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

Porovnání Java EE a Spring Frameworku

Bakalářská práce

Autor: Jan Tomáš Štekl
Studijní program: Aplikovaná informatika

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

Hradec Králové duben 2024

Prohlášení:

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

V Hradci Králové dne 22.4.2024 Jan Tomáš Štekl

Poděkování:

Děkuji vedoucímu bakalářské práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce a za návrhy na její vylepšení. Dále děkuji také rodině za celkovou podporu během studia.

Abstrakt

Bakalářská práce, jejíž cílem je porovnat Java EE (nyní Jakarta EE) a Spring Framework. Nejprve je tvořena teoretickou částí, kde se začíná kratším úvodem do Javy a souvisejících věcí. Následně teoreticky rozebere nejprve Jakarta EE a následně Spring Framework. U obou je popsána historie, principy, architektura a jejich součásti. Ke konci jsou zmíněny také jejich alternativy. Co se týče praktické části, zde je popsaný celý postup vývoje dvou aplikací. Nejprve je vytvořena aplikace stavěná na Spring Boot. Následně je vytvořena aplikace stavěná na Jakarta EE, a to předěláním aplikace předchozí. K aplikacím je vytvořena dokumentace a také skripty k jejich otestování. Aplikace jsou pak otestovány zátěžovým testem, kdy jsou pomocí K6.io skriptu odesílány POST, nebo GET požadavky testované aplikaci. Celou dobu byla snaha o to, aby ani jedna aplikace nebyla při testování znevýhodněna (např. jinou konfigurací, jinou implementací něčeho apod.).

Klíčová slova: Java, Spring, Jakarta EE, framework, testování, vývoj

Abstract

Title: Java EE and Spring Framework Comparison

Bachelor thesis, which aims to compare Java EE (now Jakarta EE) and Spring Framework. It first consists of a theoretical part, starting with a short introduction to Java and related matters. It then theoretically discusses first Jakarta EE and then Spring Framework. For both, the history, principles, architecture and their components are described. Towards the end, their alternatives are also mentioned. As far as the practical part is concerned, the whole development process of the two applications is described here. First, an application built on Spring Boot is created. Then, an application built on Jakarta EE is created by redesigning the previous application. Documentation is created for the applications, as well as scripts to test them. The applications are then stress tested by sending POST or GET requests to the application under test using the K6.io script. The whole time, the effort was to make sure that neither application was disadvantaged during testing (e.g. different configuration, different implementation of something, etc.).

Keywords: Java, Spring, Jakarta EE, framework, testing, development

Obsah

1 Úvod ... 1
2 Cíl a metodika práce ... 2
2.1 Aplikace vytvořená pomocí Spring Frameworku ... 2
2.2 Aplikace vytvořená pomocí Jakarta EE ... 2
2.3 Další specifikace ... 3
2.4 Definování aplikace ... 3
2.5 Struktura databáze – ERD Diagram ... 4
3 Úvodní teorie ... 5
3.1 Programovací jazyk Java (Java Standard Edition) ... 5
3.1.1 Ukázka programu v Javě ... 6
3.2 Klient-Server architektura ... 7
3.3 Maven ... 8
3.3.1 Cíle Maven ... 8
3.3.2 Struktura Maven projektu ... 9
3.3.3 Struktura pom.xml souboru ... 10
4 Java EE (Java Enterprise Edition) ... 11
4.1 Historie ... 11
4.2 Historie verzí (názvů) ... 11
4.3 Přeměna Java EE na Jakarta EE ... 11
4.3.1 JCP ... 12
4.3.2 JSR ... 12
4.3.3 TCK ... 12
4.4 Implementace Jakarta EE ... 12
4.5 Glassfish ... 13
4.6 Architektura ... 14
4.7 Popis vrstev architektury ... 14
4.7.1 Klient ... 14
4.7.2 Webová vrstva ... 14
4.7.3 Businessová vrstva ... 15
4.7.4 Databázová (EIS) vrstva ... 15
4.8 Příklady technologií Java EE ... 15
4.8.1 Java Servlets ... 15

4.8.2 JavaServer Pages (JSP) 16
4.8.3 JSF (Java Server Faces) 16
4.8.4 Enterprise JavaBeans (EJB) 16
4.8.5 Contexts and Dependency Injection (CDI) 16
4.8.6 Ukázka Dependency Injection v Java EE 17
4.8.7 Java Persistence API (JPA) 19
4.9 Důvody k použití Java EE 20
5 Spring Framework 21
5.1 Historie 21
5.2 Aplikační servery 21
5.3 Základní principy Spring Frameworku 21
5.3.1 Inversion of Control (IoC) 21
5.3.2 Dependency Injection (DI) 22
5.3.3 Aspect-Oriented Programming (AOP) 23
5.4 Architektura 24
5.5 Příklady modulů v architektuře 24
5.5.1 Spring Core a Spring Beans 24
5.5.2 Spring Context 24
5.5.3 Spring Expression Language 25
5.5.4 Spring Data 25
5.5.5 Spring OXM 25
5.5.6 Webová část (Spring Web) 26
5.5.7 Testovací část (Spring Test) 26
5.5.8 Spring Boot 26
5.5.9 Spring Boot Application Starters 27
5.5.10 Spring Boot Production Starters 27
5.5.11 Spring Boot Technical Starters 27
5.6 Důvody k použití Spring Frameworku 27
6 Alternativní frameworky 28
6.1 Ktor 28
6.2 Quarkus a Micronaut 28
7 Vývoj Spring aplikace 29
7.1 Inicializace 29
7.2 Perzistence 31

7.3 DTO třídy ... 34
7.4 Service třídy ... 37
7.5 Controller třídy a jejich zabezpečení ... 38
7.6 Konfigurace ... 41
8 Vývoj Jakarta EE aplikace ... 42
9 Testovací data a testování HTTP metod ... 48
10 Shrnutí a výsledky porovnání ... 52
11 Závěr ... 54
12 Seznam použité literatury ... 55
13 Přílohy ... 58

Seznam obrázků

Obrázek 1: ERD Diagram ... 4
Obrázek 2: Klient-Server architektura *(Zdroj: vlastní zpracování)*. ... 7
Obrázek 3: Struktura Maven projektu [7]. ... 9
Obrázek 4: Certifikované servery pro Jakarta EE 10 [14]. ... 13
Obrázek 5: Jakarta EE architektura [15] *(přeloženo)*. ... 14
Obrázek 6: Spring Framework architektura [29]. ... 24
Obrázek 7: Struktura Jakarta EE projektu, vytvořeno z template Web Profile v IntelliJ Idea *(Zdroj: vlastní zpracování)*. ... 42
Obrázek 8: Dokumentace v Postman – registrace uživatele ... 49
Obrázek 9: Nastavení Connection Poolu pro Datasource ve Wildfly ... 51
Obrázek 10: Počet HTTP hlaviček v response (Spring Boot) ... 53
Obrázek 11: Počet HTTP hlaviček v response (Jakarta EE) ... 53

Seznam tabulek

Tabulka 1: Verze Java EE [10] *(přeloženo, doplněno o verze z* [12]*)*. ... 11
Tabulka 2: Výsledky měření ... 52
Tabulka 3: Porovnání dalších metrik (počet řádků kódu počítán u Java tříd bez mezer) 53

Seznam příkladů

Příklad 1: Ukázka programu v Javě *(Zdroj: vlastní zpracování)*. ... 6
Příklad 2: HTTP GET požadavek [5]. ... 7
Příklad 3: Odpověď na HTTP požadavek [5]. ... 8
Příklad 4: Ukázka souboru pom.xml [7]. ... 10
Příklad 5: Ukázka Servletu v Java EE [18]. ... 15
Příklad 6: Ukázka EJB/CDI Dependency Injection *(Zdroj: vlastní zpracování)*. ... 18
Příklad 7: Ukázka Entity třídy mapující SQL tabulku [22]. ... 19
Příklad 8: Ukázka Dependency Injection ve Springu *(Zdroj: vlastní zpracování)*. ... 23
Příklad 9: Ukázka Spring Expression Language [30]. ... 25
Příklad 10: Ukázka interface repozitáře Spring Data JPA [22]. ... 25
Příklad 11: Ukázka Controlleru v Spring MVC [31]. ... 26
Příklad 12: SQL pro vytvoření potřebných tabulek *(Zdroj: vlastní zpracování)*. ... 29
Příklad 13: Inicializace Spring Boot aplikace pomocí Spring Initializr ... 30
Příklad 14: Počáteční struktura projektu *(Zdroj: vlastní zpracování)*. ... 30

Příklad 15: Ukázka namapované entity na tabulku Post, využití dědičnosti *(Zdroj: vlastní zpracování)*. 31
Příklad 16: Generická DAO třída, upravena [37]. 32
Příklad 17: Pagination pomocí JPA Criteria API [38]. 32
Příklad 18: Vlastní implementace DAO třídy *(Zdroj: vlastní zpracování)*. 33
Příklad 19: DTO třída uživatele *(Zdroj: vlastní zpracování)*. 34
Příklad 20: DTO třída uživatele pro přihlášení a registraci *(Zdroj: vlastní zpracování)*.34
Příklad 21: Nastavení maven-compiler-plugin pro Lombok a MapStruct podle [39]. ... 35
Příklad 22: Námi definovaný mapper interface a jeho vygenerovaná implementace. ... 36
Příklad 23: PostService – třída pro operace s příspěvkem, zjednodušeno (ponechány pouze dvě metody – přidání a zobrazení všech příspěvků) *(Zdroj: vlastní zpracování)*.37
Příklad 24: PostController – tvorba HTTP metod pro práci s příspěvky, zjednodušeno (podle příkladu PostService) *(Zdroj: vlastní zpracování)*. 38
Příklad 25: Spring Security konfigurace s HTTP Basic Security, připojením na databázi s uživateli a rolemi, a BCrypt hashováním hesel, zjednodušeno *(Zdroj: vlastní zpracování)*. 39
Příklad 26: Získání aktuálního uživatele v Spring Security z SecurityContext *(Zdroj: vlastní zpracování)*. 40
Příklad 27: Spring Data JPA - konfigurace připojení k databázi 41
Příklad 28: persistence.xml konfigurace *(Zdroj: vlastní zpracování)*. 43
Příklad 29: beans.xml defaultní konfigurace. 43
Příklad 30: Jakarta EE HTTP endpoint, zjednodušeno (stejný příklad, jako ve Spring verzi). *(Zdroj: vlastní zpracování)*. 44
Příklad 31: Security konfigurace v Jakarta EE *(Zdroj: vlastní zpracování)*. 45
Příklad 32: Security konfigurace, zjednodušeno (reálná konfigurace je stejná, jako v případě Spring verze aplikace). *(Zdroj: vlastní zpracování)*. 46
Příklad 33: Konfigurace Jackson stejně, jako je defaultní konfigurace ve Spring[42]. *(Zdroj: vlastní zpracování)*. 47
Příklad 34: Ukázka cyklu tvořícího fake testovací data do databáze. *(Zdroj: vlastní zpracování)*. 48
Příklad 35: Hlavní funkce JS scriptu (první je pro test GET, druhá POST požadavků) pro load testing pomocí K6, zjednodušeno a zkombinováno. *(Zdroj: vlastní zpracování)*.. 50

1 Úvod

V dnešní době je software důležitou součástí různých odvětví. Pro dlouhodobé a efektivní použití takových aplikací je klíčové, aby byly vytvořené na stabilních a škálovatelných technologiích. K dosažení těchto cílů jsou důležité frameworky a technologie, které umožní vývojářům možnosti abstrakce. Díky tomu mohou vývojáři vytvářet aplikace rychle, efektivně, a lépe je tak mohou rozšiřovat. Zároveň díky zjednodušení nemusí opakovaně vytvářet základní části programu, ačkoli v některých specifických případech může pevné stanovení určité funkčnosti ve frameworku dělat problémy.

Tato bakalářská práce podrobně rozebere dva frameworky pro programovací jazyk Java, a to konkrétně Java EE a Spring Framework. Obě možnosti jsou silnými možnostmi pro tvorbu robustních a spolehlivých aplikací (hlavně webových). Tyto frameworky ovšem mají odlišnou architekturu a některé vlastnosti, což se také rozebere. Stručně také ukáže i jiné alternativy kromě těchto dvou frameworků. Následně dojde k porovnání obou frameworků při vývoji vzorové aplikace.

Předtím, než budou podrobně představeny tyto frameworky, dojde k představení základních informací o programovacím jazyce Java. Její znalost je důležitá pro práci s technologiemi, které na ni navazují. Také popíše základní teorii ohledně klient-server architektury, která se běžně používá při vývoji webových aplikací.

2 Cíl a metodika práce

Cílem této práce je kromě teoretického úvodu do obou frameworků také jejich porovnání pomocí praktického příkladu. Tento praktický příklad bude totožná aplikace vyvinutá v každém z těchto dvou frameworků.

Půjde o aplikaci, která bude využívat perzistenci (bude pracovat s databází) a přístup k ní bude řešen pomocí REST API (s autentizací). Front-end není řešen.

V případě obou těchto frameworků je nutné specifikovat, jaké návazné technologie u nich použijeme. Musí být nastaveny takové podmínky, aby ani jeden z frameworků nebyl znevýhodněn kvůli použití jiné navazující technologie. Z tohoto důvodu byly specifikace určeny takto:

2.1 Aplikace vytvořená pomocí Spring Frameworku

- Bude použit Spring Boot 3.2.1 (pojetí Spring Frameworku pohledem jeho vývojářů)
- Jako implementace JPA bude použit Hibernate
- Jako JSON Provider bude použit Jackson
- Connection Pool připojení na databázi – HikariCP
- HTTP requesty budou implementovány pomocí Spring Web
- Jako beany budou použity Spring Bean (např. @Component anotace)

2.2 Aplikace vytvořená pomocí Jakarta EE

- Aplikace bude vyvíjena pro spuštění na WildFly aplikačním serveru
- Pro JPA bude použit Hibernate (ze základu ve Wildfly)
- Jako JSON Provider bude použit Jackson
- Connection Pool bude použit v rámci aplikačního serveru (je to z důvodu, že to tak je zamýšleno při správě JTA transakcí)
- HTTP requesty budou implementovány standardně pomocí Jakarta EE
- Beany budou typu ApplicationScoped (CDI)

2.3 Další specifikace

Vše bude běžet na počítači s Windows 11 – konkrétně Lenovo Legion 5 15ARH05H (82B1004TCK).

Pro tvorbu a běh obou aplikací bude použit jdk-17.0.9+9.1 Windows x64 distribuce Temurin. Aplikace budou spuštěny na aplikačním serveru Wildfly Distribution 30.0.1.
Jako databáze bude použita MySQL Community 8.0.35. Pro každou aplikaci bude použita stejná databáze na jednom databázovém serveru (obě implementace budou s danou DB kompatibilní).

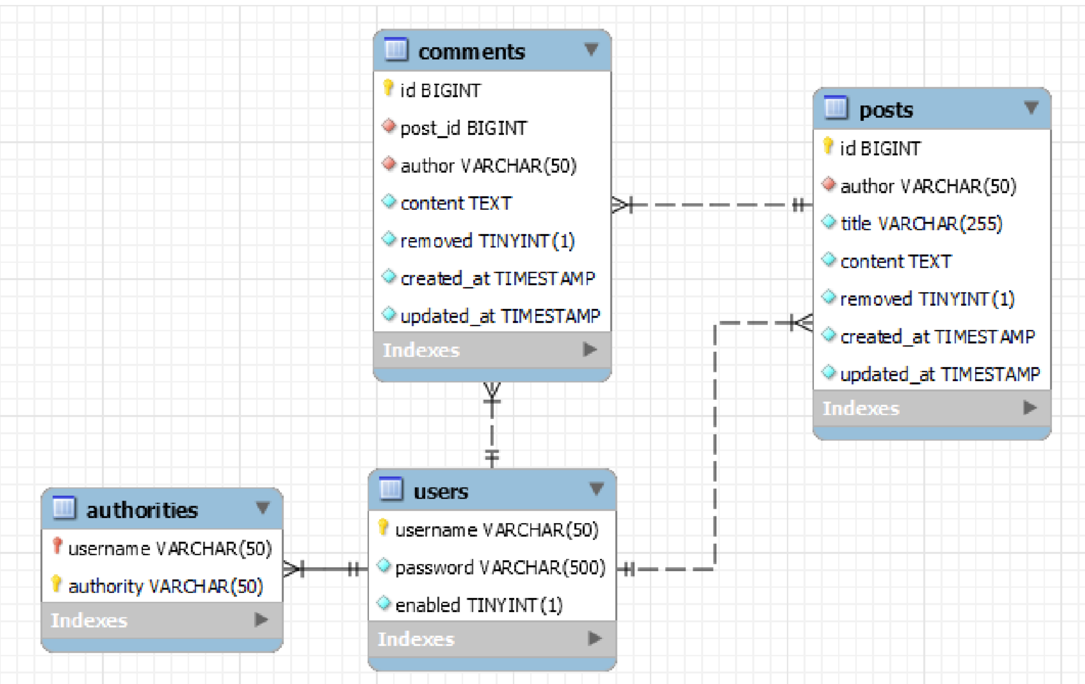
Výkon obou aplikací bude otestován pomocí programu K6.io (verze 0.48.0), kde bude každá z nich otestována stejnými parametry, a to pomocí paralelního přístupu na jejich HTTP endpointy. Bude tedy primárně měřen čas, za který zvládne aplikace odbavit určité množství požadavků.

Implementace aplikace bude dokumentována krok po kroku. Následně po dokončení obou implementací dojde k danému otestování.

2.4 Definování aplikace

V aplikaci bude moci uživatel po registraci a přihlášení přidat příspěvek a zobrazit si příspěvky všech uživatelů. K jakémukoliv příspěvku bude moci přidat komentář a zobrazit si komentáře u příspěvků. Své příspěvky a komentáře bude moci uživatel upravovat nebo odstranit (po odstranění je nepůjde vyhledat). Administrátor bude moci mazat příspěvky a komentáře všech uživatelů a také zablokovat uživatele (ten se v takovém případě nebude moci přihlásit). Zablokovaní uživatelé, smazané příspěvky a komentáře zůstanou v databázi. Administrátor bude určen tak, že mu v databázi bude ručně určena tato role (a pak bude mít práva právě na mazání a blokování).

2.5 Struktura databáze – ERD Diagram



Obrázek 1: ERD Diagram

3 Úvodní teorie

3.1 Programovací jazyk Java (Java Standard Edition)

Java je programovací jazyk pro vytváření aplikací pro různé účely. Byl vytvořen v roce 1995 společností Sun Microsystems (v roce 2010 odkoupena společností Oracle) [1]. Od základu se jedná o jazyk objektově orientovaný, a to díky podpoře konceptů abstrakce, zapouzdření a dědičnosti [1]. Prosazuje princip Write once, run anywhere (WORA) [2], jehož je docíleno kompilací kódu do byte-kódu, který následně na různých platformách běží v JVM (Java Virtual Machine).

Jednou ze zamýšlených vlastností je jednoduchost, které bylo docíleno použitím syntaxe podobné C++, ale bez komplexností [3]. Člověk mající znalost C++ by neměl mít problém začít produktivně programovat v Javě [3]. Je robustní a bezpečný, protože kód je kontrolován při kompilaci a také existují silné mechanismy pro řešení výjimek za běhu [2]. Správa paměti zde probíhá automaticky pomocí garbage collection. To usnadňuje vývojářům práci, protože nemusí řešit explicitní práci s pointery jako třeba v C++.

Za velkou výhodu Javy lze kromě vlastností popsaných výše také zmínit velkou komunitu a širokou podporu. Díky tomu je pro ni mnoho nástrojů, knihoven a frameworků. To umožňuje efektivní vývoj aplikací v Javě.

3.1.1 Ukázka programu v Javě

```
//Třída PlusOperation, v souboru PlusOperation.java
public class PlusOperation {

    //Atributy třídy
    private int a;
    private int b;

    //Konstruktor třídy
    public PlusOperation(int a, int b) {
        this.a = a;
        this.b = b;
    }

    //Metoda, která vrací součet atributů třídy
    public int plus() {
        return a + b;
    }
    //Metoda, která vrací výsledek součtu vynásobený parametrem metody
    public int resultMultiplication(int multiplyBy) {
        return plus() * multiplyBy;
    }

    //Gettery a settery pro získání/nastavení hodnot atributů.
    public int getA() {
        return a;
    }
    public void setA(int a) {
        this.a = a;
    }
    public int getB() {
        return b;
    }
    public void setB(int b) {
        this.b = b;
    }
}

//Třída Main, v souboru Main.java
public class Main {
    //Spustitelná metoda main, v souboru Main.java
    public static void main(String[] args) {
        //Vytvoření objektu
        PlusOperation plusOperation = new PlusOperation(5, 4);
        //Vypíšeme to, co metody toho objektu vrací
        //Vrací 9
        System.out.println("Výsledek součtu: " +
                           plusOperation.plus());
        //Vrací 27
        System.out.println("Vynásobení výsledku 3: " +
                            plusOperation.resultMultiplication(3));
    }
}
```

Výstup programu:

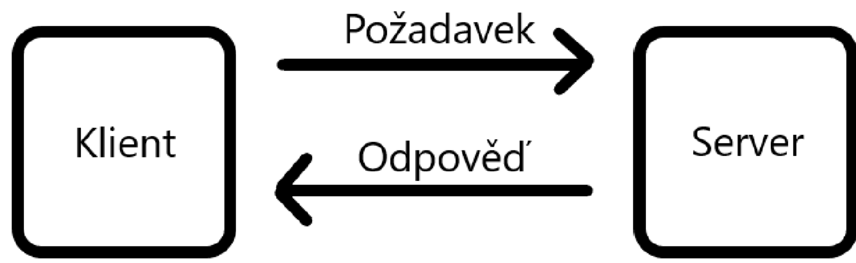
```
Výsledek součtu: 9
Vynásobení výsledku 3: 27
```

Příklad 1: Ukázka programu v Javě *(Zdroj: vlastní zpracování)*.

Ukázka představuje základní program v Javě, kde se sečtou dvě čísla a následně se nějakým číslem výsledek vynásobí. Spouštěná část programu se odehrává v statické metodě main. Z konceptů objektově-orientovaného programování zde lze vidět abstrakci (třída PlusOperation) a zapouzdření (gettery a settery).

3.2 Klient-Server architektura

Jedná se o základní architekturu používanou při vývoji webových aplikací. Ve své podstatě jsou zde dva prvky, a to klient a server. Klient posílá žádost serveru a server na tuto žádost odpovídá. Server je v tomto případě tedy pasivní (čeká na požadavek) a klient aktivní (posílá v případě potřeby požadavek). Architektura aplikace se pak většinou dělí na několik vrstev. To lze vidět právě třeba u Java EE viz. Java EE Architektura.



Obrázek 2: Klient-Server architektura *(Zdroj: vlastní zpracování)*.

Příklad komunikace vypadá takto [4]:

1. Uživatel přejde v prohlížeči na nějakou adresu (URL).
2. Prohlížeč se zeptá Domain Name System (DNS), o jakou IP adresu se jedná.
3. DNS server odpoví danou IP adresu.
4. Prohlížeč pošle HTTP/s požadavek.
5. Server pošle zpět soubory (třeba webovou stránku).
6. Klient obdrží dané soubory.

Komunikace s webovými aplikacemi probíhá často právě pomocí HTTP protokolu. Nejčastěji aktuálně používané HTTP metody jsou GET (v tomto případě klient pouze žádá o odpověď, ale sám přímo neposílá data, která by měl server zpracovat), anebo POST (klient posílá data a čeká na odpověď).

```
GET /server/http-protokol.html HTTP/1.1
HOST: www.jakpsatweb.cz
```

Příklad 2: HTTP GET požadavek [5].

Na základě tohoto HTTP požadavku přijde odpověď, v tomto případě webová stránka.

```
HTTP/1.1 200 OK
Content-type: text/html
Date: Sun, 21 May 2006 17:10:21 GMT

<html>
<head>
<title>stránka</title>
...atd.
```

Příklad 3: Odpověď na HTTP požadavek [5].

Jako první řádek v odpovědi je verze HTTP protokolu a stav zpracování požadavku. V tomto případě je stav 200 OK, takže požadavek byl zpracován v pořádku. Možných stavů je spoustu. Například chybový stav 404 not found znamená, že požadovanou věc server nemůže zpracovat, protože tam není. Stav 500 internal error vyjadřuje, že došlo k nějaké chybě v průběhu zpracování požadavku na serveru. Další řádky jsou pak hlavičky, kterých může být různé množství. Zde hlavička Content-type ukazuje, že se jedná o HTML webovou stránku. A také datum, kdy byla odpověď vytvořena. Po hlavičkách následuje mezera. Za mezerou je pak už samotný obsah, v tomto případě HTML kód dané webové stránky.

3.3 Maven

Apache Maven je nástroj usnadňující správu sestavení Java aplikací. Běžně se používá při vývoji aplikací s použitím jak Jakarty EE, tak i Spring Frameworku. Účelem je zautomatizování sestavení aplikace, aby došlo za co nejmenší čas k požadovaným operacím (otestování předem definovanými testy, kompilace, sestavení). Výhodou Mavenu je jeho repozitář, kdy vývojář nemusí stahovat jar soubory knihoven ručně, ale stáhnou se právě z repozitáře.

3.3.1 Cíle Maven

Cílem Mavenu je, aby byl proces sestavení jednoduchý a jednotný [6]. Vývojář se tedy při práci na různých projektech díky Mavenu (nebo alternativám) vyzná v projektu lépe, než kdyby si to každý dělal po svém. Dalším cílem je to, aby byly zřejmé informace o projektu (jeho struktura, knihovny) a dále se pak snaží vynucovat lepší praktiky při vývoji [6].

3.3.2 Struktura Maven projektu

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

Obrázek 3: Struktura Maven projektu [7].

V této ukázce je velmi jednoduchý projekt aplikace, kde je jedna třída a zároveň testovací třída JUnit této třídy. V kořenové složce my-app se nachází pom.xml soubor, kde se právě nastavuje konfigurace k sestavování projektu.

Ve složitějších projektech pak je více modulů. V tomto případě je zde jen rodičovský modul my-app. Mohly by zde být i podmoduly. Ty by mohly mít stejnou strukturu jako my-app modul a ve svém pom.xml by se na něj odkazovaly.

3.3.3 Struktura pom.xml souboru

Tento soubor může být velmi komplexní, ale hlavně tu jsou knihovny, které lze při sestavování použít (z repozitáře). Výhodou Maven je právě jeho repozitář Maven Repository, kde je indexovaných spoustu knihoven. Je možné ale použít i repozitář vlastní (lokální nebo vzdálený).

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.
5.   <groupId>com.mycompany.app</groupId>
6.   <artifactId>my-app</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.
9.   <properties>
10.      <maven.compiler.source>1.7</maven.compiler.source>
11.      <maven.compiler.target>1.7</maven.compiler.target>
12.   </properties>
13.
14.   <dependencies>
15.     <dependency>
16.       <groupId>junit</groupId>
17.       <artifactId>junit</artifactId>
18.       <version>4.12</version>
19.       <scope>test</scope>
20.     </dependency>
21.   </dependencies>
22. </project>
```

Příklad 4: Ukázka souboru pom.xml [7].

V tomto případě zde v pom.xml definujeme základní informace o aplikaci, čímž je groupId (do jaké skupiny aplikace patří, má být unikátní a podle konvence určené v Javě [8]), artifactId (název toho modulu/aplikace) a pak verzi. Toto se definuje i u knihoven (v elementu <dependencies> se určují <dependency>), které se vyhledají podle těchto informací v repozitáři. Dále se zde určují properties (v tomto případě, jaká je verze Javy v zdrojovém kódu a pro jakou verzi Javy se bude aplikace kompilovat). Dále se zde mohou nastavovat třeba různé dodatečné pluginy.

4 Java EE (Java Enterprise Edition)

Java Enterprise Edition (Java EE) je rozšířením Javy SE. Poskytuje API a běhové prostředí pro vývoj a provoz rozsáhlých aplikací. [9]

4.1 Historie

Do roku 1999 se tyto technologie neoddělovaly a byly přímo součástí JDK. V roce 2006 se začal používat název Java EE (poprvé Java EE 5). V roce 2017 došlo k velké změně, když se Oracle rozhodlo předat správu Java EE Eclipse Foundation. Na základě tohoto kroku se začal používat název Jakarta EE (slovo Java již nesmělo být použito) a došlo k otevření této technologie kromě některých částí dokumentace. [10]

Jakarta EE je nyní vyvíjena pod licencí Eclipse Public License 2.0. [11] Aktuální verze Jakarta EE 10 podporuje Javu 11 a 17.

4.2 Historie verzí (názvů)

Verze	Datum vydání
J2EE 1.2	prosinec 1999
J2EE 1.3	září 2001
J2EE 1.4	listopad 2003
Java EE 5	květen 2006
Java EE 6	prosinec 2009
Java EE 7	duben 2013
Java EE 8	srpen 2017
Jakarta EE 8	září 2019
Jakarta EE 9	listopad 2020
Jakarta EE 9.1	květen 2021
Jakarta EE 10	září 2022

Tabulka 1: Verze Java EE [10] *(přeloženo, doplněno o verze z* [12]*)*.

4.3 Přeměna Java EE na Jakarta EE

Poslední vydaná verze tradiční Java EE byla Java EE 8 v srpnu 2017. Jednalo se o obsahově redukované vydání. V tom samém roce pak společnost Oracle rozhodla o předání správy Java EE pod open-source sdružení Eclipse Foundation [13]. Toto předání bylo rozděleno na několik etap. Byl předán zdrojový kód. Muselo také dojít ke změně názvů z Java na Jakarta namespace (Maven groupId a až v rámci Jakarta EE 9 pak i package name) z javax.* na jakarta.* [13]. Se změnami se pojí několik pojmů.

4.3.1 JCP

JCP je zkratka pro Java Community Process. Jedná se o proces, kterým prochází tvorba nových specifikací do Javy. Standardní Java SE stále používá JCP [10], který je pod Oraclem. Java EE dříve používala také JCP, ale s přechodem správy pod Eclipse Foundation došlo k separaci tohoto procesu. Nyní se v rámci Java EE používá Eclipse Foundation Specification Process (EFSP), který je podprocesem Eclipse Development Process [10].

4.3.2 JSR

Java Specification Request (JSR) označuje první krok v procesu JCP. Jde vlastně o vytvoření interface, který bude nová funkce využívat. Jedná se např. o JSR-339 (JAX-RS) [10], který definuje tvorbu HTTP endpointů pomocí anotací (např @GET pro HTTP GET metodu.). Toto je pak implementováno např. Jersey (součást Glassfish implementace Jakarta EE).

V rámci přidávání nových funkcí se ale často osvědčil implementation-first přístup, což je případ JSR-310 – přidání java.time v Java SE 8 [10]. Před Java 8 byla spousta nedostatků v základních třídách ohledně manipulace s časem v java.util. Pro vyřešení těchto nedostatků bylo použití knihovny Joda Time velmi populární, a vyšlo z ní právě JSR-310, které je implementované standardně v Javě. Proto je cílem EFSP: „EFSP bude založen na praktickém experimentování a kódování jako způsobu, jak prokázat, že něco stojí za zdokumentování ve specifikaci.“ [10].

4.3.3 TCK

Technology Compatibility Kit (TCK) slouží k otestování funkcí, které jsou definovány JSR. Aby aplikační servery splnili soulad s Java EE, musí těmito testy projít (a tím dokážou, že správně implementují všechny JSR) [10].
Oracle tedy udělal TCK a všechny JSR open-source [10]. Díky tomu mohou být použity Eclipse Foundation. Další nové TCK budou již také open-source.

4.4 Implementace Jakarta EE

Aplikační servery, které správně implementují JSR specifikace a projdou přes všechny TCK, mohou získat po zažádání a splnění podmínek od Eclipse Foundation certifikaci.

Jakarta EE 10 Platform Compatible Products

Product		Certification Results
	Eclipse GlassFish	7
	FUJITSU Software Enterprise Application Platform	1.2.0 1.1.0
	IBM WebSphere Liberty	23.0.0.3, Java 17 23.0.0.3, Java 11
	Open Liberty	23.0.0.3, Java 17 23.0.0.3, Java 11
	Payara Server Community	6.2023.7 6.2022.1 6.2022.1.Alpha4
	Payara Server Enterprise	6.4.0
	WildFly	27.0.0.Alpha5, Java SE 17 27.0.0.Alpha5, Java SE 11

Obrázek 4: Certifikované servery pro Jakarta EE 10 [14].

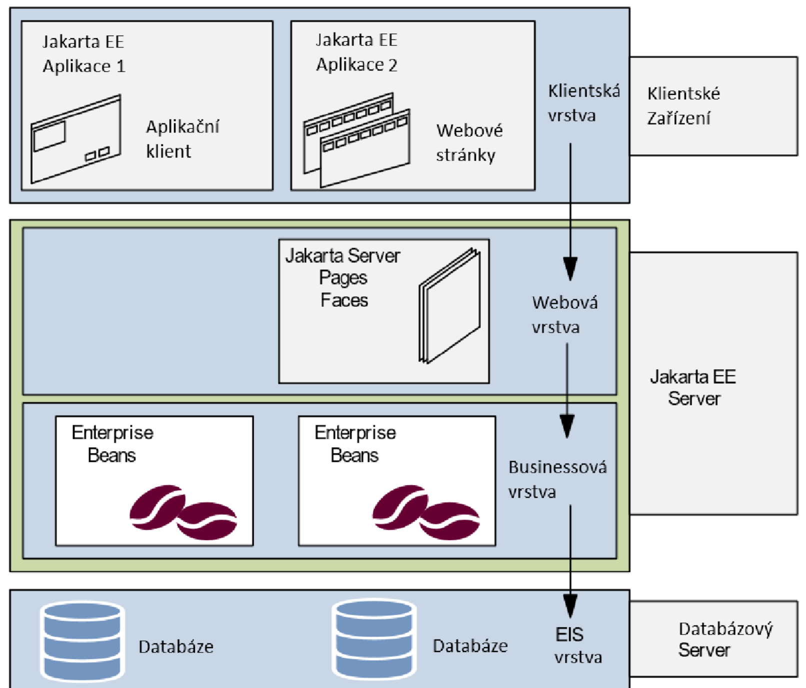
Certifikovaných serverů je více a mohou se dělit také podle profilů. Např. Web Profile je nejlehčí – nemá v sobě tolik implementací pro Jakarta EE specifikace, ale stačí pro REST API aplikaci s nějakou perzistencí. V rámci Jakarta EE 9.1 je ještě TomEE server, který vychází z Tomcatu (navíc má oproti němu v sobě další implementace Jakarta EE specifikací – pro EJB a CDI beany, OpenJPA pro JPA a další).

4.5 Glassfish

Glassfish byl dříve pod správou Oracle (v tu dobu Oracle Glassfish). Jednalo se o referenční implementaci pro JCP, což bylo sice jednoduší pro implementátory, ale docházelo k tendenci upřednostňovat jednu implementaci před druhou [10]. Po tranzici z Java EE na Jakarta EE je již pod správou Eclipse Foundation. Glassfish 5.1 byla kompatibilní s Java EE 8 a zároveň s Jakarta EE 8 (byl zde použit javax.* namespace). Glassfish 6.0 již byla kompatibilní s Jakarta EE 9, a proto použila již jakarta.* namespace. Nejnovější verze je pak Glassfish 7 (kompatibilní s Jakarta EE 10).

4.6 Architektura

Architektura je vícevrstvá, kdy u každé vrstvy je zamýšlen nějaký účel. Díky tomu by se mělo dát při vývoji lépe orientovat.



Obrázek 5: Jakarta EE architektura [15] *(přeloženo)*.

4.7 Popis vrstev architektury

4.7.1 Klient

V této vrstvě jsou komponenty/technologie, které běží přímo na klientském zařízení [16]. Může to být webový prohlížeč, desktopová aplikace a v dnešní době často i mobilní aplikace.

4.7.2 Webová vrstva

Tato vrstva slouží k řešení webových požadavků [16]. Zahrnuje Servlety, JavaServer Pages (JSP), JavaServer Faces (JSF). Dochází zde tedy k přijímání webových požadavků od klienta a k odpovědím (kdy ta odpověď může být právě třeba webová stránka vygenerovaná přes JSP nebo JSF) na ně.

4.7.3 Businessová vrstva

V této vrstvě jsou komponenty, které řeší logiku aplikace a přístup k datům [16]. Zde se tedy zpracovávají data podle námi určených pravidel. Lze zde zmínit Enterprise JavaBeans (EJB), které nám zajišťuje správu těchto komponent.

Tato vrstva se může rozdělit na dvě části, a to na Business Logic vrstvu a persistentní vrstvu [17]. V rámci Business Logic vrstvy se řeší přímo aplikační logika, zatímco ve vrstvě persistentní se řeší přístup k datům. Přístupu k datům lze docílit pomocí Java Persistence API (JPA), jenž umožňuje mimo jiné namapovat databázové tabulky na objekty v Javě.

4.7.4 Databázová (EIS) vrstva

Vrstva sloužící k získání a ukládání dat. Může se jednat o nějaký databázový systém (třeba nějaká SQL databáze), ale i např. LDAP [16]. Může běžet lokálně (na stejném zařízení kde běží Java EE server), nebo na jiném serveru.

4.8 Příklady technologií Java EE

Java EE poskytuje velké množství technologií. Zde jsou některé příklady:

4.8.1 Java Servlets

Servlet slouží k zpracování webového (HTTP/S) požadavku, a případné odpovědi na něj. Tyto požadavky jsou zpracovány podle nějakých námi určených pravidel (programové logiky).

```
@WebServlet( name = "AnnotationExample",
  description = "Example Servlet Using Annotations",
  urlPatterns = {"/AnnotationExample"}
)
public class Example extends HttpServlet {

    @Override
    protected void doGet(
      HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
                                                IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<p>Hello World!</p>");
    }
}
```

Příklad 5: Ukázka Servletu v Java EE [18].

Jednoduchá ukázka Servlet třídy, kdy při přístupu na danou URL (např. localhost:8080/AnnotationExample) je odpovědí HTML stránka s jedním odstavcem s textem „Hello World!“.

4.8.2 JavaServer Pages (JSP)

Umožní nám vytvářet dynamické stránky díky možnosti použít k HTML kódu i Java kód, který se zapisuje ve speciální direktivě [19]. Tato výsledná kombinace je uložena v JSP souboru. Jedná se o abstrakci nad Servletem, protože při kompilaci se na základě tohoto JSP souboru vytvoří Servlet.

4.8.3 JSF (Java Server Faces)

Modernější alternativa k JSP. [19] Webová stránka je tvořena XML souborem s příponou XHTML. Webová stránka se tvoří využíváním předpřipravených komponent (např. formulářů, tlačítek).

4.8.4 Enterprise JavaBeans (EJB)

Slouží k vytváření komponent (bean), jež budou ve svých atributech/metodách implementovat další komponenty [20]. Díky tomu je lze „skládat" podle potřeby pomocí Dependency Injection principu (který je u EJB i CDI podobný, jako ve Spring Frameworku, viz. Dependency Injection (DI)). Také řídí životní cyklus těchto komponent (např. pomocí metody v beaně s anotací @PostConstruct nebo @PreDestroy).

Typy EJB bean jsou Singleton, Stateful, Stateless. Singleton bean funguje tak, že se v rámci EJB kontejneru používá pouze jedna její instance na všech místech. Stateful bean má zase pro každého jejího EJB klienta (např. instance servletu) vlastní instanci. Stateless beany jsou „sdílené" mezi jejími EJB klienty a v případě skončení instance jejího klienta může instanci beany použít jiný klient. Když instance stateless beany není v daném čase v poolu dostupná, vytvoří se instance nová.

4.8.5 Contexts and Dependency Injection (CDI)

Jednodušší a modernější alternativa k EJB [19]. Hlavní rozdíl oproti EJB je, že při tvorbě beany se určuje, v jakém kontextu bude platit [21].

V rámci webové aplikace je tedy několik možností životnosti komponenty. Může být platná v rámci celé aplikace (třeba poskytnutí zpracování dat, které uvidí všichni uživatelé, lze řešit ApplicationScoped bean), nebo bude platná jenom v jedné relaci (tedy např. práce s daty jednoho uživatele se řeší v SessionScoped bean). Může ale také být platná jenom pro jeden HTTP request (RequestScoped bean), kdy každý ten request (i v rámci jedné relace) se musí zpracovat zvlášť.

4.8.6 Ukázka Dependency Injection v Java EE

```
//CDI
//@ApplicationScoped
//@SessionScoped
//@RequestScoped

//EJB
//@Stateless
//@Stateful
@Singleton
public class EJBCounter implements Serializable
 {
    private int counter = 0;

    //Lock resime pouze v pripade @Singleton
    @Lock(LockType.WRITE)
    public void add() throws InterruptedException {
        long delay = 5000;
        Thread.sleep(delay);
        counter++;
    }

    //Lock resime pouze v pripade @Singleton
    @Lock(LockType.READ)
    public int getCounter() {
        return counter;
    }
}

@Path("/counter")
public class CounterResource {

    @Inject //Lze pouzit i @EJB u EJB Bean
    private EJBCounter ejbCounter;

    @GET
    @Produces("text/json")
    public String increment() throws InterruptedException {
        ejbCounter.add();
        int number = ejbCounter.getCounter();
        return "{\"number\": " + number + "}";
    }
}

@ApplicationPath("/api")
public class CounterApplication extends Application {

}
```

Výsledky po paralelních přístupech na daný HTTP GET endpoint:

```
@Singleton nebo @ApplicationScoped
{"number": 1}, {"number": 2}

@Stateful nebo @RequestScoped
{"number": 1}, {"number": 1}

@Stateless
(1. batch): {"number": 1}, {"number": 1}
(2. batch): {"number": 2}, {"number": 2}

@SessionScoped
(1. prohlížeč): {"number": 1}, {"number": 2}
(2. prohlížeč): {"number": 1}, {"number": 2}
```

Příklad 6: Ukázka EJB/CDI Dependency Injection *(Zdroj: vlastní zpracování)*.

Tato aplikace obsahuje beanu čítače, který se po přístupu na get HTTP endpoint inkrementuje. Díky tomu zde lze vidět chování různých typů bean. V Singleton bean lze řešit zamykání. V případě použití metody s WRITE zámkem nelze použít jiné metody s WRITE zámkem a READ zámkem.

V případě těchto výsledků pro paralelní přístup na danou HTTP metodu má Stateful a RequestScoped bean stejný výsledek. Rozdíl mezi nimi je ale v tom, že RequestScoped bean platí pro jeden request. Oproti tomu Stateful bean lze použít buď přímo v rámci aplikace, nebo je možné použít jich více např. v jiné RequestScoped/SessionScoped bean. Singleton a ApplicationScoped beany mají stejný výsledek, rozdíl je v tom, že ApplicationScoped bean není thread-safe (nemá locky). SessionScoped bean má pak svůj stav pro každý prohlížeč (relaci). U Stateless bean je zase vidět, že ze základu se vytvoří dvě instance (protože je tam uspání vlákna) a u další batche se použijí opět tyto dvě instance.

4.8.7 Java Persistence API (JPA)

API, které poskytuje možnosti komunikace s databází [20]. Poskytuje možnosti k tomu, že lze namapovat třídy na tabulky v relační databázi a dále takto s databází pracovat.

```
@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String firstName;
  private String lastName;

  protected Customer() {}

  public Customer(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public String toString() {
    return String.format(
        "Customer[id=%d, firstName='%s', lastName='%s']",
        id, firstName, lastName);
  }

  public Long getId() {
    return id;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }
}
```

Příklad 7: Ukázka Entity třídy mapující SQL tabulku [22].

Příklad ukazuje mapování tabulky na třídu pomocí JPA API. Jsou zde dva sloupce (firstName, lastName) a také primární klíč id tvořený pomocí sekvence.

4.9 Důvody k použití Java EE

Použití Java EE je zamýšleno tak, aby aplikace v něm vyvíjená mohla mít tyto vlastnosti [9]:

- škálovatelná
- víceúrovňová
- robustní
- síťově bezpečná

Díky velkému množství jejích technologií usnadňuje vývoj náročných aplikací a zvyšuje produktivitu. Jak bylo zmíněno přímo u daných technologií, jedná se např. o řízení transakcí, webové rozhraní, napojení na databázi apod.

5 Spring Framework

Spring Framework je open-source Java framework, který poprvé vydal v roce 2002 Rod Johnson [23]. Je pod licencí Apache License 2.0. [24].

5.1 Historie

Rod Johnson psal knihu ohledně používání určitých konceptů (Dependency Injection a používání čistých Java objektů – POJO) v rámci tvorby Java EE aplikací. Napsal přibližně 30 tisíc řádků kódu, jak by to mohlo vypadat a bylo to základem pro vytvoření Spring Frameworku [23]. Vývoj Spring Frameworku v tuto chvíli zajišťuje VMware [25]. Aktuálně podporovaná verze je 5.3 (podporuje Javu 8, 11, 17), dále 6.0 (podporuje Javu 17) [26].

5.2 Aplikační servery

Nejjednodušší způsob, jak vytvořit webovou aplikaci využívající Spring je pomocí Spring Boot. Spring Boot má v sobě vnořený server, a při spuštění sestaveného jar souboru dojde k spuštění tohoto vnořeného serveru na námi určené adrese/portu. Ze základu používá Tomcat, ale podporuje i Jetty, Undertow nebo Netty.

Spring lze ale použít i bez Spring Boot. V tomto i předchozím případě pak můžeme aplikaci spustit běžným způsobem např. v (nevnořeném) Tomcatu nebo v našem případě ve Wildfly.

5.3 Základní principy Spring Frameworku

5.3.1 Inversion of Control (IoC)

Je to princip, kdy se inicializace objektů přenechává na nějakém kontejneru/frameworku [27]. Dojde tedy k rozdělení vykonání úlohy od implementace toho, jak je vykonána. Díky tomu se dosáhne lepší modularity aplikace a existuje možnost, jak jednoduše změnit implementaci daného rozhraní (vytvoření nové třídy implementující dané rozhraní).

5.3.2 Dependency Injection (DI)

Je to způsob, kterým se dá implementovat IoC [27]. V rámci kontextu aplikace (což je vlastně IoC kontejner, který může být určen např. XML souborem nebo anotacemi) se komponenty (beany) inicializují a dávají se (injection) do jiných komponent v tom kontextu. Framework pak řeší, které objekty se inicializují jako první (aby to mohlo fungovat). Zároveň pomocí změny kontextu (např. přepsáním XML souboru) lze měnit implementace různých injektovaných (vnořených) rozhraní v rámci aplikace. Je zde možnost určení rozsahu (pomocí anotace @Scope) daných komponent, podobně jako u CDI v Java EE. Oproti EJB v Java EE zde není alternativa k Stateless beans. V rámci běžné bean (singleton) je nutné oproti Java EE řešit thread safety ručně.

```
public interface Counter {
    int getCount();

    void addOne();

    void reset();

}



//Anotace určující, že je toto bean
@Component
public class CounterImpl implements Counter {

    //Po vytvoření beany (např. po spuštění aplikace, nebo získání
    nové instance z kontextu) se vypíše
    @PostConstruct
    public void init() {
        System.out.println("Bean Initialized!");
    }

    private int count = 0;

    @Override
    public int getCount() {
        return count;
    }

    @Override
    public void addOne() {
        count++;
    }

    @Override
    public void reset() {
        count = 0;
    }

}
```

```
//Anotace určující, že je toto bean
@Service
public class CounterService {

    //inject beany implementující Counter interface
    @Autowired
    private Counter counter;

    public void addOne() {
        counter.addOne();
    }

    public String getTextResultAndReset() {
        String result = "Counter result is " + counter.getCount()+
        "!";
        counter.reset();
        return result;
    }
}
```

Příklad 8: Ukázka Dependency Injection ve Springu *(Zdroj: vlastní zpracování)*.

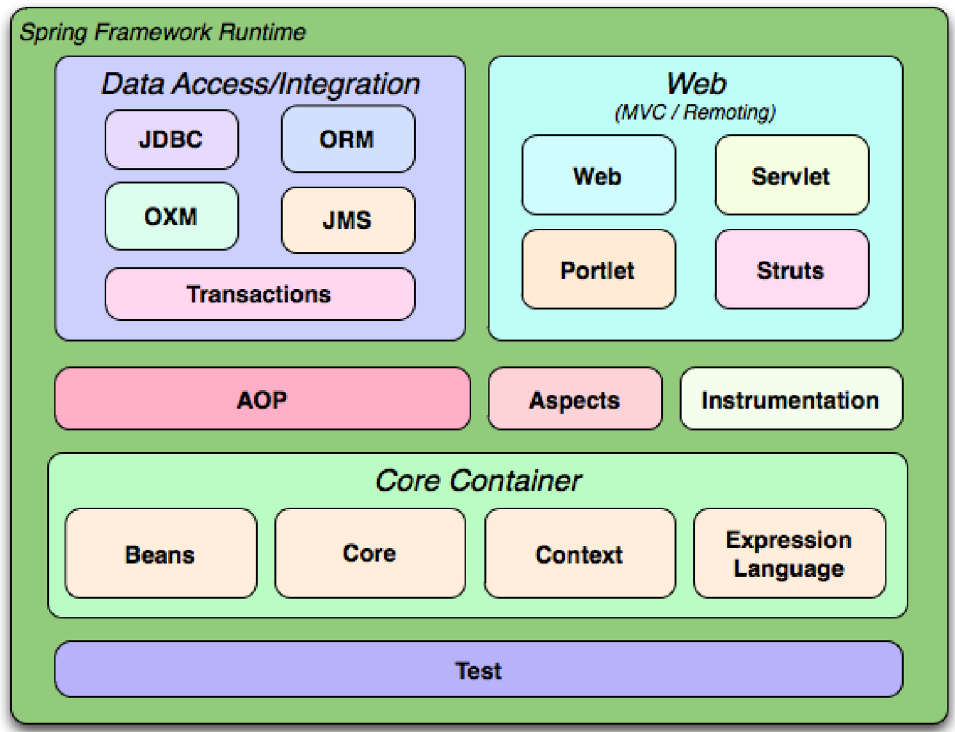
V této ukázce lze vidět interface Counter, který je implementován v třídě CounterImpl, jenž je bean. Dále je tu bean CounterService, jež injektuje bean implementující Counter interface (což je v tomto případě právě CounterImpl). V případě, že nastane situace, kdy bude potřeba změnit implementaci Counteru, tedy např. že by aktuální hodnota byla uložena v databázi, tak vývojář vytvoří novou implementaci a injectne (vnoří) se místo té původní implementace (když je více bean jednoho interface, je možnost je rozdělit názvem, takže by se použila ta s jiným názvem). V příkladu je DI provedena pomocí atributu. Jde to i pomocí konstruktoru nebo setteru.

5.3.3 Aspect-Oriented Programming (AOP)

Jde o programovací paradigma s cílem zvýšit modularitu. K tomu dochází rozdělením vykonání situací na různé části [28]. V rámci AOP se definuje aspect. Ten se provede v určité části vykonávání programu při splnění nějaké podmínky. To lze použít např. pro zalogování při určité situaci.

5.4 Architektura

Architektura je modulární, což znamená, že v případě potřeby lze přidat námi požadovaný modul. Obrázek ukazuje moduly rozdělené do skupin podle jejich účelu.



Obrázek 6: Spring Framework architektura [29].

5.5 Příklady modulů v architektuře

5.5.1 Spring Core a Spring Beans

Toto jsou hlavní moduly Spring Frameworku. Umožňují nám vytvářet beany. Poskytují IoC, a tím pádem i dependency injection. Jednou z částí je zde interface BeanFactory, který řídí životní cyklus bean [29].

5.5.2 Spring Context

Modul navazující na Core a Beans moduly [29]. Umožňuje nám manipulaci s komponenty (beany). Hlavní částí je zde interface ApplicationContext, který právě využívá BeanFactory. Např. zde jde docílit jazykové lokalizace, kdy se v rámci ApplicationContext nastaví aktuální jazyk, díky čemuž se následně použije zdroj textů daného jazyka. Také existuje možnost využívat eventy, na něž můžou beany reagovat (jsou zde základní eventy, např. při startu ApplicationContext, ale jde definovat i vlastní).

5.5.3 Spring Expression Language

Umožňuje manipulovat s objekty za běhu pomocí výrazů zadaných v textovém řetězci.

```
@Value("#{someBean.someProperty   !=   null   ?   someBean.someProperty   :
'default'}")
private String ternary;
```

Příklad 9: Ukázka Spring Expression Language [30].

V tomto příkladu je hodnota atributu komponenty nastavena podle toho, zda jiná komponenta má nastaven atribut null. V případě, že je null, tak hodnota bude ‚default‘, jinak bude hodnota hodnotou té původní komponenty.

5.5.4 Spring Data

Modul pro přístup k datům. Různé implementace podle toho, k čemu se přistupuje. Např. Spring Data JPA nebo jednodušší Spring Data JDBC pro přístup k relační databázi. Ale také třeba Spring Data MongoDB pro přístup k MongoDB.

```
public interface CustomerRepository extends CrudRepository<Customer,
Long> {

  List<Customer> findByLastName(String lastName);

  Customer findById(long id);
}
```

Příklad 10: Ukázka interface repozitáře Spring Data JPA [22].

Ukázka navazující na entitu vytvořenou pomocí JPA API (viz. Příklad 7). Pomocí Spring Data JPA lze dělat jednoduché query pro operaci s tabulkou v databázi pomocí názvů metod (v složitějších případech lze pak použít JPQL query). V námi vytvořeném interface stačí rozšířit CrudRepository (nebo JpaRepository obsahující ještě navíc sorting a paging možnosti). Do jeho generických parametrů se dodá entita a datový typ primárního klíče. Pak už tvoříme metody, v tomto případě vyhledání pomocí příjmení a také pomocí id. Interface se poté použije jako bean pomocí anotace @Repository.

5.5.5 Spring OXM

Modul umožňující pracovat s XML soubory. XML soubor se konvertuje do Java objektu (unmarshalling), nebo naopak (marshalling).

5.5.6 Webová část (Spring Web)

Pro zpracování HTTP požadavků. Stejně jako v Java EE se dají použít přímo Servlety, ale také např. RestController z modulu Spring MVC.

```
@RestController
public class RestAnnotatedController {
    @GetMapping(value = "/annotated/student/{studentId}")
    public Student getData(@PathVariable Integer studentId) {
        Student student = new Student();
        student.setName("Peter");
        student.setId(studentId);

        return student;
    }
}
```

Příklad 11: Ukázka Controlleru v Spring MVC [31].

Ukázka obsahuje základní HTTP GET metodu, vracející objekt student (POJO objekt třídy Student se dvěma atributy). Ze základu je výstup vracen ve formátu JSON, ale je možné nastavit třeba i XML apod.

5.5.7 Testovací část (Spring Test)

Tento modul podporuje testování Spring aplikace pomocí např. JUnit. Umožňuje manipulovat s ApplicationContext v rámci testů. Díky tomu se kontext může znovupoužít (když na sebe testy navazují). [29]

5.5.8 Spring Boot

Spring Boot je velké zjednodušení pro vývoj aplikací využívajících Spring Framework, aby mohlo být více času využito vývojem, než správou knihoven [32]. Jedná se o pohled na Spring a další knihovny podle názoru vývojářů Springu, aby bylo možné rychle tvořit aplikace použitelné pro produkční prostředí [33]. Spring Boot aplikace lze jednoduše spustit jako jar soubor (má v sobě vnořený server, který lze vybrat z několika možných).

Existuje spoustu Spring Boot Starter knihoven pro různé účely, díky kterým lze docílit jednoduše potřebného účelu (je usnadněna celková konfigurace oproti samotnému použití knihoven). Také se docílí sjednocení struktury na různých vývojových projektech.

Dalším cílem také je, aby nebylo při vývoji vyžadováno používání XML konfigurace nebo generování kódu [33]. Je díky němu také snadné přidat funkce rovnou připravené do produkčního prostředí jako metriky, health checky a externí konfigurace [33].

Spring Boot má oproti verzím samotných Spring knihoven jiné verzování. Při jejich kombinování je tak nutné dávat pozor na to, aby byly knihovny správně kombinovány.

Spring Boot Starter knihovny přímo patřící pod Spring Boot je možné rozdělit na 3 skupiny, a to na Application Starters, Production Starters a Technical Starters [32].

5.5.9 Spring Boot Application Starters

Knihovna spring-boot-starter je základním jádrem Spring Bootu. Dále důležitými knihovnami je spring-boot-starter-web (starter pro Spring Web), spring-boot-starter-jpa (starter pro Spring Data JPA), spring-boot-starter-jdbc (starter pro Srping Data JDBC) a spousta dalších.

5.5.10 Spring Boot Production Starters

V této skupině je spring-boot-starter-actuator, který obsahuje production-ready funkce pro snadný monitoring a konfiguraci aplikace.

5.5.11 Spring Boot Technical Starters

Tyto knihovny slouží k tomu, že díky nim je umožněno vyměnit implementaci některých částí základní spring-boot-starter knihovny. Konkrétně jde o knihovny pro logování a vnořené servery. Lze tak vyměnit např. původní vnořený server spring-boot-server-tomcat (excludne se v spring-boot-starter) za spring-boot-starter-jetty. Stejným způsobem lze docílit i výměnu logovací knihovny spring-boot-starter-logging (jde o logback implementaci) za spring-boot-starter-log4j2 (log4j2 implementace).

5.6 Důvody k použítí Spring Frameworku

Některé důvody k použití Spring Frameworku[17]:

- Využívá POJO (Plain Old Java Object), což dělá framework jednoduchým. Díky tomu se dá aplikace různě dělit a některé třídy se mohou použít v aplikaci i přímo bez Spring Frameworku.
- Flexibilita – možnost při vývoji použít buď XML, nebo anotace.
- Zjednodušení použití různých částí Java EE, což je také primárním důvodem jeho vzniku.

Podle JVM Ecosystem Report 2021 [34] je Spring mezi Java vývojáři výrazně populárnější než přímo použít Java EE. Dle tohoto reportu 58% vývojářů využívá Spring Boot a Spring MVC 29%, zatímco Java EE 24% a Jakarta EE 13%.

6 Alternativní frameworky

6.1 Ktor

Ktor je framework založený na programovacím jazyku Kotlin pro tvorbu webových aplikací [35] (na straně klienta i serveru). Jak Ktor, tak i Kotlin byl vytvořen společností JetBrains (ta vytvořila i známé IDE pro vývoj v Javě IntelliJ Idea).

Kotlin je programovací jazyk, který byl vytvořen jako vylepšení Javy, je s Javou plně kompatibilní (v jednom projektu lze kombinovat s Java kódem). Je kompilován do Java byte-kódu (pouští se stejně jako Java v JVM). Hlavní knihovny a ty podporované lze kompilovat také do Javascriptu.

Co se Ktor frameworku týče, je rozdělen na dvě části, Ktor Server a Ktor Client. Ktor Server slouží k tvorbě HTTP a WebSocket backendu. Ktor Client pak slouží pro přístup k HTTP a WebSocket službám. U Client části je zajímavostí, že ho lze použít pro tvorbu víceplatformních aplikací (v kombinaci s dalšími balíčky lze použít i pro tvorbu JavaScript front-endu nebo mobilních aplikací pro iOS a Android). Ktor ze základu neobsahuje Dependency Injection, pro její použití je nutné použít jeden z několika balíčků pro to určených, např. Kain. Dokumentace[35] Ktor frameworku je přehledná, hlavně u Server části je v ní přehledně ukázáno, jak co vytvořit.

6.2 Quarkus a Micronaut

Jedná se o Java frameworky pro vývoj webových aplikací, jejichž výhody jsou využity při použití Microservice architektury [36]. Cíl je, aby aplikace zabírala málo místa a rychle se spouštěla oproti rozsáhlým frameworkům jako je Spring. Toho je docíleno díky optimalizacím a podpoře kompilace do nativního kódu (místo byte-kódu běžícího v JVM).

7 Vývoj Spring aplikace

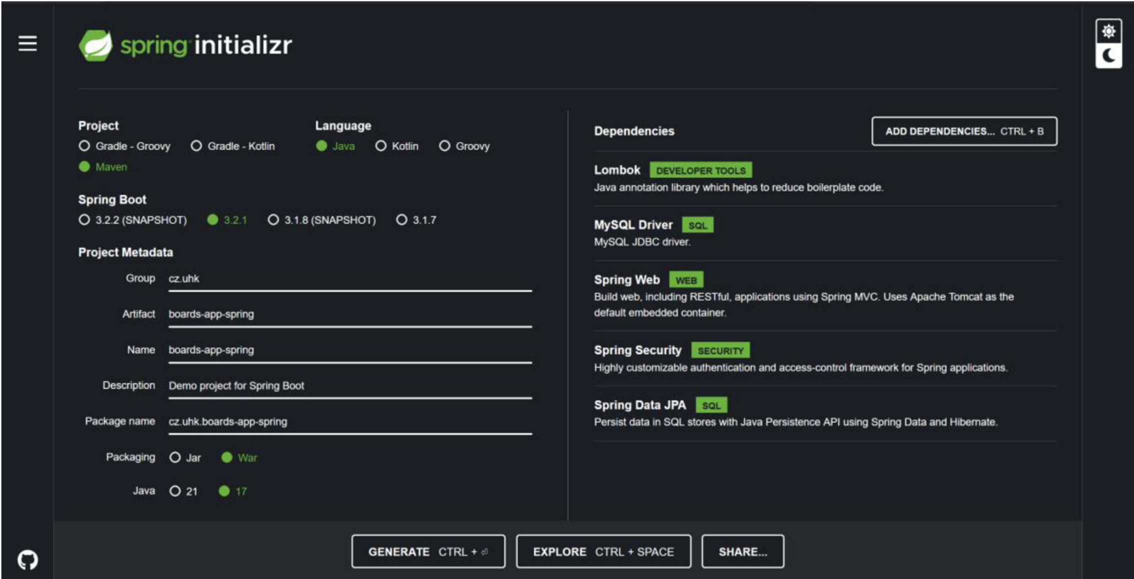
7.1 Inicializace

V prvním kroku vývoje byla zprovozněna databáze MySQL Community 8.0.35. Bude zde vytvořeno schéma boards-app, ve kterém budou vytvořeny tyto tabulky:

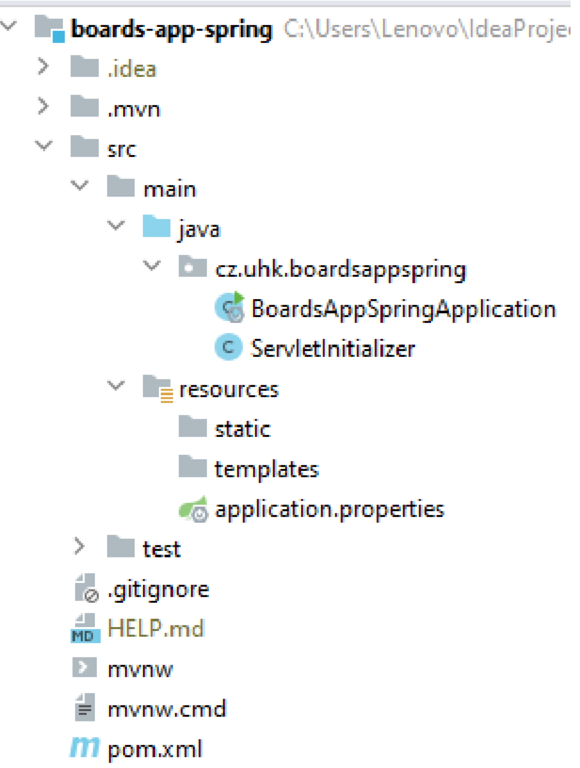
```
CREATE TABLE IF NOT EXISTS Users (
    username VARCHAR(50) PRIMARY KEY,
    password VARCHAR(500) NOT NULL,
    enabled BOOLEAN NOT NULL DEFAULT 1
);

CREATE TABLE IF NOT EXISTS Authorities (
    username VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    PRIMARY KEY (username, authority),
    FOREIGN KEY (username) REFERENCES Users(username)
);

CREATE TABLE IF NOT EXISTS Posts (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    author VARCHAR(50) NOT NULL,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    removed BOOLEAN NOT NULL DEFAULT 0,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (author) REFERENCES Users(username)
);

CREATE TABLE IF NOT EXISTS Comments (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    author VARCHAR(50) NOT NULL,
    content TEXT NOT NULL,
    removed BOOLEAN NOT NULL DEFAULT 0,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (author) REFERENCES Users(username),
    FOREIGN KEY (post_id) references Posts(id)
);
```

Příklad 12: SQL pro vytvoření potřebných tabulek *(Zdroj: vlastní zpracování)*.

Následně pomocí Spring Initializr se vytvoří základní balíček naší aplikace. V pom.xml budou rovnou přítomny dependencies, které si zde navolíme.



Příklad 13: Inicializace Spring Boot aplikace pomocí Spring Initializr



Příklad 14: Počáteční struktura projektu *(Zdroj: vlastní zpracování)*.

7.2 Perzistence

Nyní je potřeba vytvořit JPA entity, které budou namapovány na SQL tabulky. S těmito entitami se bude pracovat v DAO (Data Access Object) třídách. DAO třídy, které pracují s EntityManager třídou, se použijí místo Repository interface poskytnutého Spring Data JPA (což je vlastně vrstva právě nad EntityManagerem). Je to z důvodu, aby se tyto třídy daly využít i v aplikaci vyvinuté v Jakarta EE. Budou použity anotace dependency Lombok, pro snadnější a přehlednější tvorbu redundantních základních věcí (constructors, getters, setters).

```
@Data
@MappedSuperclass
public abstract class AbstractUserContent implements UserContent {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author")
    private User author;

    private String content;

    @Column(insertable = false)
    private boolean removed;

    @Column(insertable = false, updatable = false)
    private Timestamp createdAt;

    @Column(insertable = false, updatable = false)
    private Timestamp updatedAt;
}

@Data
@EqualsAndHashCode(callSuper = true)
@Entity(name="Comments")
public class Comment extends AbstractUserContent {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;
}
```

Příklad 15: Ukázka namapované entity na tabulku Post, využití dědičnosti *(Zdroj: vlastní zpracování)*.

V příkladu je vidět, jak Lombok anotace zjednodušily kód a zmenšily jeho délku. V rámci multiplicitních vztahů se používá FetchType.LAZY, aby nedošlo k problémům s výkonem (V tomto případě tedy příspěvek bude získán pomocí selectu z DB až po zavolání getteru getPost).

Jako základ DAO třídy bude tato abstraktní třída, která bude děděna třídami pro dané tabulky.

```
public abstract class AbstractJpaDAO< T, U > {

    private Class< T > clazz;

    @PersistenceContext
    EntityManager entityManager;

    public final void setClazz( Class< T > clazzToSet ){
        this.clazz = clazzToSet;
    }

    public T findOne( U id ){
        return entityManager.find( clazz, id );
    }

    public T getReference ( U id ){
        return entityManager.getReference( clazz, id );
    }

    public List< T > findAll(){
        return entityManager.createQuery( "from " + clazz.getName()
        )
        .getResultList();
    }

    public void create( T entity ){
        entityManager.persist( entity );
    }

    public T update( T entity ){
        return entityManager.merge( entity );
    }

    public void delete( T entity ){
        entityManager.remove( entity );
    }

    public void deleteById( U entityId ){
        T entity = getReference( entityId );
        delete( entity );
    }
}
```

Příklad 16: Generická DAO třída, upravena [37].

V rámci DAO tříd lze i jednoduše implementovat stránkování:

```
Query query = entityManager.createQuery("From Foo");
int pageNumber = 1;
int pageSize = 10;
query.setFirstResult((pageNumber-1) * pageSize);
query.setMaxResults(pageSize);
List <Foo> fooList = query.getResultList();
```

Příklad 17: Pagination pomocí JPA Criteria API [38].

Vlastní implementace DAO třídy, konkrétně pro komentáře:

```
@Repository
public class CommentDAO extends AbstractJpaDAO<Comment,Long> {
    public CommentDAO() {
        setClazz(Comment.class);
    }

    public Long createAndReturnId(Comment comment) {
        entityManager.persist(comment);
        entityManager.flush();
        return comment.getId();
    }

    public List<Comment> findVisibleCommentsByPostId(Long postId) {
        return getVisibleCommentsByPostIdSelectQuery(postId)
                .getResultList();
    }

    public List<Comment> findVisibleCommentsByPostId(Long postId,
        int pageNumber, int pageSize)
    {
        return getVisibleCommentsByPostIdSelectQuery(postId)
                .setFirstResult((pageNumber-1)*pageSize)
                .setMaxResults(pageSize)
                .getResultList();
    }

    @Override
    public Comment findOne(Long id) {
        return entityManager.createQuery("select c from Comments c
            join fetch c.author join fetch c.post where
            c.id=:idParam", Comment.class)
            .setParameter("idParam", id)
            .getSingleResult();
    }

    private TypedQuery<Comment>
            getVisibleCommentsByPostIdSelectQuery(Long postId)
    {
        return entityManager
            .createQuery("select c from Comments c join fetch
            c.author where c.post.id=:postIdParam and
            c.removed=false order by c.createdAt asc",
            Comment.class)
            .setParameter("postIdParam", postId);
    }
}
```

Příklad 18: Vlastní implementace DAO třídy *(Zdroj: vlastní zpracování)*.

V DAO třídě pro komentáře a příspěvky je použito řazení podle created_at TIMESTAMP sloupce. V případě příspěvků se řadí od nejnovějších (descending) a v případě komentářů naopak. Také v rámci query je kvůli optimalizaci použit join fetch, kdy hlavním důvodem je to, aby nedošlo k rozdělení query (nejdříve by se načetli komentáře a pak až uživatelé pro každý komentář zvlášť, potřeba ale je kvůli rychlosti načíst obojí najednou). Také tím dojde k vyřešení případné LazyInitializationException – k komentářům budou

načteni uživatelé a tudíž s nimi bude možno pracovat. Je zde metoda k získání komentářů se stránkováním, anebo pouze získání všech komentářů k příspěvku. S komentáři je načten uživatel, ale nejsou načteny role uživatele (což je kolekce v uživateli) – v případě, že by se načetly i role, došlo by k n+1 problému.

7.3 DTO třídy

Následně budou potřeba DTO (Data Transfer Object) třídy pro účel použití v rámci Service tříd a Controlleru. Budou používány, jako reprezentace vstupních a výstupních dat. K převodu dojde pomocí mapper tříd.

```
@Data
public class InformationUserDTO {

    private String username;
    private boolean enabled;
    private List<AuthorityDTO> authorities;
}
```

Příklad 19: DTO třída uživatele *(Zdroj: vlastní zpracování)*.

V DTO třídě uživatel není heslo. Je to právě to, proč se používá (v JSONu u příspěvků nesmí být vidět heslo uživatele). V DTO třídě uživatele pro registraci a přihlášení pak heslo potřeba je.

```
@Data
public class LoginUserDTO {
    private String username;
    private String password;
}
```

Příklad 20: DTO třída uživatele pro přihlášení a registraci *(Zdroj: vlastní zpracování)*.

Pro tvorbu mapper tříd je dobré použít generování kódu, a to kvůli úspoře času. Výhodou také je, že když se změní implementace třídy z/do které se konvertuje, nově vygenerovaný kód mapper třídy tuto změnu reflektuje. K tomuto účelu bude použita knihovna MapStruct. Pro její konfiguraci v pom.xml je potřeba změnit plugin z spring-boot-maven-plugin na maven-compiler-plugin.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.12.1</version>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${org.mapstruct.version}</version>
                    </path>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
                    <dependency>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok-mapstruct-binding</artifactId>
                        <version>0.2.0</version>
                    </dependency>
                </annotationProcessorPaths>
                <compilerArgs>
                    <compilerArg>
                        -Amapstruct.defaultComponentModel=spring
                    </compilerArg>
                </compilerArgs>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Příklad 21: Nastavení maven-compiler-plugin pro Lombok a MapStruct podle [39].

Mapper interface pro jednoduchou třídu s jedním parametrem a podle toho interface vygenerovaná implementace:

```
@Mapper
public interface AuthorityMapper {
    AuthorityDTO authorityToAuthorityDTO(Authority authority);
    Authority authorityDTOToAuthority(AuthorityDTO authorityDTO);
}

@Component
public class AuthorityMapperImpl implements AuthorityMapper {

    @Override
    public AuthorityDTO authorityToAuthorityDTO(Authority authority)
    {
        if ( authority == null ) {
            return null;
        }

        AuthorityDTO authorityDTO = new AuthorityDTO();

        authorityDTO.setAuthorityName( authority.getAuthorityName()
        );

        return authorityDTO;
    }

    @Override
    public Authority authorityDTOToAuthority(AuthorityDTO
    authorityDTO)
    {
        if ( authorityDTO == null ) {
            return null;
        }

        Authority authority = new Authority();

        authority.setAuthorityName( authorityDTO.getAuthorityName()
        );

        return authority;
    }
}
```

Příklad 22: Námi definovaný mapper interface a jeho vygenerovaná implementace.

Implementace je vygenerována v mavenu při buildu projektu pomocí kroků clean a install (*maven clean install*).

7.4 Service třídy

Nyní je třeba vytvořit service třídy (UserService, PostService, CommentService), které budou používat DAO třídy s dodatečnou logikou a mappery pro transformaci entit z/do DTO tříd.

```
@Service
public class PostServiceImpl implements PostService {

    @Autowired
    private PostDAO postDAO;

    @Autowired
    private UserDAO userDAO;

    @Autowired
    private UserService userService;

    @Autowired
    private PostDTOMapper postDTOMapper;

    @Override
    @Transactional
    public Long addNewPost(NewPostDTO newPostDTO) {
        try {
            Post post = postDTOMapper.newPostDTOToPost(newPostDTO);
            post.setAuthor(userDAO.getReference(
              userService.getCurrentUsername())
            );
            return postDAO.createAndReturnId(post);
        }
        catch (Exception e) {
            e.printStackTrace();
            throw new IllegalStateException("Failed to create
            post");
        }
    }

    @Override
    public List<PostDTO> findVisiblePosts() {
        try {
            return postDAO.findVisiblePosts().stream().map(post ->
            postDTOMapper.postToPostDTO(post)).toList();
        }
        catch (Exception e) {
            e.printStackTrace();
            throw new IllegalStateException("Failed to find visible
            posts");
        }
    }
}
```

Příklad 23: PostService – třída pro operace s příspěvkem, zjednodušeno (ponechány pouze dvě metody – přidání a zobrazení všech příspěvků) *(Zdroj: vlastní zpracování)*.

V Service třídě se řeší základní zachytávání chyb (v případě, že by aplikace měla běžet v produkci, muselo by být uděláno více rozšířeně). V rámci tříd, kde se přímo pracuje s řádky v databázi (přidání, úprava, smazání), je použita anotace @Transactional pro korektní úpravu dat v databázi v případě chyb a paralelního (víceuživatelského) přístupu.

7.5 Controller třídy a jejich zabezpečení

Následuje tvorba Controller tříd, kterými se vytvoří HTTP endpointy. Při přístupu na tyto endpointy se bude pracovat právě s Service třídami.

```
@RestController
@RequestMapping("/api/posts/")
public class PostController {

    @Autowired
    private PostService postService;

    @PostMapping("/new")
    public ResponseEntity addNewPost(@RequestBody NewPostDTO
      newPostDTO)
    {
        try {
            Long id = postService.addNewPost(newPostDTO);
            return ResponseEntity.ok(new SuccessDTO("New post with
            id " + id + " created"));
        }
        catch (IllegalStateException e) {
            return ResponseEntity.badRequest().body(new
            ErrorDTO(e.getMessage()));
        }
    }
    @GetMapping("/all")
    public ResponseEntity findVisiblePosts() {
        try {
            return
            ResponseEntity.ok(postService.findVisiblePosts());
        }
        catch (IllegalStateException e) {
            return ResponseEntity.internalServerError().body(new
            ErrorDTO(e.getMessage()));
        }
    }
}
```

Příklad 24: PostController – tvorba HTTP metod pro práci s příspěvky, zjednodušeno (podle příkladu PostService) *(Zdroj: vlastní zpracování)*.

V příkladu je ukázáno vytvoření HTTP POST endpointu, který na adrese /api/posts/new při obdržení NewPostDTO request body v JSON formátu vytvoří příspěvek. HTTP GET endpoint /api/posts/all zase zobrazí všechny příspěvky. V rámci Controller tříd jsou i metody pro získání komentářů/příspěvků s parametry pro stránkování, úpravu, vymazání (adminem) a další.

Nyní je potřeba ošetřit bezpečnost a přihlašování (aby příspěvky mohli upravovat pouze jejich autoři a mazat pouze administrátoři). Toto bude řešeno pomocí HTTP Basic Security. Jméno a heslo je v takovém případě posíláno v hlavičce HTTP headeru v Base64 kódování (lze převést do plaintextu). V produkčním použití by muselo být použito SSL, aby nemohlo dojít k odchytávání hesel.

```
@Configuration
public class SecurityConfiguration {
    @Bean
    public UserDetailsManager userDetailsManager(DataSource
    dataSource)
    {
        JdbcUserDetailsManager jdbcUserDetailsManager = new
            JdbcUserDetailsManager();
        jdbcUserDetailsManager.setDataSource(dataSource);
        return jdbcUserDetailsManager;
    }
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
    Exception
    {
        return http.csrf(AbstractHttpConfigurer::disable)
                .anonymous(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers(HttpMethod.GET,
                "/").permitAll()
                        .requestMatchers(HttpMethod.POST,
                "/api/posts/new").authenticated()
                        .requestMatchers(HttpMethod.POST,
                "/api/posts/update/*").authenticated()
                        .requestMatchers(HttpMethod.POST,
                "/api/posts/remove/*").hasRole(Role.ADMIN.name())
                        .requestMatchers(HttpMethod.GET,
                "/api/posts/post/*").permitAll()
                        .requestMatchers(HttpMethod.GET,
                "/api/posts/all").permitAll()
                        .requestMatchers(HttpMethod.GET,
                "/api/posts/all-paged*").permitAll()
                ).httpBasic(withDefaults()).build();
    }
    @Bean
    @Primary
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder
             (BCryptPasswordEncoder.BCryptVersion.$2A, 10);
    }
}
```

Příklad 25: Spring Security konfigurace s HTTP Basic Security, připojením na databázi s uživateli a rolemi, a BCrypt hashováním hesel, zjednodušeno *(Zdroj: vlastní zpracování)*.

Je také nutné vědět v aplikaci stav uživatele, který k dané HTTP metodě v danou chvíli přistupuje. Toto bude implementováno v UserService třídě.

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDTOMapper userDTOMapper;

    @Autowired
    private UserDAO userDAO;

    @Autowired
    private AuthorityDAO authorityDAO;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public InformationUserDTO getCurrentUser() {
        try {
            return
            userDTOMapper.userToUserDTO(
              userDAO.findOne(getCurrentUsername())
            );
        }
        catch (Exception e) {
            e.printStackTrace();
            throw new IllegalStateException("Failed to find current
            user information");
        }
    }

    @Override
    public String getCurrentUsername() {
        Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
        return authentication.getName();
    }

    @Override
    public List<String> getCurrentRoles() {
        Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
        return
         authentication.getAuthorities().stream()
         .map(GrantedAuthority::getAuthority).toList();
    }
}
```

Příklad 26: Získání aktuálního uživatele v Spring Security z SecurityContext *(Zdroj: vlastní zpracování)*.

Údaje z těchto metod následně používáme ve všech Service třídách. Např. při vytvoření příspěvku nebo komentáře pro nastavení jeho autora. Dále třeba pro zjištění role uživatele (zda je administrátor) při určitých operacích.

7.6 Konfigurace

Co se týče konfigurace, ta se provádí v application.properties.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.url=jdbc:mysql://localhost:3306/boards-app

spring.jpa.hibernate.ddl-auto=validate
spring.jpa.properties.hibernate.jpa.compliance=false

spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=false
```

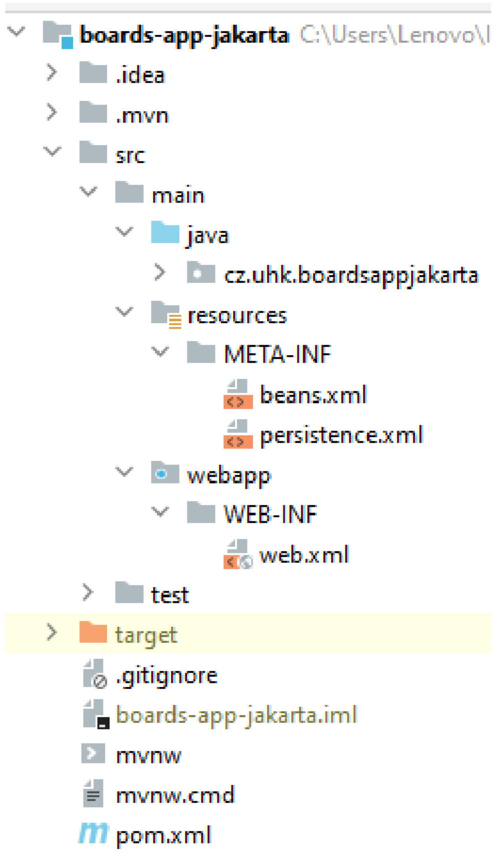
Příklad 27: Spring Data JPA - konfigurace připojení k databázi

Konfigurují se údaje pro připojení k databázi, JDBC driver, hibernate validace (zda jsou při startu v databázi tabulky s korektní strukturou jako entity v aplikaci). Dále možnost zobrazení SQL (k ověření, že jsou hlavně select query korektní, aby nedošlo k problémům s výkonem). Co se týče JPA Complience u Hibernate, ta je nastavena na false (hibernate pak použije specifické optimalizace, které ale nejsou definovány v JPA specifikaci). V rámci Spring DATA JPA je ze základu false, ale v Jakarta EE je true. Nastavení tedy bylo potřeba sjednotit pro budoucí otestování.

Do složky webapps/WEB-INF bylo potřeba ještě přidat konfigurační soubor jboss-deployment-structure.xml, ve kterém se zakáže použití logging modulu WildFly (aplikace bez této změny ve WildFly nenastartuje).

8 Vývoj Jakarta EE aplikace

V rámci této kapitoly budou popsány rozdíly a postup konverze Spring Boot aplikace na Jakarta EE aplikaci.



Obrázek 7: Struktura Jakarta EE projektu, vytvořeno z template Web Profile v IntelliJ Idea *(Zdroj: vlastní zpracování)*.

Co se týče DAO a Service tříd, tak ty jsou totožné jako v Spring aplikaci. Liší se akorát anotacemi, kdy dependency Injection provádíme pomocí anotace @Inject a bean vytvoříme pomocí anotace @ApplicationScoped. Je také použita jiná @Transactional anotace (jméno je stejné, ale import je místo z Spring Transaction použit z JTA). Liší se hlavně implementace Security contextu a filteru podle práv uživatelů. Jakarta EE Aplikace také neobsahuje přímo runtime dependencies, které jsou vlastně už přímo v aplikačním serveru. Aplikace je tedy programována jako API.

Nastavení databáze probíhá v souboru persistence.xml. Datasource je ale definován (adresa pro připojení k DB, uživatel, driver) na aplikačním serveru. Na aplikační server WildFly tedy bylo nutné doinstalovat JDBC driver pro MySQL databázi a nakonfigurovat zde připojení k databázi (docíleno pomocí [40]). Databáze je konfigurována v aplikačním serveru z důvodu, aby bylo možné použít JTA transakce, které jsou spravovány

implementací v aplikačním serveru (aby bylo možné použít JTA @Transactional anotaci v Jakarta EE podobným způsobem, jako je @Transactional Spring Frameworku).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
          https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
version="3.0">
    <persistence-unit name="boards-persistence-unit" transaction
    type="JTA">
        <jta-data-source>java:jboss/datasources/BoardsDS</jta-data
        source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="hibernate.hbm2ddl.auto"
            value="validate"/>
            <property name="hibernate.jpa.compliance"
            value="false"/>
            <property name="hibernate.show_sql" value="false"/>
            <property name="hibernate.format_sql" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Příklad 28: persistence.xml konfigurace *(Zdroj: vlastní zpracování)*.

Persistence.xml konfigurace je jednoduchá. Důležitý je element <exclude-unlisted-classes> s false hodnotou, díky kterému se v rámci JPA použijí všechny třídy s @Entity anotací pro namapování s databází (aby se nemusely ty třídy zmiňovat přímo v persistence.xml). Také je nastaven transaction-type na JTA a datasource definován jako jta-data-source, aby bylo možné řídit transakce pomocí JTA anotací.

Jako knihovny jsou použity stejně jako ve Spring verzi aplikace Lombok, Mapstruct. Navíc je ještě použita BCrypt Java Library pro zprovoznění BCrypt hashování a verifikace v Jakarta EE Security. V pom.xml v build konfigurace je pro Mapstruct nastaven jako Component model nastaven cdi místo spring. Díky tomu budou mít vygenerované mapper classy anotaci @ApplicationScoped, takže se budou moci injectnout pomocí @Inject.

Konfigurace souboru beans.xml vypadá takto. Jde o nastavení, aby byly vytvořeny beany z tříd, kde je anotace (v tomto případě používáme pouze @ApplicationScoped).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
       bean-discovery-mode="annotated">

</beans>
```

Příklad 29: beans.xml defaultní konfigurace.

HTTP endpointy jsou řešeny v Jakarta EE anotacemi. Anotace ApplicationPath nad třídou implementující interface Application definuje počáteční cestu pro dané Controllery a také konfiguraci pro JSON serializaci/deserializaci (ObjectMapperContextResolver). Nad samotnými controllery je pak anotace Path určující další část cesty. V controlleru přímo nad danými metodami (které určují HTTP endpointy) je také anotací Path určena finální část cesty. V této ukázce je cesta /api/posts/new.

```
@ApplicationPath("/api/")
public class ApiApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return Set.of(UserController.class, PostController.class,
         CommentController.class, ObjectMapperContextResolver.class
        );
    }
}

@Path("/posts/")
public class PostController {
    @Inject
    private PostService postService;

    @POST
    @Produces("application/json")
    @Consumes("application/json")
    @Path("/new")
    public Response addNewPost(NewPostDTO newPostDTO) {
        try {
            Long id = postService.addNewPost(newPostDTO);
            return Response.ok().entity(
                new SuccessDTO("New post with id " + id +
                "created")).build();
        }
        catch (IllegalStateException e) {
            return
            Response.status(Response.Status.BAD_REQUEST).entity(new
            ErrorDTO(e.getMessage())).build();
        }
    }

    @GET
    @Produces("application/json")
    @Path("/all")
    public Response findVisiblePosts() {
        try {
           return
            Response.ok().entity(postService.findVisiblePosts())
            .build();
        }
        catch (IllegalStateException e) {
           return
            Response.status(Response.Status.INTERNAL_SERVER_ERROR)
            .entity(new ErrorDTO(e.getMessage())).build();
        }
    }
}
```

Příklad 30: Jakarta EE HTTP endpoint, zjednodušeno (stejný příklad, jako ve Spring verzi). *(Zdroj: vlastní zpracování)*.

Security je pak nastavena ve třídě SecurityConfiguration. Stejně jako ve Spring verzi jde o HTTP Basic zabezpečení. Bylo také nutné implementovat PasswordHash interface s BCrypt implementací. Implementace byla vytvořena pomocí BCrypt Java Library (třída BCrypt).

```
@BasicAuthenticationMechanismDefinition
@DatabaseIdentityStoreDefinition(
        dataSourceLookup = "java:jboss/datasources/BoardsDS",
        callerQuery = "select password from users where username = ?
and enabled=true",
        groupsQuery = "select authority from authorities where
username = ?",
        hashAlgorithm = BCryptPasswordHash.class)
@DeclareRoles({"ROLE_ADMIN", "ROLE_USER"})
@ApplicationScoped
public class SecurityConfiguration {

}


@ApplicationScoped
public class BCryptPasswordHash implements PasswordHash {

    @Override
    public String generate(char[] chars) {
        return
BCrypt.with(BCrypt.Version.VERSION_2A).hashToString(10, chars);
    }

    @Override
    public boolean verify(char[] chars, String s) {
        return BCrypt.verifyer().verify(chars, s).verified;
    }
}
```

Příklad 31: Security konfigurace v Jakarta EE *(Zdroj: vlastní zpracování)*.

Nyní je nutné přímo nastavit přístupy k daným URL v web.xml. Lze buď definovat role, které budou mít přístup, nebo * (pak má přístup uživatel s alespoň jednou jakoukoliv rolí), anebo ** (pak má přístup přihlášený uživatel a role se neřeší).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app …>
    <display-name>Board App Jakarta</display-name>

    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
    <security-role>
        <role-name>ROLE_USER</role-name>
    </security-role>
    <security-role>
        <role-name>ROLE_ADMIN</role-name>
    </security-role>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>posts remove</web-resource-name>
            <url-pattern>/api/posts/remove/*</url-pattern>
        </web-resource-collection>


        <auth-constraint>
            <role-name>ROLE_ADMIN</role-name>
        </auth-constraint>
    </security-constraint>
</web-app>
```

Příklad 32: Security konfigurace, zjednodušeno (reálná konfigurace je stejná, jako v případě Spring verze aplikace). *(Zdroj: vlastní zpracování)*.

V rámci konfigurace Jakarta Security se ve Wildfly serveru muselo vypnout Integrated JASPI u základní Security domain se jménem other. Jednalo se o specifickou věc pro Wildfly [41].

Aby byl stejný JSON výstup z REST API, jako je ve Spring aplikaci, bylo nutné nakonfigurovat Jackson ObjectMapper (ze základu nebyl Jackson použit). Toho bylo docíleno pomocí níže uvedené třídy, pomocí nastavení context-param resteasy.preferJacksonOverJsonB na true v web.xml a také přiřazení do třídy anotované @ApplicationPath.

```
@Provider
@Produces("application/json")
@Consumes("application/json")
public class ObjectMapperContextResolver implements
ContextResolver<ObjectMapper> {

private final ObjectMapper objectMapper;

 public ObjectMapperContextResolver() {
 objectMapper = JsonMapper.builder().findAndAddModules()
  .configure(MapperFeature.DEFAULT_VIEW_INCLUSION, false)
  .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false)
  .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
  .configure(SerializationFeature.WRITE_DURATIONS_AS_TIMESTAMPS,
false)
  .build();
 }

 @Override
  public ObjectMapper getContext(Class<?> aClass) {
 return objectMapper;
 }
}
```

Příklad 33: Konfigurace Jackson stejně, jako je defaultní konfigurace ve Spring[42].
(Zdroj: vlastní zpracování).

9 Testovací data a testování HTTP metod

Pro generování dat bude použit node.js script využívající tyto dependencies: @faker-js/faker, mysql, bcrypt. Díky faker-js nebudou v databázi totožná data, ale náhodná. MySQL server musí mít nastaven čas na UTC, protože v případě CET/CEST dojde k chybě při vygenerování času, který bude v době, kdy dochází ke změně času mezi CET a CEST.

```
for (let i = 0; i < 1_000; i++) {
    const author = `user${Math.floor(Math.random() * 4) + 1}`;
    const title = faker.word.words(3);
    const content = faker.word.words(15);
    //const removed = (Math.random() < 0.05);
    const removed = false;

    //MySQL needs to be set to UTC, otherwise error in 'banned
hours' during change from cet to cest (or otherwise)
    const time = faker.date.between(
      {
        from: new Date(2023, 1, 1),
        to: new Date(2023,12,31)
      }
    )

    const updateTime = ((Math.random() < 0.05) || removed) ?
faker.date.soon({days: 10, refDate: time}) : time;

    connection.query(
        'INSERT INTO Posts (author, title, content, removed,
created_at, updated_at) VALUES (?, ?, ?, ?, ?, ?)',
        [author, title, content, removed, time, updateTime]
    );
}
```

Příklad 34: Ukázka cyklu tvořícího fake testovací data do databáze. *(Zdroj: vlastní zpracování)*.

Následně je udělán pomocí Postman scénář pro použití REST API. Pomocí něho lze otestovat postupně dané HTTP metody a vygenerovat dokumentaci. Dokumentaci je možné zveřejnit na webu Postman. V případě potřeby exportu dokumentace pro offline použití lze vygenerovat pomocí nástroje postmanerator (nebo pomocí dalších nástrojů).

http

```
POST /api/user/register HTTP/1.1
Host: localhost:8080
Content-Length: 64

{
    "username": "newUsername",
    "password": "password"
}
```

Example Response

Body Headers (14)

json

```
{
  "text": "Created user with username: newUsername!"
}
```

Obrázek 8: Dokumentace v Postman – registrace uživatele

Testovací scénář pro otestování GET requestů je tvořen tak, že se nejprve naplní pomocí generátoru testovacích dat 1000 příspěvků a k nim náhodně 5000 komentářů (vygenerování do databáze ale proběhne pouze jednou a DB bude použita v obou variantách aplikace, aby byl stav DB při obou testech stejný). První test bude mít 100 iterací, které bude odbavovat paralelně 10 uživatelů. Jedna iterace testu proběhne tak, že se nejdříve načtou pomocí stránkování postupně po 20 příspěvky (takže ve výsledku 50 stránek - requestů). Následně se ke každému příspěvku načtou jeho komentáře (1000 requestů). Jedna iterace tedy má 1050 GET requestů a iterací je 100. V jednom testu tedy proběhne 105 000 GET requestů celkem.

Následně je vytvořen i scénář pro otestování POST requestů. Bude probíhat vždy při promazaných tabulkách v DB. První test má 1000 iterací, které bude stejně jako u GET requestů odbavovat paralelně 10 uživatelů. V rámci jedné iterace testu dojde k přidání 1 příspěvku (1 POST request), k němuž bude přidáno 5 komentářů (5 POST requestů). V jednom testu tedy proběhne celkem 6000 POST requestů (6 * 1000).

Scénář pro otestování výkonu je vytvořen v Grafana K6 load testing frameworku. Je to z důvodu jednoduchosti a také výrazně lepšího výkonu při použití K6 např. oproti Postman. K6 se nainstaluje jednoduše pomocí Node.js.

```
export default function() {
  for(let i=1; i<=50; i++) {
    http.get("http://localhost:8080/api/posts/all-paged?page=" + i +
      "&size=20");
  }
  for(let i=1; i<=1000; i++) {
    http.get("http://localhost:8080/api/comments/all/" + i);
  }
}

export default function() {
    const id = `${exec.scenario.iterationInTest + 1}`;
    const postData = {
        title: `Title of Post ${id}`,
        content: `Content of Post ${id}`
    };
    http.post(`http://${credentials}@localhost:8080/api/posts/new`,
    JSON.stringify(postData), params);

    for(let i=1; i<=5; i++) {
        const commentData = {
            content: `Content of Reply ${i} to Post ${id}`
        };
        http.post(`http://${credentials}@localhost:8080/api/comments/ne
          w/${id}`, JSON.stringify(commentData), params);
    }
}
```

Příklad 35: Hlavní funkce JS scriptu (první je pro test GET, druhá POST požadavků) pro load testing pomocí K6, zjednodušeno a zkombinováno. *(Zdroj: vlastní zpracování)*.

Skript pro otestování GET požadavků lze následně spustit pomocí příkazu *k6 run –env scenario=per_vu_scenario K6_Perf_Test.js*, nebo *k6 run –env scenario=shared_iter_scenario K6_Perf_Test.js*. V případě shared_iter scénáře bude 10 klientů přistupovat k dalšímu požadavku ihned, když je předchozí odbaven. V případě per_vu scénáře každý klient zpracuje svoji 1/10 požadavků. K6 scénáře jsou vytvořeny podle [43].

V případě testu, který testuje POST requesty dojde stejným způsobem ke spuštění souboru *K6_Perf_Test_POST.js* (také s výběrem per_vu, nebo shared_iter scénáře).

V rámci WildFly serveru bylo ještě nutné nastavit Connection Pool na podobné parametry, které jsou ze základu nastavené v rámci HikariCP[44]. Sice tím nedojde k úplnému vyrovnání těchto DB Connection Pools (Hikari funguje na jiném principu pomocí ConcurrentBag), ale alespoň se srovnají jejich parametry.

Initial Pool Size	10
Min Pool Size	10
Max Pool Size	10
Flush Strategy	*FailingConnectionOnly*
Pool Fair	true
Pool Prefill	true
Pool Use Strict Min	*false*
Use Fast Fail	false

Obrázek 9: Nastavení Connection Poolu pro Datasource ve Wildfly

Při testování Spring Boot aplikace ve WildFly pak bude manuálně tento connection pool vypnut, aby nebyl připojen k DB. Důvodem je, aby tím nedošlo k omezení výkonu Spring Boot aplikace. Spring Boot aplikace totiž používá vlastní HikariCP connection pool (a Wildfly connection pool by tak byl připojen k DB navíc).

10 Shrnutí a výsledky porovnání

Co se týče části vývoje aplikací, tak snadněji se vyvíjela Spring verze aplikace. V Jakarta EE ale nebyl vývoj o tolik složitější oproti původnímu předpokladu. Hlavní výhodou u Springu bylo to, že je dostupných výrazně více materiálů při řešení problémů. U Jakarta EE byly některé problémy s Wildfly, které byly specifické pro něj (např. konfigurace Security), k čemuž by dle mě nemělo docházet (specifikaci by měl každý splňovat stejně).

Testování probíhalo tak, že v případě testů GET požadavků se DB naplnila jednou fake testovacími daty, se kterými pak proběhly všechny tyto testy. V případě testů POST požadavků pak byla DB před každým testem vyresetována do původního stavu (smazání a znovuvytvoření tabulek). Před každým testem došlo také k restartu DB serveru.

Výsledky dopadly u testů GET požadavků lépe pro Jakarta EE aplikaci. Hlavním důvodem tohoto bude větší počet HTTP headerů v odpovědích, které posílá Spring Boot aplikace. U testů GET požadavků tak Spring Boot aplikace odeslala 174 MB dat, zatímco Jakarta EE 144 MB dat. Rozdíl v hlavičkách je uveden v obrázcích níže. Co se týče POST requestů, ty naopak časově lépe dopadly v Spring Boot aplikaci. Níže je uvedena tabulka s výsledky.

Typ požadavku	**Scénář**	**Aplikace**	**Celkový čas trvání**	**Průměrný čas trvání jednoho požadavku**
GET	shared_iter_scenario	Spring Boot	37,3s	3,13ms
GET	shared_iter_scenario	Jakarta EE	34,2s	3,07ms
GET	per_vu_scenario	Spring Boot	36,8s	3,35ms
GET	per_vu_scenario	Jakarta EE	34,4s	3,1ms
POST	shared_iter_scenario	Spring Boot	47,1s	78,28ms
POST	shared_iter_scenario	Jakarta EE	52,3s	86,34ms
POST	per_vu_scenario	Spring Boot	48,7s	80,68ms
POST	per_vu_scenario	Jakarta EE	52,4s	86,52ms

Tabulka 2: Výsledky měření

Key	Value
Expires	0
Cache-Control	no-cache, no-store, max-age=0, must-revalidate
X-XSS-Protection	0
Pragma	no-cache
X-Frame-Options	DENY
Date	Tue, 19 Mar 2024 17:58:34 GMT
Connection	keep-alive
Vary	Origin
Vary	Access-Control-Request-Method
Vary	Access-Control-Request-Headers
X-Content-Type-Options	nosniff
Transfer-Encoding	chunked
Content-Type	application/json

Obrázek 10: Počet HTTP hlaviček v response (Spring Boot)

Key	Value
Expires	0
Connection	keep-alive
Cache-Control	no-cache, no-store, must-revalidate
Pragma	no-cache
Content-Type	application/json
Content-Length	63
Date	Tue, 19 Mar 2024 18:07:52 GMT

Obrázek 11: Počet HTTP hlaviček v response (Jakarta EE)

V porovnání metrik kódu si obě aplikace vedly velmi podobně. Byl ale velký rozdíl ve velikosti obou aplikací. Důvodem je to, že při vývoji Jakarta EE byl psán kód pro Jakarta EE API, a proto knihovny implementace tohoto API byly poskytnuty Jakarta EE aplikačním serverem (Wildfly). V případě Spring aplikace jsou knihovny přímo ve WAR souboru aplikace, aby mohl být spouštěn i na servletových serverech typu Tomcat.

Aplikace	Počet Java tříd	Počet řádků zdrojového kódu	Velikost aplikace (WAR souboru)
Jakarta EE	45	1162	194 KiB
Spring Boot	46	1147	39 MiB

Tabulka 3: Porovnání dalších metrik (počet řádků kódu počítán u Java tříd bez mezer)

11 Závěr

V úvodní části práce byl představen programovací jazyk Java a základní teorie ohledně klient-server architektury, k jejímuž využívání dochází právě při vývoji webových aplikací. Hlavní částí bylo seznámení s hlavními rysy Java EE (nyní Jakarta EE), Spring Frameworku a specifikace jejich konceptů a principů. Tato bakalářská práce popsala části těchto technologií, zasazených v dané architektuře, a ve kterých případech nebo proč se používají, a to i v několika ukázkách. V rámci vývoje webových aplikací v Javě se používají i jiné frameworky, což bakalářská práce také stručně v teoretické části obsáhla.

V rámci části praktické se probral vývoj Spring aplikace a její transformace na Jakarta EE aplikaci. Následně došlo k vygenerování testovacích dat a k otestování výkonu obou aplikací. Při tvorbě obou aplikací byla snaha o jejich stejné nakonfigurování, aby nedošlo k zvýhodnění jedné z nich. Konkrétně šlo o použití Hibernate v obou případech, dále i Jackson a třeba nastavení Connection Pool ve WildFly podle HikariCP. Obě aplikace také byly spuštěny na stejném aplikačním serveru WildFly.

Díky úspěšnému vytvoření obou aplikací a jejich otestování se podařilo splnit stanovený cíl této bakalářské práce. Co se týče Spring Boot aplikace, původně nebylo zamýšleno ji spouštět na Wildfly, ale na Tomcatu. Nakonec se podařilo vyřešit problém s logováním, kvůli kterému Spring aplikace na Wildfly padala, a díky tomu byly aplikace při testování více vyrovnané.

V testu GET požadavků dopadla lépe Spring Boot aplikace, ale v testu POST požadavků naopak Jakarta EE aplikace. Rozdíl ve výkonu v obou případech ale není až tak zásadní, aby rozhodl o tom, který framework použít. Hlavním důvodem k výběru frameworku by měly být technologie, které daný framework poskytuje a v aplikaci je využijeme. Jak Jakarta EE, tak i Spring Framework poskytují mnoho různých technologií a záleží tedy na tom, jaký typ aplikace vyvíjíme. V případě, že programátor má již zkušenosti z vývoje s jedním z těchto frameworků, může to být také důvodem, proč daný framework vybrat a ušetřit tak náklady.

12 Seznam použité literatury

[1] BAELDUNG. *A Brief History of the Java Programming Language | Baeldung* [online]. 2. leden 2022 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/java-history

[2] *The 7 benefits of Java* [online]. [vid. 2023-10-18]. Dostupné z: https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/7-Benefits-Java-advantages-dynamic-robust-performance-security-objects-simple

[3] *The Java Language Environment* [online]. [vid. 2023-10-18]. Dostupné z: https://www.oracle.com/java/technologies/introduction-to-java.html#334

[4] *What is Client-Server Architecture? Everything You Should Know | Simplilearn* [online]. [vid. 2023-10-18]. Dostupné z: https://www.simplilearn.com/what-is-client-server-architecture-article

[5] *HTTP protokol* [online]. [vid. 2023-10-18]. Dostupné z: https://www.jakpsatweb.cz/server/http-protokol.html

[6] *Maven – Introduction* [online]. [vid. 2023-10-29]. Dostupné z: https://maven.apache.org/what-is-maven.html

[7] *Maven – Maven in 5 Minutes* [online]. [vid. 2023-10-29]. Dostupné z: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

[8] *Packages* [online]. [vid. 2023-10-29]. Dostupné z: https://docs.oracle.com/javase/specs/jls/se6/html/packages.html#7.7

[9] *Differences between Java EE and Java SE - Your First Cup: An Introduction to the Java EE Platform* [online]. [vid. 2023-10-18]. Dostupné z: https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html

[10] GRACIANO, Rodrigo. *Java EE vs J2EE vs Jakarta EE | Baeldung* [online]. 28. prosinec 2018 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/java-enterprise-evolution

[11] jakarta.ee/LICENSE at src · jakartaee/jakarta.ee. *GitHub* [online]. [vid. 2023-10-18]. Dostupné z: https://github.com/jakartaee/jakarta.ee/blob/src/LICENSE

[12] BEATON, Wayne. Jakarta EE Platform. *projects.eclipse.org* [online]. 27. červen 2018 [vid. 2023-10-18]. Dostupné z: https://projects.eclipse.org/projects/ee4j.jakartaee-platform

[13] TIJMS, Arjan. *Transition from Java EE to Jakarta EE* [online]. [vid. 2023-10-27]. Dostupné z: https://blogs.oracle.com/javamagazine/post/transition-from-java-ee-to-jakarta-ee

[14] FOUNDATION, Eclipse. Jakarta EE Compatible Products | Enterprise Java Application and Web Servers | The Eclipse Foundation. *Jakarta® EE: The New Home of Cloud Native Java* [online]. [vid. 2023-10-29]. Dostupné z: https://jakarta.ee/compatibility/certification/10/

[15] *The Jakarta® EE Tutorial* [online]. [vid. 2023-10-18]. Dostupné z: https://eclipse-ee4j.github.io/jakartaee-tutorial/

[16] *Chapter 2 Java Enterprise System Architecture* [online]. [vid. 2023-10-18]. Dostupné z: https://docs.oracle.com/cd/E19263-01/817-5764/architecture.html

[17] *Java EE vs Spring: Frameworks Comparison | EPAM Startups & SMBs* [online]. [vid. 2023-10-18]. Dostupné z: https://anywhere.epam.com/business/spring-vs-java-ee

[18] BAELDUNG. *How to Register a Servlet in Java | Baeldung* [online]. 3. březen 2017 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/register-servlet

[19] HARTINGER, David Čápka. *Lekce 1 - Úvod do Java Enterprise Edition (JEE)* [online]. [vid. 2023-10-18]. Dostupné z: https://www.itnetwork.cz/java-enterprise-edition-uvod-do-jee-j2ee

[20] *Java EE APIs* [online]. [vid. 2023-10-18]. Dostupné z: https://javaee.github.io/tutorial/overview008.html

[21] *Contexts and Dependency Injection in Java EE 6* [online]. [vid. 2023-10-18]. Dostupné z: https://www.oracle.com/technical-resources/articles/java/cdi-javaee-bien.html

[22] Getting Started | Accessing Data with JPA. *Getting Started | Accessing Data with JPA* [online]. [vid. 2023-11-07]. Dostupné z: https://spring.io/guides/gs/accessing-data-jpa/

[23] Spring and Spring Boot Frameworks: A Brief History - DZone. *dzone.com* [online]. [vid. 2023-10-18]. Dostupné z: https://dzone.com/articles/history-of-spring-framework-spring-boot-framework

[24] spring-framework/LICENSE.txt at main · spring-projects/spring-framework. *GitHub* [online]. [vid. 2023-10-18]. Dostupné z: https://github.com/spring-projects/spring-framework/blob/main/LICENSE.txt

[25] VMware to Acquire SpringSource. *VMware News and Stories* [online]. 10. srpen 2009 [vid. 2023-10-18]. Dostupné z: https://news.vmware.com/releases/springsource

[26] Spring Framework. *endoflife.date* [online]. 13. říjen 2023 [vid. 2023-10-18]. Dostupné z: https://endoflife.date/spring-framework

[27] CRUSOVEANU, Loredana. *Inversion of Control and Dependency Injection with Spring | Baeldung* [online]. 28. prosinec 2016 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

[28] BAELDUNG. *Introduction to Spring AOP | Baeldung* [online]. 5. listopad 2017 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/spring-aop

[29] *1. Introduction to Spring Framework* [online]. [vid. 2023-10-18]. Dostupné z: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html

[30] BAELDUNG. *Spring Expression Language Guide | Baeldung* [online]. 18. duben 2016 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/spring-expression-language

[31] *Quick Guide to Spring Controllers | Baeldung* [online]. 2. srpen 2016 [vid. 2023-10-18]. Dostupné z: https://www.baeldung.com/spring-controllers

[32] Spring Boot - Starters. *GeeksforGeeks* [online]. 10. červenec 2021 [vid. 2023-11-07]. Dostupné z: https://www.geeksforgeeks.org/spring-boot-starters/

[33] Spring Boot. *Spring Boot* [online]. [vid. 2023-11-07]. Dostupné z: https://spring.io/projects/spring-boot

[34] JVM Ecosystem Report 2021. *Snyk* [online]. [vid. 2023-10-18]. Dostupné z: https://snyk.io/reports/jvm-ecosystem-report-2021/

[35] Welcome | Ktor. *Ktor Help* [online]. [vid. 2023-10-18]. Dostupné z: https://ktor.io/docs/2.3.5/welcome.html

[36] Spring Boot, Quarkus, or Micronaut? - DZone. *dzone.com* [online]. [vid. 2023-10-18]. Dostupné z: https://dzone.com/articles/spring-boot-quarkus-or-micronaut

[37] PARASCHIV, Eugen. *The DAO with JPA and Spring | Baeldung* [online]. 15. srpen 2013 [vid. 2023-12-31]. Dostupné z: https://www.baeldung.com/spring-dao-jpa

[38] PARASCHIV, Eugen. *JPA Pagination | Baeldung* [online]. 7. duben 2014 [vid. 2023-12-31]. Dostupné z: https://www.baeldung.com/jpa-pagination

[39] JT. *Using MapStruct with Project Lombok - Spring Framework Guru* [online]. 5. únor 2021 [vid. 2024-01-02]. Dostupné z: https://springframework.guru/using-mapstruct-with-project-lombok/, https://springframework.guru/using-mapstruct-with-project-lombok/

[40] F.MARCHIONI. How to install a JDBC Driver on WildFly. *Mastertheboss* [online]. 20. září 2021 [vid. 2024-04-20]. Dostupné z: https://www.mastertheboss.com/jbossas/jboss-datasource/how-to-install-a-jdbc-driver-on-wildfly/

[41] ZOLNOWSKI, Marcos. Answer to „Why does this simple Jakarta Security example from Soteria work on Payara but not on WildFly?" In: *Stack Overflow* [online]. 6. prosinec 2021 [vid. 2024-01-21]. Dostupné z: https://stackoverflow.com/a/70240973

[42] *“How-to” Guides* [online]. [vid. 2024-01-21]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto.spring-mvc.customize-jackson-objectmapper

[43] How to execute single scenario out of multiple scenarios in a script? - Grafana k6 / OSS Support. *Grafana Labs Community Forums* [online]. 2. duben 2021 [vid. 2024-04-20]. Dostupné z: https://community.grafana.com/t/how-to-execute-single-scenario-out-of-multiple-scenarios-in-a-script/99301/2

[44] WOOLDRIDGE, Brett. *brettwooldridge/HikariCP* [online]. Java. 21. leden 2024 [vid. 2024-01-21]. Dostupné z: https://github.com/brettwooldridge/HikariCP

13 Přílohy

1. Zdrojový kód Spring Boot aplikace – https://github.com/MrRazor/boards-app-spring/
2. Zdrojový kód Jakarta EE aplikace – https://github.com/MrRazor/boards-app-jakarta/
3. Ostatní – https://github.com/MrRazor/insert_generator/
    a. index.js – Node.js script pro vytvoření testovacích dat
    b. složka API – Postman JSON pro import do Postman, vygenerovaná HTML dokumentace REST API
    c. složka testing – K6.io scripty pro výkonové testy
    d. složka results – podrobné výsledky výkonových testů z předešle zmíněných scriptů
    e. složka installation – návod k zprovoznění WildFly aplikačního serveru (a jak v něm rozeběhnout obě aplikace), související soubory s instalací

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Jan Tomáš Štekl**
Studium:	I2100286
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Porovnání Java EE a Spring Frameworku**
Název bakalářské práce AJ:	Java EE and Spring Framework Comparison

Cíl, metody, literatura, předpoklady:

Cíl: Cílem práce je porovnat Spring Framework a Java EE, jejich architekturu, části a užívání podle určené metodologie. Frameworky budou také porovnány na vzorové aplikaci, kde budou vidět praktické rozdíly mezi nimi.

Osnova:

1. Úvod
2. Metodologie
3. Popis Java EE
4. Popis Spring Frameworku
5. Vývoj vzorové aplikace
6. Shrnutí a výsledky porovnání
7. Závěr

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 1.10.2023