 10 – Ukázka anotace @Service (zdroj: autor)

3.8.3 @Repository

Označuje třídu, která přímo zasahuje do samotné databáze a provádí veškeré potřebné úkony s ní spojené. Jedná se tedy konkrétně například o provádění dotazů SELECT na databázi, aktualizaci existujících záznamů a podobně. Zde používáme zkratku *DAO* (Data Access Object).

Možností repositáře je i definování vlastních nejrůznějších podmínek, které se doplňují například právě do daného SELECT SQL příkazu – níže je ukázka používání anotace pro funkci na vyhledávání shody v rámci označení skladových míst v informačním systému. Díky využití funkce „lower" je vyhledávání case-insensitive a navíc je možné zadat jen začátek řetězce (použití operátoru „like", a „%" na konci řetězce).

```
// Vyhledávání v seznamu skladových míst
± Vítek Vaníček
@Repository
class SkladovaMistaVyhledavaniRepository {
    @PersistenceContext
    var em: EntityManager? = null

    ± Vítek Vaníček
    fun vyhledatSkladoveMisto(request: SkladovaMistaRequest): Set<SkladoveMistoDetail>? {
        val cb = em!!.criteriaBuilder
        val cq: CriteriaQuery<SkladoveMistoDetail> = cb.createQuery(SkladoveMistoDetail::class.java)
        val master: Root<SkladoveMistoDetail> = cq.from(SkladoveMistoDetail::class.java)

        // podmínky
        val predicates: ArrayList<Predicate> = ArrayList()
        if(request.skmhala.isNotEmpty())       { predicates.add(cb.equal(master.get<Any>("SKMHALA"), request.skmhala))     }
        if(request.skmregal.isNotEmpty())      { predicates.add(cb.equal(master.get<Any>("SKMREGAL"), request.skmregal))   }
        if(request.skmpatro != null)           { predicates.add(cb.equal(master.get<Any>("SKMPATRO"), request.skmpatro))   }
        if(request.skmokno != null)            { predicates.add(cb.equal(master.get<Any>("SKMOKNO"), request.skmokno))     }
        if(request.pstatus != null)            { predicates.add(cb.equal(master.get<Any>("PSTATUS"), request.pstatus))     }
        if(request.vyhledavani.isNotEmpty())   {
            predicates.add(cb.like(cb.lower(master.get("KODSKLMISTA")), "${request.vyhledavani.lowercase()}%"))
        }

        // FINAL podmínky
        predicates.add(cb.equal(master.get<Any>("PSTORNO"), 0))
        cq.where(cb.and(*predicates.toTypedArray()))

        cq.orderBy(cb.asc(master.get<Any>("KODSKLMISTA")))
        return em!!.createQuery(cq).resultList.toSet()
    }
}
```

Obrázek 11 – Ukázka anotace @Repository (zdroj: autor)

3.8.4 @Bean, @Autowired, @Component

Oproti předchozím anotacím, je *@Bean* anotace aplikovatelná na metody, a nikoliv na třídy. Konkrétně kompilátoru říká, že má danou metodu provést a její návratovou hodnotu uložit do kontextu aplikace, který je spravován právě Springem. Využívání

@Bean anotace je speciální přístup zvaný „dependency injection", v českém překladu tedy „vstřikování závislostí", a umožňuje nám používat metody a třídy (ty například pomocí anotace *@Component*) kdekoliv v aplikaci, respektive prostředí Spring Boot. Nemusíme se starat o vytvoření jejich instance. Anotace *@Autowired* pak následně prohledá kontext aplikace a nalezne požadovanou závislost pro další využití v kódu. Metody označené touto anotací se za normálních podmínek zavedou při startu aplikace.

Níže je ukázka použití anotace pro základní inicializaci nastavení komponenty pro práci s e-mailovými službami. Jedná se tedy o nastavení konkrétních hodnot pro SMTP komunikaci, v tomto případě jde o hodnoty načtené z databáze při startu API.

```
// SMTP inicializace
Vítek Vaníček
@Bean
@DependsOn("main")
fun getJavaMailSender(): JavaMailSender {
    val mailSender = JavaMailSenderImpl()
    mailSender.host              = globCfg.SMTP_HOST
    mailSender.port              = globCfg.SMTP_PORT
    mailSender.username          = globCfg.SMTP_USERNAME
    mailSender.password          = globCfg.SMTP_PASSWORD
    mailSender.protocol          = "smtp"
    mailSender.defaultEncoding   = "UTF-8"

    val props = mailSender.javaMailProperties
    props["mail.transport.protocol"]        = "smtp"
    props["mail.smtp.auth"]                 = globCfg.SMTP_USERNAME.isNotEmpty()
    props["mail.smtp.starttls.enable"]      = globCfg.SMTP_AUTOTLS
    props["mail.smtp.connectiontimeout"]    = 5000
    props["mail.smtp.timeout"]              = 3000
    props["mail.smtp.writetimeout"]         = 5000
    props["mail.debug"]                     = SGA_DEV_MAIL
    return mailSender
}
```

Obrázek 12 – Ukázka anotace @Bean (zdroj: autor)

Níže je ukázka použití anotace *@Autowired*, kdy do proměnné „globCfg" žádáme Spring Boot o vložení závislosti na třídu „GlobalConfig", která je inicializována při startu (jedná se o *@Component*) a v daný okamžik dostupná v kontextu aplikace. Vzhledem k nucené posloupnosti inicializace je navíc použita anotace *@Lazy*, která při zavádění aplikace dočasně odsune vložení závislosti do proměnné „globCfg".

```kotlin
// Export dat do routovacích systémů - např. TASHA
Vítek Vaníček
@Repository
class RozvozExportRepository {
    @Lazy
    @Autowired
    lateinit var globCfg: GlobalConfig

    @PersistenceContext
    var em: EntityManager? = null
```

Obrázek 13 – Ukázka anotace @Autowired (zdroj: autor)

3.8.5 Další anotace použitelné pro REST a MVC

Pro specifikaci jednotlivých metod ve třídě anotované *@RestController* (nebo *@Controller*) můžeme využít například anotace uvedené níže v tabulce. Jedná se především o anotace umožňující zpracování příchozích požadavků a konkretizace stylu odpovědi. Můžeme jimi například vynucovat některé parametry, dále je možné anotacemi a příslušnými závislostmi například zajistit kontrolu délky a obsahu vstupních dat (z request body).

Tabulka 2 – Popis vybraných anotací pro REST a MVC (8)

<table>
<tr><th>Označení</th><th>Popis</th></tr>
<tr><th colspan="2">Příklad</th></tr>
<tr><td>@GetMapping</td><td>Mapuje požadavky GET na konkrétní metodu dle URI</td></tr>
<tr><td colspan="2">@Operation(summary = "Načtení nového globálního nastavení API", security = [(SecurityRequirement(name = "basicAuth"))])
@ApiResponses(value = [ApiResponse(responseCode = "200", description = "OK", content = [Content()])])
@GetMapping("/load-settings", produces = [MediaType.TEXT_PLAIN_VALUE])
fun loadGlobalSettings(): ResponseEntity<Any> {
    globCfg.main()
    return ResponseEntity.ok(body: "OK - globální nastavení načteno")
}
Obrázek 14 – Ukázka anotace @GetMapping (zdroj: autor)</td></tr>
<tr><td>@RequestBody</td><td>Spojuje data z HTTP požadavku s parametrem metody (automatické parsování JSON)</td></tr>
</table>

```
@Operation(summary = "Generování reportu - PDF", security = [(SecurityRequirement(name = "basicAuth"))
@ApiResponses(value = [
    ApiResponse(responseCode = "200", description = "OK", content = [(Content(mediaType = "application,
    ApiResponse(responseCode = "400", description = "Nenalezen / chyba", content = [Content()])])
@PostMapping("/pdf")
fun postReport(@RequestBody request: ReportRequest,
               httpRequest: HttpServletRequest, response: HttpServletResponse): ResponseEntity<Any> {
    val userProfile = uzivatelService.getUserProfile(SecurityContextHolder.getContext().authentication
```

Obrázek 15 – Ukázka anotace @RequestBody (zdroj: autor)

@ResponseBody	Návratová hodnota z metody je zároveň tělem odpovědi na požadavek – dojde k převodu objektu do JSON (dle nastavení)

```
@ResponseBody
@ExceptionHandler(value = [Exception::class])
fun handleGeneralException(exception: Exception, httpRequest: HttpServletRequest): I
    val userProfile = uzivatelService.getUserProfile(SecurityContextHolder.getConte:
```

Obrázek 16 – Ukázka anotace @ResponseBody (zdroj: autor)

@RequestParam	Spojuje parametr dotazu z URL adresy

```
@GetMapping("")
@Throws(IOException::class)
fun getDownload(@Parameter(description = "OIDDOCEVDM01")
                @RequestParam("soubor", defaultValue = "") oiddocevdm01: String): ResponseEntity<Any> {
```

Obrázek 17 – Ukázka anotace @RequestParam (zdroj: autor)

@PathVariable	Extrahuje hodnotu parametru z adresy URL určeným klíčovým názvem

```
@PutMapping("/pozice/{id}", produces = [MediaType.APPLICATION_JSON_VALUE])
fun putPozice(@Parameter(description = "OIDCENKTGM01") @PathVariable("id") id: String,
              @RequestBody request: ArtiklUpravaPoziceRequest, httpRequest: HttpServletRequest): ResponseEntity<Any> {
    val userProfile = uzivatelService.getUserProfile(SecurityContextHolder.getContext().authentication.name)
    return ResponseEntity.ok(artiklService.zmenaPozice(id, request, userProfile, httpRequest))
}
```

Obrázek 18 – Ukázka anotace @PathVariable (zdroj: autor)

3.9 Rozšíření

Během samotného vývoje API pro informační systém, bylo nutné implementovat několik dalších rozšíření funkčnosti samotného API – nejvýraznějšími funkcí je možnost generování dokumentů (PDF) dle předem připravených šablon s daty z databáze a monitoring všech instancí včetně základních informací o nich.

3.9.1 Generování dokumentů

Požadavkem bylo generování sestavy do formátu PDF podle předem definované šablony – nejprve bylo vybráno řešení generování pomocí HTML šablony s přímým předáváním dat do sestavy z API, a to konkrétně s použitím balíčku „*Thymeleaf*“. Během prvních několika nasazení u klientů se ukázalo, že toto řešení není vyhovující vzhledem k velkému množství různorodých požadavků na datové složení sestav – každý klient a každá sestava vyžaduje různá data a není možné tyto požadavky obsloužit centrálním předávání dat z API.

Implementovalo se tedy řešení s návrhářem sestav „*Jaspersoft Studio*“ (založený na Javě), který umožňuje díky předání připojení k databázi vykonat při generování sestavy samostatně definované SQL příkazy k načtení konkrétních dat pro danou sestavu. Z API tedy dochází jen k předání základních informací – parametrů, jako je například unikátní ID pro doklad nebo uživatelské informace (obsahující podpisový alias uživatele, osobní číslo atd.), a samotného připojení uvolněného po dobu generování sestavy z poolu. Je možné používat jak základní sestavu ve formátu „jrxml“, kde dojde při každém načtení šablony ke kompilaci, nebo již formát „jasper“, který je již zkompilován a generování je následně mnohonásobně rychlejší.

Tato funkce se používá pro generování produktových štítků tisknuté na tzv. štítkové tiskárny s různými rozměry a pak i pro generování celých A4 dokladů (dodací listy, faktury a podobně) okamžitě odesílaných e-mailem zákazníkům. Výhodou je naprostá datová nezávislost na poskytnutých datech z API.

```kotlin
// Převod na PDF
val file: File = File.createTempFile( prefix: "temp",  suffix: ".pdf")
val outputStream: OutputStream = FileOutputStream(file)
val renderer = ITextRenderer()

val fontpath = "fonts/Arial/Arial-Bold.ttf".replace( oldValue: "/", File.separator)
renderer.fontResolver.addFont(ClassPathResource(fontpath).path, BaseFont.IDENTITY_H, BaseFont.NOT_EMBEDDED)

val fontpath2 = "fonts/Arial/Arial.ttf".replace( oldValue: "/", File.separator)
renderer.fontResolver.addFont(ClassPathResource(fontpath2).path, BaseFont.IDENTITY_H, BaseFont.NOT_EMBEDDED)
renderer.sharedContext.replacedElementFactory = BarcodeReplacedElementFactory(renderer.outputDevice)

// render první stránky
renderer.setDocumentFromString(htmls[0])
renderer.layout()
renderer.createPDF(outputStream,  finish: false)

// render dalších stránek PDF
for(i in 1 ≤  until  < htmls.size) {
    renderer.setDocumentFromString(htmls[i], ClassPathResource( path: "/templates/").url.toExternalForm())
    renderer.layout()
    renderer.writeNextDocument()
}

// dokončení renderu PDF
renderer.finishPDF()
outputStream.close()
file.deleteOnExit()
```

Obrázek 19 – Ukázka řešení generování PDF pomocí knihovny Thymeleaf (zdroj: autor)

```kotlin
// Načtení template
val report: JasperReport = loadTemplate(absolutniCesta)
var filledReport: JasperPrint

if(listOID.isNotEmpty()) {
    // Inicializace PDFMergerUtility class
    val pdfMerger = PDFMergerUtility()
    val pdfDocOutputstream = ByteArrayOutputStream()
    pdfMerger.destinationStream = pdfDocOutputstream

    // pro každé OID v listu vygenerovat vlastní sestavu
    listOID.forEach { it: MutableMap<String, Any>
        // Soubor PDF
        val pdfFile = File.createTempFile( prefix: "temp",  suffix: ".pdf")

        // Vyplnění template a následný render do PDF
        filledReport = JasperFillManager
            .fillReport(report, it,
                        DataSourceUtils.getConnection(
                            (applicationContext.getBean("dataSource") as DataSource)))

        // export a zápis do PDF
        pdfFile.writeBytes(JasperExportManager.exportReportToPdf(filledReport))

        // přidání zdrojových souborů
        pdfMerger.addSource(pdfFile)
    }

    // Sloučení
    pdfMerger.mergeDocuments( memUsageSetting: null)

    // Zápis finálního souboru
    finalFile.writeBytes(pdfDocOutputstream.toByteArray())
} else {
    // Vyplnění template a následný render do PDF
    filledReport = JasperFillManager
        .fillReport(report, parameters,
                    DataSourceUtils.getConnection(
                        (applicationContext.getBean("dataSource") as DataSource)))

    // export a zápis do PDF
    finalFile.writeBytes(JasperExportManager.exportReportToPdf(filledReport))
}
```

Obrázek 20 – Ukázka použití knihovny Jaspersoft pro generování PDF (zdroj: autor)

Při implementaci obou řešení bylo nutné vyřešit fonty, které nemusejí být v cílovém JVM dostupné. Jako nejlepší řešení se ukázalo přibalit pár vybraných, nejpoužívanějších fontů přímo do samotného balíčku API a dále napřímo definovat

cestu k nim. Pro balíček *Jaspersoft* se tyto cesty k souborům TTF definují ve specifickém souboru „fonts.xml“, který vypadá následovně:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fontFamilies>
    <fontFamily name="Arial">
        <normal><![CDATA[fonts/Arial/Arial.ttf]]></normal>
        <bold><![CDATA[fonts/Arial/Arial-Bold.ttf]]></bold>
        <italic><![CDATA[fonts/Arial/Arial-Italic.ttf]]></italic>
        <boldItalic><![CDATA[fonts/Arial/Arial-Bold-Italic.ttf]]></boldItalic>
        <pdfEmbedded><![CDATA[true]]></pdfEmbedded>
        <pdfEncoding>Cp1250</pdfEncoding>
        <exportFonts/>
    </fontFamily>

    <fontFamily name="Arial Black">
        <normal><![CDATA[fonts/ArialBlack/Arial-Black.ttf]]></normal>
        <bold><![CDATA[fonts/ArialBlack/Arial-Bold.ttf]]></bold>
        <italic><![CDATA[fonts/ArialBlack/Arial-Italic.ttf]]></italic>
        <boldItalic><![CDATA[fonts/ArialBlack/Arial-Bold-Italic.ttf]]></boldItalic>
        <pdfEmbedded><![CDATA[true]]></pdfEmbedded>
        <pdfEncoding>Cp1250</pdfEncoding>
        <exportFonts/>
    </fontFamily>

    <fontFamily name="Segoe UI">
        <normal><![CDATA[fonts/SegoeUI/SegoeUI.ttf]]></normal>
        <bold><![CDATA[fonts/SegoeUI/SegoeUI-Bold.ttf]]></bold>
        <italic><![CDATA[fonts/SegoeUI/SegoeUI-Italic.ttf]]></italic>
        <boldItalic><![CDATA[fonts/SegoeUI/SegoeUI-Bold-Italic.ttf]]></boldItalic>
        <pdfEmbedded><![CDATA[true]]></pdfEmbedded>
        <pdfEncoding>Cp1250</pdfEncoding>
        <exportFonts/>
    </fontFamily>
</fontFamilies>
```

Obrázek 21 – Ukázka definice fontů pro Jaspersoft (zdroj: autor)

3.9.2 Monitorování instancí

V rámci přehledu, sledování a následné správy rozrůstajících se instancí API bylo nutné vyřešit otázku monitoringu. Po zvážení všech variant (různá cloudová řešení) bylo zvoleno víceméně lokální řešení určené přímo pro Spring Boot – a to konkrétně balíček *Spring Boot Admin* (zkratkou SBA). Jeho instance je provozována na vlastním firemním serveru a do API byla implementována jen jeho klientská část. Oproti cloudovému řešení to má hned několik výhod: automatická registrace a deregistrace

instancí, okamžitě zobrazitelné přehledy konkrétní instance včetně systémových parametrů a statistik využití zdrojů serveru. Jedná se o neplacené řešení, umožňuje jednoduché definování vlastní parametrů (tagů) zobrazených u instancí v centrálním přehledu.

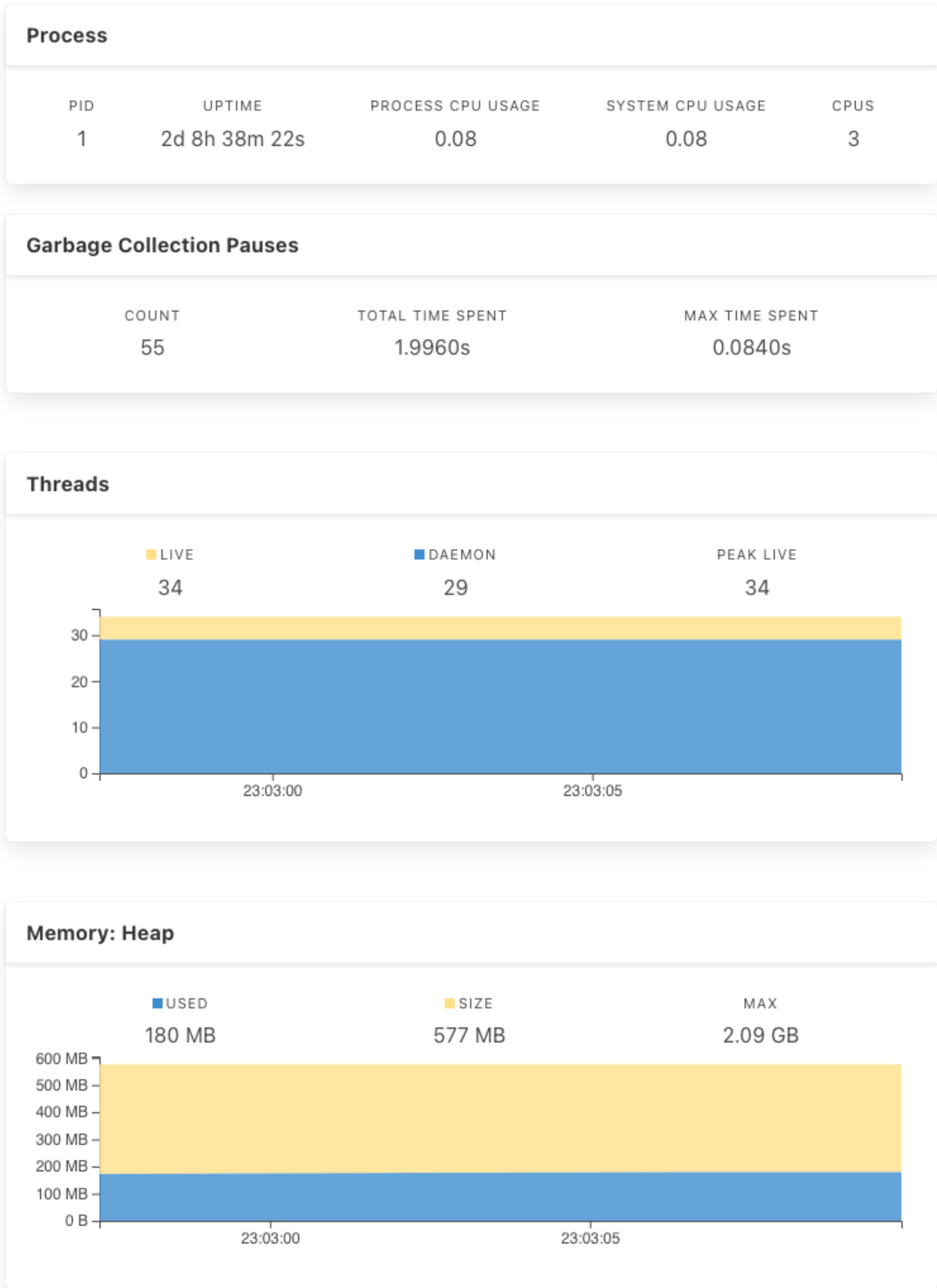
Největším problémem bylo obejití kontroly validace SSL certifikátu při přístupu ze serveru na klienty (nevalidní self-signed certifikáty na IP adresách), k čemuž bylo nutné přidat vlastní konfiguraci HTTP klienta na serverové straně, jelikož samotná implementace SBA nenabízí tuto možnost v základu například formou parametru.

```
@Configuration
class CustomHttpClientConfig {

    Vítek Vaníček
    @Bean
    @Throws(SSLException::class)
    fun customHttpClient(): ClientHttpConnector {
        val sslContext = SslContextBuilder
            .forClient()
            .trustManager(InsecureTrustManagerFactory.INSTANCE)
            .build()
        val httpClient: HttpClient = HttpClient.create().secure { ssl -> ssl.sslContext(sslContext) }

        return ReactorClientHttpConnector(httpClient)
    }
}
```

Obrázek 22 Ukázka vynucené komunikace i bez validního SSL certifikátu (zdroj: autor)

Mezi základní nastavení SBA patří mimo jiné i intervaly pro zjišťování stavu, které fungují tak, že instance API se nejprve ohlásí nadefinovanému monitoring serveru, kde provede registraci (pošle základní data). Následně serverová část provádí v předem definovaných intervalech requesty na health URI dané instance a tyto informace zobrazuje v dashboardu monitoringu. Získaná data mají nastavenou konkrétní životnost, po jak dlouhou dobu jsou považovány za platné.

Monitoring instancí obsahuje například: informace o buildu a stavu databáze, využití CPU, RAM, nastavení environment parametrů, přehled beans a nastavení aplikace včetně balíčků, naplánované úlohy, log, endpointy a jejich URI včetně metody a další. Nejrůznější informace lze následně speciálně vytáhnout z metrik – informace o Tomcat, HikariCP a podobně. V případě otevření detailu dané instance jsou tyto informace získávány téměř „realtime" s minimálním zpožděním.



Obrázek 23 – Ukázka přehledu informací ze Spring Boot Admin dashboardu (zdroj: autor)

Health	
Instance	UP
db	UP
database	Firebird 4.0
validationQuery	isValid()
diskSpace	UP
total	82 GB
free	14.3 GB
threshold	10.5 MB
exists	true
ping	UP

Obrázek 24 – Ukázka informací o databázi ze SBA dashboardu (zdroj: autor)

Dané řešení není naprosto dokonalé, avšak pokrývá většinu potřeb (poměr náklady na implementaci/užitek) a umožňuje efektivně monitorovat všechny instance s včetně těch nejdůležitějších informací k identifikaci. Podmínkou je zpřístupněný port instance API z internetu (respektive minimálně pro IP adresu monitorovacího serveru). Z hlediska zabezpečení je komunikace ověřována pomocí pevného uživatelského účtu, jenž má přístup jen na monitorovací endpointy a nijak nemůže pracovat se samotným API. Komunikace probíhá přes HTTPS.

Výhodou jsou i široké možnosti notifikací, které v základu podporují hned několik platforem včetně notifikování o změně stavu instancí do kanálu Slacku (zde pomocí WebHook), které funguje naprosto okamžitě a spolehlivě (samozřejmě v rámci nastavených intervalů zjišťování stavu instancí). Kromě notifikací do zmíněného Slacku, je možné využít e-mailové notifikace, následně platformy Microsoft Teams, Discord, Telegram, Hipchat, OpsGenie (Atlassian) a další. (9)

4 Charakteristika Node.js

Open source multiplatformní runtime prostředí Node.js je založeno na jazyku JavaScript, s tím hlavním rozdílem oproti běžnému použití JavaScriptu, že umožňuje spuštění kódu mimo webový prohlížeč, a tak je možné jej využít pro tvorbu serverových částí aplikací. Jedná se o vysoce škálovatelný a výkonný systém, který může být použit pro různý počet klientů zároveň.

Node.js využívá interpret V8 z jádra Google Chrome a to jednovláknově. Jedná se o vysoce výkonný open source engine pro jazyk JavaScript s možností implementace do aplikace psané v jazyce C++. (10)

Příchozí požadavky se odbavují v jednom vlákně API (respektive interpretu V8), kdy je velice kvalitně zvládnuta minimalizace blokování kódu a systémových prostředků. Díky tomu je možné zpracovávat tisíce souběžných připojení. (11)

Node.js defaultně používá velice rozšířený správce balíčků npm (Node.js package manager), kde je k dispozici velké množství balíčků a knihoven pro okamžité použití. Obsahuje více než 1 000 000 balíčků pro volné použití (open source). (11)

Komunita kolem Node.js vytvořila velké množství knihoven – frameworků, které mají za cíl usnadnit vývojáři implementaci. Samotný Node.js je totiž nízkoúrovňový, a tudíž by se vývojář musel zaobírat například samotnou implementací komunikace přes protokol HTTP, tedy parsování requestů, definování response headers a podobně. (12)

Jednou z nejpoužívanějších knihoven je „Express", díky které je vytvoření webového serveru (API) jednoduchou operací a výše uvedené nedostatky z větší části řeší. Vývojář se tak může soustředit na samotný vývoj obchodní logiky.

4.1 Nedostatky

4.1.1 Request body mapping

Jednou z problémových oblastí, kterou je potřeba zmínit, je mapování request body. Node.js neobsahuje možnost nadefinování datových typů v očekávaných requestech, takže je nutné provádět velmi neprakticky kontrolu datových typů přímo v samotném programu ručně. Díky tomuto nedostatku je možné bez problému například zaslat na API request s parametrem „cislo" a jeho hodnotou

„string“, i když program očekává desetinné číslo. Pokud tedy nedojde k zachycení před zpracováním, může dojít k neošetřené výjimce.

Oproti frameworku Spring Boot, který samotným definováním datových typů u objektu pro parsování requestu provádí kontrolu na datový typ automaticky, bez potřeby dalších závislostí. Všechny výjimky jsou tedy v tomto případě ošetřené a request nebude zpracován.

4.1.2 Validace request body

Node.js, na rozdíl od Spring Boot, neumožňuje v základu jednoduchou validaci příchozích dat u request body a vývojář si musí naprogramovat vlastní validaci (minimální, maximální délka textu, hodnota a podobně), případně je nutné využít například balíček *„express-validator“* a následně před zpracováním dat provést definování pravidel pro validaci. Ve Spring Boot je samotná validace vyřešena automaticky na základě anotace *„@field“* přímo v objektu, na který se request mapuje. Obecně má Spring Boot lépe vyřešenou například implementaci a ošetření záležitostí kolem HTTP, takže vývojář se oproti právě Node.js nemusí zabývat implementací základních věcí, kde může docházet k chybě.

```
const { body, validationResult } = require('express-validator');

app.post(
  '/user',
  // username must be an email
  body('username').isEmail(),
  // password must be at least 5 chars long
  body('password').isLength({ min: 5 }),
  (req, res) => {
    // Finds the validation errors in this request
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    User.create({
      username: req.body.username,
      password: req.body.password,
    }).then(user => res.json(user));
  },
);
```

Obrázek 25 – Ukázka validace request body Node.js (13)

```
@Schema(description = "Název firmy", example = "", required = true)
@field:Size(min = 1, max = 100)
var firma: String = "",

@Schema(description = "Kód firmy", example = "", required = true)
@field:Size(max = 15)
var kodfirmy: String = "",

@Schema(description = "Okruh", example = "", required = false)
@field:NotEmpty @field:Size(max = 30)
var okruh: String = "",

@Schema(description = "Typ", example = "", required = false)
@field:NotEmpty @field:Size(max = 30)
var typ: String = "",
```

Obrázek 26 – Ukázka anotace field pro validaci request body (zdroj: autor)

5 Popis implementace a vývoje REST API

5.1 REST API

Zkratka REST pramení z anglického „Representation State Transfer“ a jedná se o styl architektury rozhraní (s určitými doporučeními a zásadami), datově orientovaný (oproti například SOAP), který umožňuje přistupovat ke zdrojům pomocí protokolu HTTP a jeho metod (není na něj však přímo vázán, jen se jedná o nejrozšířenější protokol pro přenos dat). Samozřejmostí je podpora SSL -> HTTPS protokolu. (14)

REST implementuje tzv. CRUD metody – Create, Retrieve, Update a Delete, které jsou následně dostupné pomocí odpovídajících HTTP metod: GET (Retrieve), POST (Create), PUT (Update) a DELETE. (14)

Nejčastějším formátem zpráv je zde JSON (JavaScript Object Notation), může se ale jednat i o HTML, XML a další formáty. V tomto se REST odlišuje od protokolu SOAP (Simple Object Access Protocol), kde se zásadně pro zprávy využívá XML formát. (14)

REST API by mělo být tvořeno jako bezstavové – takže veškeré potřebné informace, například ověřovací údaje, by měly být zasílány s každým požadavkem. (14)

5.1.1 Stavové kódy

Díky využití protokolu HTTP existuje nepřeberné množství stavových kódů, případně je možné si pro konkrétní využití vytvořit a použít vlastní. Nejběžnější je „200 OK“ – požadavek byl úspěšně zpracován, „404 Not Found“ – Požadovaný zdroj nebyl nalezen nebo „500 Server Error“ – Interní chyba serveru.

Samozřejmě je možné používat jakékoliv další stavy. Například pro oznámení nezdařené operace vlivem neplatné validace dat – chybějící nebo neplatné vstupní parametry, zakázaná operace s konkrétní položkou a podobně je ve vyvíjeném API využito HTTP stavu „400 Bad Request“. Pro neplatnou transakci (síťová kolize) je naopak využit stav „409 Conflict“, kdy je zároveň do response přidán správný, aktualizovaný objekt.

6 Napojení na databáze

Připojení k databázi je ve většině případů nedílnou součástí vývoje API pro poskytování plnohodnotných služeb klientům, ať už se jedná o připojení k databázi za účelem evidence přístupových údajů/účtů nebo práci se samotnými daty (například API pro e-shop bude potřebovat načíst seznam artiklů vedených v databázi). Jedním ze základních cílů vývojáře je vytvořit a používat jednotnou unifikovanou vrstvu pro práci s databází tak, aby se dala použít na různé typy databázových systémů s odlišnými přístupy. Tato práce se bude zabývat relační databází Firebird, která není až tak známá a tím pádem je nutné při výběru prozkoumat podporu této databáze v daných technologiích. S nejpoužívanějšími databázemi, jako jsou například Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL a podobně, není při použití technologií Node.js, ani Spring Boot žádný zásadní problém.

V případě provozu API na cloudových službách se dá uvažovat například o Amazon RDS jakožto jedné ze služeb AWS, kde lze provozovat například PostgreSQL v prostředí přímo vázaném na další služby AWS (kde mohou následně běžet instance API obohacené například o load balancer), nebo využít jejich relační vysoce výkonnou, zabezpečenou a škálovatelnou databázi Amazon Aurora (kompatibilní s MySQL a PostgreSQL).

6.1 Firebird databáze

Firebird je open source relační databáze, kterou je možné provozovat jak na Windows, Linux, tak i například macOS operačním systému. Samotný projekt Firebird oficiálně vznikl v roce 2000 a původně vychází z databáze InterBase. (15) Jedná se o velice výkonný databázový systém, na kterém není problém provozovat více jak 10 GB, i 100 GB databáze s vyššími desítkami, až stovkami souběžných připojení. Aktuální verze je Firebird 4.0. Velkou výhodou je možnost použití pro komerční účely a také neexistující omezení (žádné omezování prostředků, velikosti databáze, počtu připojení atd).

Během implementace se podařilo vyřešit samotné napojení na Firebird databázi, ale hlavně také vlivem vyžadované kompatibility s verzí 4.0 bylo třeba vynutit použití *Legacy Authentication* jako hlavního metody pro ověřování místo *SRP* (Secure Remote Password), které prozatím není kompatibilní s celým informačním systémem.

Vynucení bylo provedeno přidáním parametru „?authPlugins=Legacy_Auth“ do URL datasource ve Spring Boot konfiguraci.

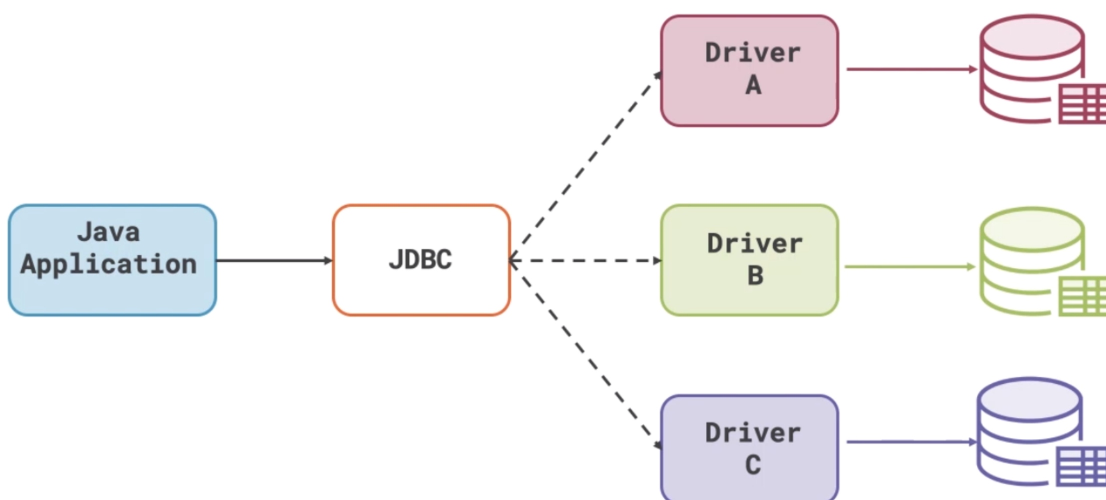
Dalším nutným zásahem vzhledem k databázi, bylo přepnutí datových typů u některých atributů z Integer (32 bitů) na BigInt (64 bitů), jelikož rokem 2022 by došlo v některých případech, zejména u číslování dokladů k přetečení rozsahu. Z hlediska API to znamenalo jen přetypování dotčených atributů v entitách z datového typu Int na Long. Žádné problémy se po této úpravě neobjevily, databáze se provozují souběžně jak ve variantě s rozšířením na BigInt, tak i bez této úpravy.

6.2 Spring Boot

6.2.1 JDBC – Java Database Connectivity

Jedná se o rozhraní, které poskytuje výše uvedený unifikovaný přístup k databázím. Samotnému JDBC API je nutné podsunout konkrétní JDBC ovladač příslušný k použité databázi (ten poskytuje vývojář databáze), který překládá námi definované požadavky na nativní volání.

Problémem vývojáře tedy zůstává jen mnohdy nekompatibilní zápis SQL, kdy se jednotlivé typy databází mohou lišit a SQL je nutné přizpůsobit pro konkrétní databázi (nedostupnost některých funkcí, jiný zápis a podobně).



Obrázek 27 – Ukázka rozhraní JDBC (16)

```
# DATABASE
spring.datasource.driver-class-name=org.firebirdsql.jdbc.FBDriver
spring.datasource.url=jdbc:firebirdsql://${SGA_DB_HOST}:${SGA_DB_PORT}/${SGA_DB_NAME}?authPlugins=Legacy_Auth&encoding=${SGA_DB_ENCODING}
spring.datasource.username=${SGA_DB_USERNAME}
spring.datasource.password=${SGA_DB_PASSWORD}
```

Obrázek 28 – Ukázka definování ovladače pro JDBC a parametrů pro připojení k Firebird databázi (zdroj: autor)

6.2.2 Pooling

Z hlediska výkonu je nepraktické pro každý SQL příkaz navazovat nové připojení na databázi a po dokončení příkazu jej zase ukončovat. Z tohoto důvodu se nejčastěji používá tzv. pooling, který při startu naváže požadovaný počet spojení na databázi (určeno v konfiguraci) a následně se spojení propůjčují pro vykonání konkrétních příkazů, po jejich dokončení je spojení zase uvolněno a navráceno zpět do poolu pro další využití. Díky tomu je ušetřeno velké množství času a výkonu potřebného k navazování nových spojení. (17)

Ve Spring Boot se nejčastěji používá rychlý framework HikariCP, který se právě o cykly spojení s databází stará a spravuje je (propůjčuje). Požadovaný maximální počet spojení, které HikariCP vytvoří, můžeme ovlivnit v konfiguračním souboru nejčastěji pojmenovaném jako „*application.properties*“ parametrem „*spring.datasource.hikari.maximum-pool-size*“, kromě toho můžeme dále například ovlivnit čas timeoutu, vynucenou kontrolu spojení, časy pro udržování spojení a podobně.

6.2.3 ORM – Hibernate – JPA (Java persistence API)

Objektově relační mapování (ORM) je programovací technika, která výrazně ulehčuje práci s daty relační databáze, a to především díky tomu, že není potřeba programovat například základní SELECT příkazy nebo další operace pro entity (INSERT, UPDATE, DELETE), a zároveň i postupy jako je zahájení transakce, její potvrzení (COMMIT) a podobně. ORM jednoduše udržuje konzistenci mezi objekty v programu a databází s důrazem na co nejmenší režii ze strany vývojáře.

JPA je pouze specifikací a základní vrstvou pro následnou implementaci ve frameworku Hibernate (může být použit i jiný). Ta následně umožňuje anotovat třídu jako *@Entity* – díky této anotaci je následně entita považována „za řádek tabulky“ v databázi. Poté je možné následně využívat všechny výhody i například v přípravě

SELECT příkazu – zde se používá trochu odlišné SQL, respektive HQL (Hibernate Query Language), který je záležitostí Hibernate a je inspirován syntaxí SQL. Samotné vygenerování příslušného SQL příkazu probíhá automaticky dle vyvolané akce, kterou může být například (18):

- „getById()“ – načtení konkrétního objektu dle ID, v případě nenalezení záznamu dojde k výjimce EntityNotFoundException;
- „findById()“ – podobné jako „getById()“ s rozdílem, že v případě nenalezení nebude vyvolána výjimka ale výsledek bude prázdný;
- „existsById()“ – ověření existence záznamu v databázi, obvykle se používá před „getById()“;
- „save()“ – v rámci transakce dojde k uložení nového nebo aktualizovaného záznamu do databáze;
- „saveAll()“ – možnost uložení více záznamů naráz (například všechny položky dokladu);
- „saveAndFlush()“ – oproti předchozím se zde provede uložení okamžitě a nečeká se na potvrzení transakce například při dokončení celé funkce.

```
@Entity
@Table(name="PHBSKLD01")
class SkladovaPolozka: PHBSKLD01() {

    @JsonManagedReference
    @ManyToOne
    @JoinColumn(name = "OID\$CENKTGM01",
                referencedColumnName = "OID\$CENKTGM01",
                insertable = false, updatable = false)
    var artikl: OnlyArtiklDetail? = null

    @JsonManagedReference
    @ManyToOne
    @JoinColumn(name = "KODSKLMISTA",
                referencedColumnName = "KODSKLMISTA",
                insertable = false, updatable = false)
    var skmdet: SkladoveMistoDetail? = null

    @Transient
    var doklad: DklSubSkl? = null

    @Transient
    var dodavatelInfo: ArtiklDodavatel? = null

    @Transient
    var skladovaKarta: SkladSkladovaKarta? = null
}
```

Obrázek 29 – Hibernate – ukázka @Entity (zdroj: autor)

Každá entita – třída, musí mít svůj identifikátor (anotovaný *@Id*), obvykle je to tedy primární klíč dané tabulky. Vzhledem k „otisku“ entity vůči databázovému záznamu je možné provádět automatické načítání objektů a dalších operací s ním bez nutnosti psaní specifického kódu ze strany vývojáře.

Další velkou výhodou používání Hibernate je možnost vytváření vztahů mezi entitami – stejně jako se v databázi řeší vztahy pomocí cizích klíčů, je zde možné využít anotace *@OneToMany*, *@OneToOne* a podobně s konkrétní definicí spojovacího prvku. Díky tomu může Hibernate automaticky načíst nejen entitu samotnou, ale i

nadefinované související entity včetně ulehčené práce s nimi (nemusí být zapotřebí například ukládat každou entitu zvlášť, ale uložením hlavní entity se mohou automaticky uložit i její potomci – určujeme pomocí „*CascadeType*“). Dalším parametrem (volitelným) je způsob načtení (tzv. „*fetch*“), kde je možné zvolit mezi možností „*EAGER*“ (náročnější – okamžitě při načtení hlavního objektu) nebo „*LAZY*“ (optimální – načtení prvků až při přístupu k nim).

```
@JsonManagedReference
@OneToMany(cascade = [CascadeType.ALL], fetch = FetchType.EAGER)
@JoinColumn(name = "OID\$DKLEVDM01", referencedColumnName = "OID\$DKLEVDM01",
            insertable = false, updatable = false)
@Where(clause = "P\$DELETE=0")
@OrderBy("PORADI")
var polozky: Set<SkladovaPolozka>? = null
```

Obrázek 30 – Hibernate – ukázka @OneToMany (zdroj: autor)

Ve své podstatě tedy Hibernate provede automatické vygenerování vlastního SQL dotazu podle vytvořené entity, ať už se bude jednat o SELECT, INSERT, UPDATE nebo DELETE. S tím souvisejí zároveň ale i nevýhody – vývojář nemá tento proces pod kontrolou a může ovlivnit jen minimálně. Je tedy třeba dbát co nejvíce na správnou konfiguraci entit – jejich atributů a vztahů. Jeden z běžných problémů, který se u rozsáhlých propojených entit vyskytuje, je posloupnost uložení nových záznamů, kdy musí být splněny vazby na cizí klíče v každé tabulce.

6.3 Node.js

Napojení Node.js na běžně používané databáze, je možné provést přes konkrétní balíčky z repositáře *npm* – například *node-mssql* (klient pro Microsoft SQL Server), *node-mysql* (klient pro MySQL), *node-mariadb*, *node-postgresql* a podobně.

Balíček *node-firebird* jako klient Firebird databáze je též dostupný. Bohužel není z hlediska aktualizací a komunity až tak aktivní, jak by bylo potřeba, navíc se objevují dlouhodobě neřešené problémy na platformě GitHub. Vzhledem ke komerčnímu využití API je zásadní spolehlivost a podpora – v ideálním případě by mělo jít o oficiální ovladač.

7 Porovnání Spring Boot a Node.js

7.1 Prostředí vývoje

7.1.1 Spring Boot

Pro vývoj API s frameworkem Spring Boot se osvědčilo vývojové prostředí IntelliJ IDEA od české společnosti JetBrains, které nabízí plnou podporu pro programovací jazyk Kotlin (také vývojáři z JetBrains). Fungování celého Spring Boot v tomto prostředí je spolehlivé a za celou dobu vývoje nebyly odhaleny žádné nedostatky. IntelliJ IDEA nabízí nepřeberné množství nastavení a pluginů, které vývojáři usnadňují práci a orientaci v kódu. Mezi takové patří například *Comments Highlighter* pro barevné zvýraznění komentářů nebo *Rainbow Brackets* pro lepší orientaci v kódu díky barevnému odlišení párů závorek.

Mezi další užitečné funkce lze zařadit možnost automatického nahrávání výsledného package na server pomocí SFTP protokolu nebo možnost napojení na databází za účelem efektivní práce při vytváření objektů.

Za zmínku stojí také velice propracovaná funkce Smart Cast, která se snaží odvodit datový typ a případně i nullable typ proměnné dle předchozích podmínek v kódu a umožnit tak bezpečný přístup k dané proměnné (v IDE následně zvýrazněno).

Průměrná doba čistého buildu na sestavovacím prostředí Maven, se všemi potřebnými knihovnami vyvíjeného API, se pohybuje okolo 20 sekund (na stroji MacBook Pro s ARM čipem M1 Pro a nativním OpenJDK verze 17).

7.1.2 Node.js

Vývojové prostředí IntelliJ IDEA od JetBrains lze využít i pro vývoj ve frameworku Node.js, případně lze využít i IDE WebStorm od stejného vývojáře. Zásadní nedostatky při vývoji ve zmíněných IDE nejsou. Základním předpokladem je nainstalování Node.js runtime prostředí (dostupné z https://nodejs.org/en/download/ pro platformy Windows, macOS a Linux) se správcem balíčků npm.

7.2 Prostředí pro běh API

API na frameworku Spring Boot je provozováno na více jak desítce produkčních serverech, ať už s OS Windows nebo Linux, a do této chvíle nebyly zaznamenány jakékoliv problémy s výkonem nebo spolehlivostí. Zásadním prvkem je instalace JVM – například OpenJRE od Adoptium. Vzhledem k využívání wrapperu WinSW pro spouštění API jako služby na Windows OS je zapotřebí dodržovat doporučené verze Windows Serveru (2016 a vyšší). Se staršími verzemi Windows Serveru (2008, 2011) nebyla aplikace optimálně stabilní – docházelo ke ztrátě vazby procesu mezi wrapperem a samotným JVM procesem Javy, což následně znemožňovalo ovládání služby do doby manuálního vynuceného ukončení procesu. Po klientech, kteří mají zájem o mobilní aplikaci k informačnímu systému, je tedy vyžadována minimální verze OS Windows Server 2016.

Sledováním v reálném provozu bylo zjištěno, že průměrná instance API při obsluhování požadavků, z cca 20 souběžně připojených klientských zařízení (mobilní aplikace) zabere na serveru přibližně 1,5-3 GB paměti RAM podle aktuálního využití a prováděných operací. V případě provádění náročnějších operací (například hromadný příjem/výdej položek ze skladu) si instance vezme i 4 GB paměti RAM a až 100 % přiděleného výkonu CPU na nezbytně nutnou dobu.

Provozování více instancí na serveru zároveň, napojených na různé databáze, není žádný problém. Běžně jsou provozovány na serverech 2-6 souběžných instancí (produkční prostředí, testovací prostředí a dále případně rozdělené instance pro různé provozy dle potřeby).

7.3 Ukázka základní implementace endpointu ve Spring Boot

Níže je k dispozici ukázka implementace endpointu s metodou GET, určeného pro načtení modulových nastavení pobočky. Anotace *@Operation* a *@ApiResponses* souvisí s automaticky generovanou dokumentací OpenAPI Swagger a konkrétně definují návratové hodnoty, popis a zabezpečení endpointu. Anotace *@GetMapping* definuje URI vůči root URL třídy anotované jako *@RestController* a *@RequestMapping*, zároveň je zde definován typ návratových dat – zde jde konkrétně o formát JSON (důležité pro správné automatické zakódování dat do response).

Vstupními parametry této metody jsou zkratka modulu, dále pak evidenční řada a subřada, u kterých jsou zároveň definovány výchozí hodnoty a není nutné je zadávat (jsou nepovinné).

Příklad volání metody GET tohoto endpointu pro získání informací může být „https://localhost:8443/v1/system/OZS?evdrada=1&evdsubrada=1“.

Návratová hodnota bude v případě HTTP kódu 200 list entit „SystemInfo“, v případě HTTP kódu 400 (ValidationException) se bude jednat o plain text „Neplatný modul – xxx“.

```
// GET parametry modulu
Vítek Vaníček
@Operation(summary = "Výpis modulových firemních parametrů pobočky", security = [(SecurityRequirement(name = "basicAuth"))])
@ApiResponses(value = [
    ApiResponse(responseCode = "200", description = "OK", content = [
        (Content(mediaType = "application/json", array = (ArraySchema(schema = Schema(implementation = SystemInfo::class)))))])])
@GetMapping("/{modul}", produces = [MediaType.APPLICATION_JSON_VALUE])
fun getParametryModulu(
    @Parameter(description = "Zkratka modulu")
        @PathVariable("modul") modul: String,
    @Parameter(description = "Evidenční řada")
        @RequestParam("evdrada", required = false, defaultValue = "1") evdrada: Int,
    @Parameter(description = "Evidenční subřada")
        @RequestParam("evdsubrada", required = false, defaultValue = "1") evdsubrada: Int
): ResponseEntity<Any> {
    val response = when(modul.uppercase()) {
        // Odeslané zásilky
        "OZS" -> globalSettings.getFiremniParModulOdeslaneZasilky()
        else -> throw ValidationException("Neplatný modul - $modul")
    }

    return ResponseEntity.ok(response)
}
```

Obrázek 31 – Ukázka implementace endpointu metody GET (zdroj: autor)

Závěr

Zkoumáním a testováním bylo zjištěno, že obě technologie jsou pro vývoj API vhodné a je především na zvážení vývojáře, kterou možnost se rozhodne využít. Žádná z uvedených technologií nemá zásadní nedostatky, avšak Spring Boot nabízí rozsáhlejší možnosti ve směru výběru programovacího jazyka a sestavovacího prostředí a má lépe zvládnutou implementaci a ošetření některých záležitostí z oblasti HTTP a podobně. Obě technologie mají velice rozsáhlou a aktivní komunitu a jsou hojně využívány v produkčních systémech od API pro bankovní sektor až po e-shopy a běžné weby & aplikace.

Pro vývoj API k účelu zprostředkování komunikace mezi mobilní aplikací a databází informačního systému byla zvolena technologie Spring Boot. Především se jedná o dostupnější použití a kvalitnější podporu databázového systému Firebird (podmínka pro vývoj API), než má technologie Node.js. Dále byla jasná výhoda v možnosti použití programovacího jazyka Kotlin, dostupnost knihoven, velice jednoduché spuštění na cílových platformách bez nutnosti instalace dalších podpůrných systémů (JRE již na serverech ve většině případů je nainstalované) oproti Node.js ke kterému by bylo nutné nainstalovat běhové prostředí. Velkým plusem pro volbu Spring Boot je také možnost zpracovávání požadavků současně, jelikož je oproti Node.js vícevláknový, což se hodí například pro hromadné akce náročnější na CPU (hromadný výdej položek ze skladu, propočtová kontrola položek na desítkách dokladů zároveň a podobně).

Programování v jazyce Kotlin je velice intuitivní a efektivní, samotné IDE a kompilátor následně eliminuje zanesení nejrůznějších chyb. Spring Boot je pravidelně aktualizován a pružně reaguje na objevené bezpečnostní a jiné problémy.

V rámci vývoje API k mobilní aplikaci pro informační systém nebyly zjištěny žádné zásadní nedostatky, které by zpochybňovaly volbu technologie Spring Boot. API je nasazeno v několika firmách v České republice i zahraničí a úspěšně obsluhuje mobilní aplikace běžících v logistických skladech, v terénu u řidičů, obchodních zástupců nebo například servisních techniků.

Použité zdroje

1. Best practices for REST API design. *Stack Overflow Blog.* [Online] [Citace: 8. březen 2022.] https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/.

2. Introducing JSON. [Online] [Citace: 26. únor 2022.] https://www.json.org/json-cz.html.

3. Stack Overflow Trends. *Stack Overflow Insights.* [Online] [Citace: 18. březen 2022.] https://insights.stackoverflow.com/trends?tags=json%2Cxml%2Csoap%2Crest.

4. Google Trends. *Porovnání.* [Online] [Citace: 20. duben 2022.] https://trends.google.com/trends/explore?date=all&q=json,xml.

5. Java Spring Boot. *IBM .* [Online] [Citace: 10. duben 2022.] https://www.ibm.com/cloud/learn/java-spring-boot.

6. Java vs Kotlin. *Educba.* [Online] [Citace: 17. duben 2022.] https://www.educba.com/java-vs-kotlin/.

7. Gradle vs Maven: Performance Comparison. *Gradle Build Tool.* [Online] [Citace: 22. duben 2022.] https://gradle.org/gradle-vs-maven-performance/.

8. Spring Boot Annotations. *JavaTpoint.* [Online] [Citace: 26. březen 2022.] https://www.javatpoint.com/spring-boot-annotations.

9. *Spring Boot Admin Reference Guide.* [Online] [Citace: 6. březen 2022.] https://codecentric.github.io/spring-boot-admin/.

10. *V8 JavaScript engine.* [Online] [Citace: 20. březen 2022.] https://v8.dev/.

11. *Introduction to Node.js.* [Online] [Citace: 22. duben 2022.] https://nodejs.dev/learn.

12. Rozběhnutí projektu a první řádky v Expressu. *ITnetwork.cz.* [Online] [Citace: 25. duben 2022.] https://www.itnetwork.cz/javascript/nodejs/rozbehnuti-projektu-a-prvni-radky-v-expressu.

13. Documentation. *Express-validator.* [Online] [Citace: 23. duben 2022.] https://express-validator.github.io/docs/.

14. REST APIs. *IBM.* [Online] [Citace: 2. duben 2022.] https://www.ibm.com/cloud/learn/rest-apis.

15. About Firebird. *Firebird.* [Online] [Citace: 29. březen 2022.] https://firebirdsql.org/en/about-firebird/.

16. How to use JDBC to connect database in Java project. *Manh Phan.* [Online] [Citace: 13. leden 2022.] https://ducmanhphan.github.io/2020-01-09-How-to-use-JDBC-to-connect-database-in-Java-project/.

17. JDBC Connection Pooling Explained with HikariCP. *CodersTea.* [Online] [Citace: 16. leden 2022.] https://coderstea.in/post/best-practices/jdbc-connection-pooling-explained-with-hikaricp/.

18. *Spring Data JPA Documentation.* [Online] [Citace: 18. duben 2022.] https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html.

Přílohy

1. Zdrojový kód API

Dostupný na adrese: https://github.com/vanicekv/BP_SignysLogisticsAPI

Přístupové údaje na vyžádání poskytne autor.

Zadání bakalářské práce

Autor: **Vítek Vaníček**

Studium: I1900656

Studijní program: B0688A140001 Informační management

Studijní obor: Informační management

Název bakalářské práce: **Využití Node.js a Spring Boot při tvorbě REST API**

Název bakalářské práce AJ: Use of Node.js and Spring Boot for creation of REST API

Cíl, metody, literatura, předpoklady:

Cíl práce: Porovnání jednoduchosti implementace a vývoje REST API v prostředí Node.js a frameworku Spring Boot 2 (jazyk Kotlin), zejména pak možnosti napojení na různé databázové systémy, možnosti rozšíření a škálovatelnost.

Osnova práce:

1. Úvod
2. Charakteristika Node.js a Spring Boot, a jejich odlišnosti
3. Popis implementace a vývoje REST API
4. Možnosti napojení na databázové systémy
5. Výsledky porovnání
6. Závěr

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 26.1.2021