F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Generátor dokumentace ze Spring Boot konfiguračních tříd

Lukáš Kaňka
Softwarové inženýrství a technologie
lukaskabc@gmail.com

Květen 2024
Vedoucí práce: Ing. Martin Ledvinka, Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kaňka** Jméno: **Lukáš** Osobní číslo: **507216**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

Specializace: **Enterprise systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Generátor dokumentace ze Spring Boot konfiguračních tříd

Název bakalářské práce anglicky:

Generator of Documentation for Spring Boot Configuration Classes

Pokyny pro vypracování:

1. Seznamte se s technologií Spring Boot, její konfigurací a nasazení Spring Boot aplikací (včetně jejich nasazení v rámci Docker kontejnerů).
2. Analyzujte způsoby, jakými by bylo možné vygenerovat dokumentaci ze Spring Boot konfiguračních tříd. Zvláštní pozornost věnujte generování této dokumentaci do tabulek v Markdown.
3. Navrhněte nástroj, který bude možné použít pro generování dokumentace konfiguračních tříd Spring Boot aplikací na základě Javadocu. Nástroj musí být snadné použít v rámci projektů, měl by tedy mít podobu např. Maven pluginu.
4. Navržený nástroj implementujte.
5. Ověřte implementaci porovnáním vygenerované dokumentace konfigurace systému TermIt s ručně psanou variantou dostupnou na portálu GitHub.

Seznam doporučené literatury:

1. https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config.typesafe-configuration-properties.using-annotated-type
2. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html
3. https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-synta

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, Ph.D. skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **25.01.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Martin Ledvinka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Poděkování / Prohlášení

Chtěl bych poděkovat Ing. Martinu Ledvinkovi, Ph.D., za pomoc a cenné rady při vedení bakalářské práce. Děkuji také své rodině a přátelům za jejich podporu.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

.....

Abstrakt / Abstract

Tato práce se zabývá hledáním možných řešení pro automatické generování dokumentace z konfiguračních tříd Java Spring Boot projektů. Nalezená řešení jsou porovnána a následně je realizována implementace jednoho z nich.

Cílem je vytvoření nástroje, který bude možné snadno zapojit do stávajícího Java projektu. Na úvod jsou představeny technologie související s problematikou Spring Boot projektů, jejich konfigurací a dokumentací. Dále jsou popsány možné přístupy k řešení a jejich porovnání. Při výběru vhodného řešení je kladen důraz na jednoduchost návrhu, implementaci a případnou rozšiřitelnost výsledného nástroje. Následně se již práce věnuje návrhu a výsledné implementaci.

Výsledkem je nástroj, který je možné snadno spustit ve stávajícím Java Spring Boot projektu. Nástroj provede analýzu zdrojového kódu a z dostupné programátorské dokumentace, která se nachází ve zdrojovém kódu v podobě Javadoc komentářů, vygeneruje dokumentaci konfigurovatelných hodnot aplikace.

Klíčová slova: SpringBoot, Konfigurace, Markdown, Javadoc, Annotation Processor, Maven, Plugin, Dokumentace

This thesis examines possible solutions for the automatic generation of documentation from configuration classes of Java Spring Boot projects. The solutions found are compared, and then an implementation of one of them is realized.

The goal is to create a tool that can be easily integrated into an existing Java project. Initially, technologies related to the issues of Spring Boot projects, as well as their configuration and documentation, are presented. Next, possible approaches to the solution are described and compared. When selecting a suitable solution, the focus is on simplicity of design, implementation, and possible extensibility of the resulting tool. Subsequently, the work focuses on the design and the resulting implementation.

The result is a tool that can be easily run in an existing Java Spring Boot project. It performs an analysis of the source code and generates documentation of configurable values from the available Javadoc comments.

Keywords: SpringBoot, Configuration, Markdown, Javadoc, Annotation Processor, Maven, Plugin, Documentation

Title translation: Generator of documentation for Spring Boot configuration classes

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Při nasazování aplikace je obvykle třeba ji patřičně nakonfigurovat a přizpůsobit provoznímu prostředí. Začlenění konkrétních konfigurovatelných hodnot do zdrojového kódu je nevhodné, pro jejich úpravy by bylo nutné opětovné sestavení celé aplikace. Konfigurace je tedy často umístěna do externích zdrojů (například souborů), které jsou snadno upravitelné a ze kterých jsou pak hodnoty načítány dynamicky při spuštění aplikace [1].

Konfigurace tvoří rozhraní mezi kódem a uživatelem[1]. Běžně musí vývojář explicitně vytvářet uživatelskou dokumentaci, která popisuje dostupné možnosti konfigurace. Existence takové dokumentace, která je oddělena od zdrojového kódu, představuje nové povinnosti, na které musí vývojář pamatovat. Dokumentace musí být udržována aktuální a konzistentní se zdrojovým kódem, což může být časově náročné a přinášet nový prostor pro vznik chyb. Vývojář má samozřejmě možnost dokumentaci umístit přímo do zdrojového kódu a uživateli zpřístupnit programátorskou dokumentaci, která je ovšem často obsáhlá, složitá a celkově uživatelsky nepřívětivá. Uživatel k ní tedy standardně přístup nepotřebuje.

Spring Boot je knihovna pro tvorbu aplikací na platformě Java, mimo jiné také nabízí automatické mapování konfiguračních souborů, proměnných prostředí (environmental variables), parametrů z příkazové řádky a dalších zdrojů na konkrétní proměnné a třídy. Během procesu načítání konfigurace také provádí typovou kontrolu dat a umožňuje vygenerování metadat pro vývojová prostředí za účelem napovídání při úpravách konfiguračních souborů. Těmito funkcemi tak usnadňuje externalizaci konfigurace, její načítání a zpracovávání.

Tato práce se zabývá analýzou dostupných možností pro čtení a následné zpracování zdrojového kódu Java aplikací, konkrétně tedy Spring Boot konfiguračních tříd. Cílem je vytvoření nástroje, který automaticky zanalyzuje zdrojový kód a ze získaných informací vygeneruje jednoduchou dokumentaci konfigurace ve formátu, který bude pro uživatele snadno zobrazitelný a čitelný bez nutnosti nahlížení do kódu či programátorské dokumentace. Zdrojem dokumentace budou standardní Javadoc komentáře, které jsou součástí zdrojového kódu aplikace. Tak bude zajištěna konzistence mezi kódem, programátorskou dokumentací a uživatelskou dokumentací popisující konfiguraci aplikace.

[1] Uživatelem se zde rozumí ten, kdo aplikaci spouští a provozuje a tedy je po něm vyžadována její konfigurace.

Kapitola 2
Představení technologií

Na začátku práce je vhodné seznámit se s technologiemi, které souvisí s tématem, a mít přehled o jejich významu a vztahu k tématu. V následujících sekcích budou stručně představeny relevantní technologie, které budou dále v práci využívány.

2.1 Apache Maven

Apache Maven je nástroj pro správu Java projektů. Zajišťuje kompilaci, testování, analýzu, generování dokumentace a nabízí další možnosti automatizace. Významnou součástí je správa a distribuce závislostí – pro tento účel zavádí repozitáře, které uchovávají výstupy projektů (artifakty). Artifaktem může být zkompilovaný soubor, zdrojový kód, nebo také dokumentace. Repozitář může být lokální, nebo vzdálený. Lokální obsahuje kopie závislostí ze vzdálených repozitářů a funguje tedy jako cache závislostí pro účely kompilace místních projektů. Také může obsahovat artifakty lokálních projektů, které nebyly publikovány do vzdáleného repozitáře. Vzdálený je jakýkoli jiný repozitář přístupný přes protokoly jako „`file://`“ a „`https://`“. Proces sestavení projektu Maven rozděluje do několika fází:

1. `Validate` – Ověří správnost definice projektu a dostupnost všech nezbytných informací.
2. `Compile` – Zkompiluje zdrojový kód projektu.
3. `Test` – Otestuje zkompilovaný kód pomocí vhodného testovacího frameworku.
4. `Package` – Zabalí zkompilovaný kód do požadovaného formátu, jako je například JAR.
5. `Verify` – Zkontroluje, zda výsledky integračních testů splňují požadovaná kritéria.
6. `Install` – Nainstaluje artifakt do lokálního repozitáře.
7. `Deploy` – Zkopíruje výsledný balík do vzdáleného repozitáře.

Hlavní fáze se ještě dále dělí na menší podfáze jako „`pre-compile`“, „`post-compile`“ atd. Každá fáze navazuje na tu předchozí, tedy spuštění testů vyžaduje kompilaci, ale nezávisí na zabalení (package). Ve skutečnosti Maven sám o sobě s projektem nijak nemanipuluje, všechny funkce jsou zaštítěny pluginy. Pro každou fázi je definován výchozí plugin. Ten ovšem může být nahrazen jiným. V rámci jedné fáze může být také spuštěno více pluginů. Jeden plugin se také může nacházet ve více fázích současně. Maven se tedy „pouze“ stará o spuštění příslušných pluginů v jejich fázích.

Pro popis a konfiguraci projektu používá Maven „Project Object Model“ (POM), reprezentovaný souborem `pom.xml`. Konfigurace obsahuje informace o projektu (skupinu, název artifaktu, ale také třeba nezbytnou verzi jazyka Java), závislosti, pluginy a jejich konfigurace, viz příklad konfigurace Maven projektu 2.1 [2].

```xml
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>cz.lukaskabc.cvut.processor</groupId>
  <artifactId>spring_configuration</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Spring Boot project with configuration</name>

  <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.7.14</version>
      <relativePath/>
  </parent>

  <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <start-class>cz.lukaskabc.cvut.processor.App</start-class>
      <java.version>11</java.version>
  </properties>

  <dependencies>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
      </dependency>

      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-configuration-processor</artifactId>
      </dependency>

      <!-- ... -->
  </dependencies>

  <build>
      <plugins>
          <plugin>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>

          <!-- ... -->
      </plugins>
  </build>
</project>
```

Výpis kódu 2.1. Příklad konfigurace Maven projektu v souboru `pom.xml`.

2.2 Anotace

Anotace v jazyce Java, psané se znakem „@“ na začátku, umožňují spojit metadata s konkrétními elementy programu, např. s třídami, metodami, nebo atributy [3]. Jsou převážně využívány během kompilace, kdy slouží jako upřesňující instrukce, které blíže popisují kód. Anotace jsou definovány jako rozhraní označené klíčovým slovem „`@interface`“. Ukázku definice anotačního typu je možné vidět ve výpisu kódu 2.2. V ukázce jsou definovány „metody“ `id`, `synopsis`, `engineer` a `date`. Protože se jedná o definici anotace, jedná se spíše o parametry, jejichž hodnotu je nutné specifikovat při použití anotace, pokud není určena výchozí hodnota. Prostřednictvím těchto metod je pak možné zpětně přistupovat k hodnotám, které byly specifikovány při použití anotace. Ve výpisu kódu 2.3 je znázorněno použití takové anotace s nezbytnými parametry.

Pokud není u anotačního typu určeno jinak (prostřednictvím anotace `@Retention`), jsou anotace využity pouze během kompilace, během které jsou odebrány. Pokud je ovšem definováno, že má být anotace ponechána i pro spuštění programu („`@Retention(RetentionPolicy.RUNTIME)`“), je možné k anotaci a jejím parametrům přistoupit prostřednictvím reflexe. Toho využívá řada frameworků, které tak mohou získat informace o elementech vytvořených uživatelem.

```
@interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer()  default "[unassigned]";
    String date()      default "[unimplemented]";
}
```

Výpis kódu 2.2. Definice anotace `@RequestForEnhancement` s atributy, převzato z [3]

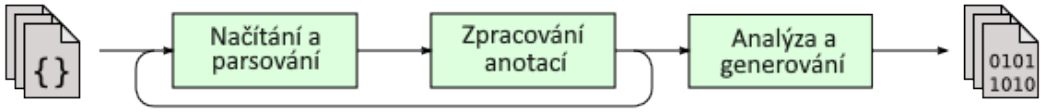
```
@RequestForEnhancement(
id       = 4561414,
synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

Výpis kódu 2.3. Použití anotace definované ve výpisu kódu 2.2, převzato z [3]

2.2.1 Anotační procesory

Anotační procesory jsou nástroje využívané kompilátorem pro zpracování anotací ve zdrojovém kódu. Rozhraní pro anotační procesory bylo primárně určeno pro generování nových souborů (například tříd) na základě anotací. Avšak existují způsoby, jakými lze obejít toto základní rozhraní a využít API kompilátoru pro přímé úpravy zpracovávaného kódu. Proces kompilace probíhá ve třech fázích, jak je znázorněno na obrázku 2.1.

1. Všechny zdrojové soubory, které byly specifikovány při spuštění kompilátoru, jsou načteny a je vytvořen „syntax tree“, tedy reprezentace zdrojového kódu ve formě stromové struktury.
2. Jsou spuštěny anotační procesory. Proces kompilace se opakuje, dokud anotační procesory generují nové zdrojové, nebo class[1] soubory.
3. Výsledné „syntax trees“ jsou zpracovány a přeloženy do class souborů [4].



Obrázek 2.1. Proces kompilace zdrojového kódu Java, převzato z [4], přeloženo

2.3 Spring

Populární open source Java framework[2] pro vytváření produkčních aplikací. Dle JetBrains průzkumů se za posledních šest let jedná o nejpoužívanější webový framework v ekosystému jazyka Java [5]. Spring poskytuje Inversion of Control (IoC) a Dependency Injection (DI) kontejner [6], který se stará o spuštění a následné řízení životního cyklu aplikace. V jeho rámci také zajišťuje inicializaci a propojení jednotlivých komponent bez nutnosti vytváření vzájemných úzkých vazeb.

Spring se skládá z četného množství modulů, které jednotlivě implementují různé funkce. Ne každá aplikace je potřebuje všechny – kvůli modularizaci může importovat pouze ty moduly, které vyžaduje. Příkladem může být modul Spring MVC, který zajišťuje HTTP rozhraní aplikace, nebo moduly pro integraci s různými databázemi s podporou objektově-relačního mapování [7–8].

2.3.1 Spring Boot

Knihovna, která značně zjednodušuje konfiguraci a umožňuje jednoduchý start vývoje. Framework Spring je sám o sobě velmi komplexní a obsáhlý. Jeho modularita umožňuje přizpůsobení jednotlivých funkcí, podmínkou je ovšem patřičná konfigurace. Spring Boot přebírá konkrétní nastavení klientského projektu a předpokládá jeho potřeby, na jejichž základě doplní další výchozí nastavení. Nabízí automatickou konfiguraci frameworku a základních knihoven třetích stran, umožňuje také spuštění samostatné aplikace, což u Spring frameworku pro webové aplikace není běžně možné, a vyžaduje nasazení na aplikační server (Apache Tomcat nebo jiné). Prakticky je funkcí knihovny Spring Boot doplnit aplikaci o nezbytné komponenty a konfiguraci pro její spuštění [7, 9].

[1] Class soubor má příponu `.class` a obsahuje Java bytecode, který je výstupem kompilace.
[2] Framework je software či platforma poskytující základní stavební kameny a strukturu, kterou vývojáři využívají pro budování nové aplikace, aniž by museli vše programovat od samotných základů.

2.3.2 Spring Boot konfigurace

Framework Spring umožňuje externalizaci konfigurace, tedy umístění konfigurace mimo zdrojový kód do snadno editovatelného zdroje, který je možné následně dynamicky načítat za běhu programu. Spring Boot přináší možnost tuto konfiguraci mapovat na konfigurační třídy. Zdrojem hodnot mohou být konfigurační soubory s podporou různých formátů (properties, YAML, JSON, viz příklad konfigurace 2.4), proměnné prostředí (environmental variables) a argumenty z příkazové řádky. Hodnoty z těchto zdrojů je Spring schopný na základě anotace `@Value`[3] mapovat na atribut třídy (hodnota atributu), metodu (metoda bude zavolána při načtení konfigurace s hodnotami parametrů z konfigurace), parametr metody (v kombinaci s anotací metody blíže specifikuje hodnotu konkrétního parametru), parametr konstruktoru (pokud je konstruktor anotován `@Autowired`[4], bude objekt vytvořen s hodnotou parametru z konfigurace) nebo ji použít jako součást jiného anotačního typu. Použití na atributu a na metodě je naznačeno ve výpisu kódu 2.5. Anotace `@Value` je vyhodnocována při spuštění kompilované Spring Boot aplikace. Jako argument anotace String přijímá parametr, který specifikuje název a cestu ke konfigurační hodnotě. Argumentem může být SpEL výraz 2.3.3 v podobě `#{systemProperties.propertyName}`, nebo cesta k hodnotě v konfiguraci ve formě `${my.app.propertyName}`. Při spuštění aplikace jsou všechny zdroje konfigurací načteny do strukturovaného kontextu, pomocí syntaxe `${}` se specifikuje cesta k hodnotě v rámci této struktury.

Takto je možné mapovat jednotlivé hodnoty. Druhou možností, kterou Spring Boot nabízí, je anotace `@ConfigurationProperties`[5]. Tu je možné použít buďto na třídu, nebo na metodu. Způsob jejího mapování na třídu je vidět na obrázku 2.2. Za vytvoření instance anotované třídy zodpovídá Spring, který při vytvoření přiřadí atributům hodnoty a dále objekt spravuje. Může být vytvořen automaticky na základě anotace třídy, nebo také metodou s anotací `@Bean`[6,7], jak je vidět ve výpisu kódu 2.6. Objekt, který taková metoda vrátí, je začleněn do Spring kontextu. Pokud má metoda navíc anotaci `@ConfigurationProperties`, jsou na objekt namapovány hodnoty z konfigurace. Tento způsob mapování podporuje „relaxed binding“, který přináší jistou volnost v zápisu názvů konfigurace[8]. U YAML hodnot je například možné zaměnit následující dva formáty – „kebab-case“ `prefix.sub-class.sub-value` a „camelCase“ `prefix.subClass.subValue`. Různé formáty zápisu umožňují mapování i z proměnných prostředí, kde lze v názvu použít pouze alfanumerické znaky a podtržítko [10]. Ekvivalentní zápisy konfigurace lze vidět na příkladu 2.7. Očekávaným formátem názvu proměnné prostředí je tedy `PREFIX_SUBCLASS_SUBVALUE`. Při mapování hodnot konfigurace je automaticky provedena konverze do cílového datového typu [9, 11–12].

[3] `org.springframework.beans.factory.annotation.Value`
[4] `org.springframework.beans.factory.annotation.Autowired`
[5] `org.springframework.boot.context.properties.ConfigurationProperties`
[6] `org.springframework.context.annotation.Bean`
[7] Bean je označení pro objekt, jehož životní cyklus je spravován Spring kontextem.
[8] Relaxed binding není u anotace `@Value` podporován.

```
# properties
value.attribute="attribute"
value.method=10
# YAML
value:
    attribute: "attribute"
    method: 10
# JSON
"value": {
    "attribute": "attribute",
    "method": 10
}
```

Výpis kódu 2.4. Konfigurace pro třídu `ValueAnnotationConfig` 2.5, uvedeny jsou formáty properties, YAML a JSON

```
@Configuration
public class ValueAnnotationConfig {
    @Value("${value.attribute:#{null}}")]
    String attribute;

    Integer methodValue;

    @Value("${value.method}")
    public void setMethodValue(Integer parameter) {
        methodValue = parameter;
    }
}

@Value("#{valueAnnotationConfig.attribute != null}")
boolean isConfigAttributePresent;
```

Výpis kódu 2.5. Spring Boot konfigurační třída využívající anotaci `@Value` a SpEL výrazy

```
@Configuration
@ConfigurationProperties(prefix = "prefix")
public class ConfigurationPropertiesConfig {

    String value;

    SubClass subClass = new SubClass();

    public static class SubClass {
        Integer subValue;
    }
}
```

```
prefix.value: "attribute"
prefix.subClass.subValue: 10
```

Obrázek 2.2. Mapování konfigurace na třídu s anotací `@ConfigurationProperties`

```
@Configuration()
public class ValueAnnotationConfig {

    @Bean
    @ConfigurationProperties(prefix = "prefix")
    public ConfigurationPropertiesConfig getConfiguration() {
        return new ConfigurationPropertiesConfig();
    }
}
```

Výpis kódu 2.6. Použití anotací `@ConfigurationProperties` a `@Bean` na metodě

```
# V konfiguračním souboru jsou následující zápisy ekvivalentní:
prefix.subClass.subValue: 10
prefix.sub-class.sub-value: 10
prefix.sub_class.sub_value: 10
prefix.subClass.sub-value: 10 # zápisy je možné i kombinovat

# Ekvivalentní zápis proměnné prostředí (environmental value):
PREFIX_SUBCLASS_SUBVALUE=10
```

Výpis kódu 2.7. Ekvivalentní zápisy konfigurace využívající relaxed binding

2.3.3 Spring Expression Language

Spring Expression Language (SpEL) je jazyk pro dotazování a manipulaci s objekty za běhu programu. Vznikl za účelem poskytnutí jednotného formátu výrazů v ekosystému Spring projektů. Dokáže zpracovávat regulární výrazy, volat metody, konstruktory, přiřazovat hodnoty a manipulovat s nimi, procházet kolekce i přistupovat k referencím. Každý výraz je vyhodnocován ve svém konkrétním kontextu, v rámci kterého může přistupovat k proměnným [8].

Ve výpisu kódu 2.5 je znázorněna konfigurační třída `ValueAnnotationConfig` s atributem `String attribute`, do kterého je dosazena hodnota z konfigurace (`value.attribute`). Zároveň je zde použita výchozí hodnota, pokud není v konfiguraci definována. V takovém případě je dosazen výsledek SpEL výrazu, tedy `null`. Třída `ValueAnnotationConfig` je instancována jako objekt (Bean) s názvem `valueAnnotationConfig`. V jiné komponentě lze použít výraz, který odkazuje na vytvořený objekt konfigurace `valueAnnotationConfig` a jeho atribut, který porovnává s hodnotou `null`, výsledek je uložen do atributu. Pokud je tedy atribut v konfiguraci nedefinovaný, je hodnota záporná (false), v opačném případě kladná (true).

2.4 JSR 303: Bean Validation

S tématem konfigurace úzce souvisí způsob validace. Přirozeně ne každá hodnota dává smysl pro konkrétní konfiguraci. Za využití Spring Boot konfiguračních tříd se validní hodnoty výrazně zúží na vyžadovaný datový typ. I přesto zůstává počet možných hodnot příliš velký. Spring Boot proto nabízí u `@ConfigurationProperties` anotovaných tříd možnost blíže omezit povolené hodnoty jednotlivých atributů díky podpoře JSR[9] 303 anotací. Pokud načtená hodnota nesplní stanovená omezení, je vyhozena výjimka

[9] Java Specification Request (JSR) je prvním krokem ve vývoji specifikace Java technologií.

a spuštění Spring Boot aplikace je přerušeno. V dokumentaci je vhodné tato omezení uvést, protože chybné hodnoty budou bránit úspěšnému spuštění aplikace.

Specifikací Java API pro validaci JavaBean je JSR 303, později vznikly navazující specifikace JSR 349 a 380. V práci bude jednotně odkazováno na JSR 303. U Spring konfigurační třídy (`@ConfigurationProperties`) je možné validaci aktivovat Spring anotací `@Validated`. Pro kontrolu vnořených struktur je ještě vyžadována anotace `@Valid` na konkrétním atributu. Příkladem validačních anotací jsou `@NotNull` – hodnota není null, `@Min` – hodnota je větší nebo rovna parametru, `@Max` – hodnota je menší nebo rovna a další [13].

2.5 Javadoc

Nástroj distribuovaný jako součást Java Development Kitu (JDK) *„generuje ze zdrojových Java souborů HTML stránky API dokumentace“* [14]. Javadoc za pomoci Java kompilátoru vybuduje hierarchickou strukturu tříd, ze které vygeneruje výsledný HTML dokument. Vždy dochází ke kompletní nové generaci.

Pro Javadoc jsou klíčovou částí zdrojového kódu komentáře. Jazyk Java využívá běžné komentáře `/* blokové */` a `// řádkové`. Komentáře ve tvaru `/** Javadoc */` jsou interpretovány jako Javadoc. Ty je možné uvést u třídy (class), rozhraní (interface), konstruktoru, metody, nebo atributu.

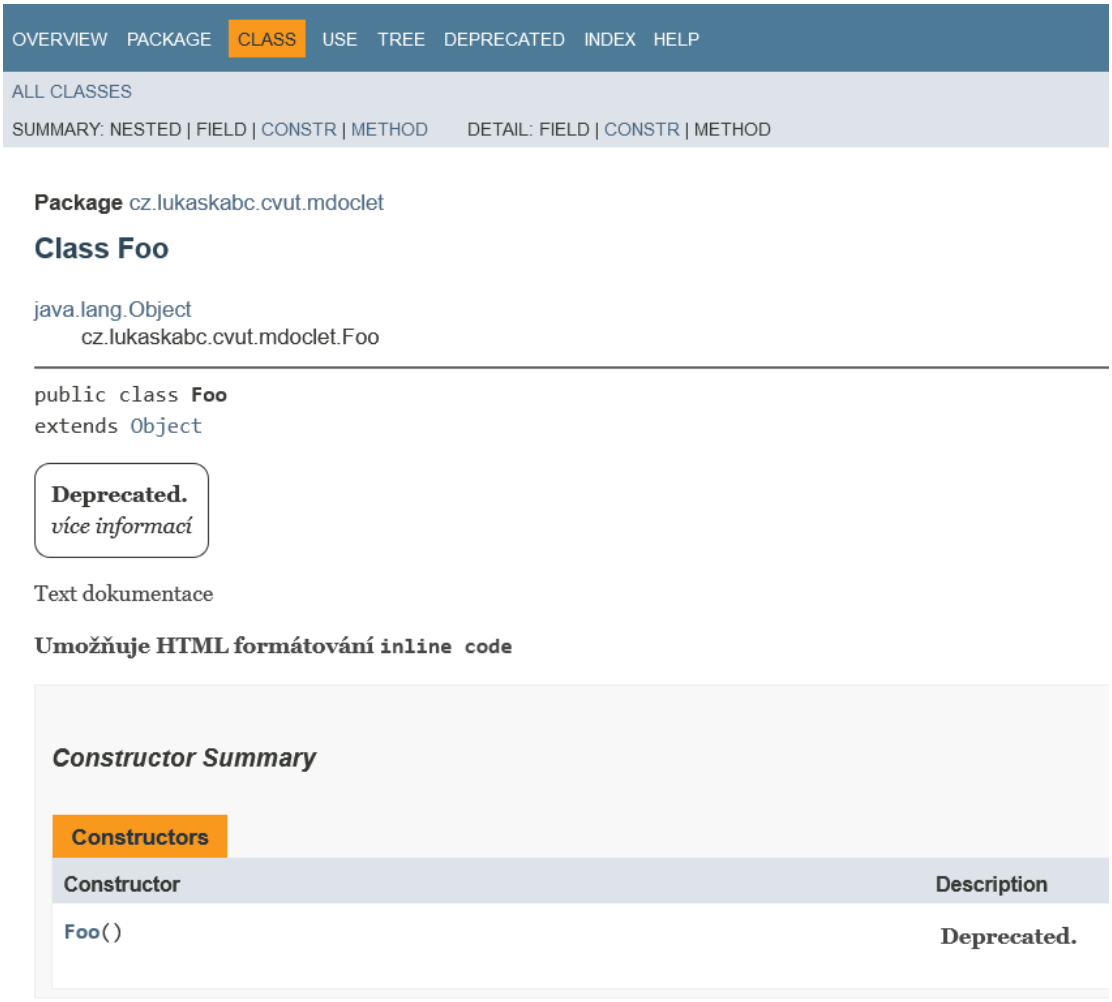
Dokumentace je generována do formátu HTML, viz obrázek 2.3, a jak je znázorněno ve výpisu kódu 2.8, i Javadoc komentáře mohou obsahovat HTML formátování. Další funkcí jsou tagy. Jedná se o klíčová slova, s jejichž pomocí lze dokumentaci obohatit např. o formátování nebo související odkazy. V ukázce je jako příklad uveden tag `@code`, který poskytuje formátování kódu, jež je součástí komentáře, a tag `@deprecated` poskytuje upozornění o zastaralém rozhraní a případné bližší informace [14].

Javadoc komentáře jsou součástí standardu a samotného jazyka Java, nachází se přímo ve zdrojovém kódu a generuje se z nich programátorská dokumentace. Komentáře mohou být umístěny u atributů tříd, na které je mapována konfigurace, a tedy jsou vhodným zdrojem dokumentace pro konfiguraci aplikace.

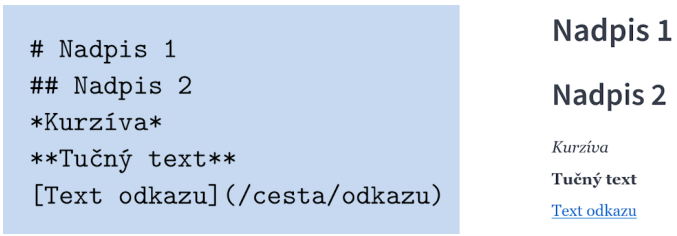
```
package cz.lukaskabc.cvut.processor;

/**
* Text dokumentace
* <p>
* <b>Umožňuje HTML formátování {@code inline code}</b>
* @deprecated více informací
*/
public class Foo {}
```

Výpis kódu 2.8. Ukázka Javadoc komentáře u třídy, použity HTML tagy, Javadoc inline i blokové tagy



Obrázek 2.3. Ukázka vygenerované Javadoc dokumentace



Obrázek 2.4. Základní syntaxe Markdown formátu a její výstup

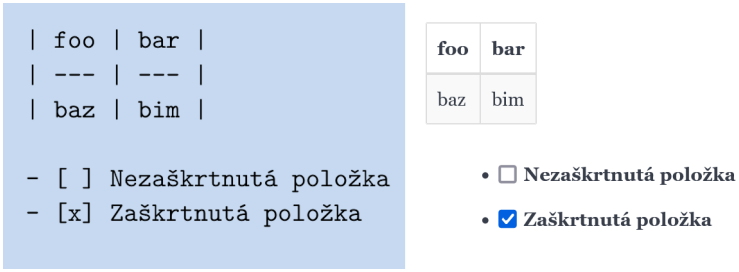
2.6 Markdown

Jednoduchý textový formát určený pro psaní strukturovaných dokumentů. Původní implementace představovala konverzi do HTML. Díky své čitelnosti se stal Markdown velmi rozšířeným formátem. Nyní se využívá pro formátování komentářů, příspěvků, a dokonce i knih. Na obrázku 2.4 je zobrazeno základní formátování a jeho výstup [15].

2.6.1 GitHub Flavored Markdown

Dialekt formátu Markdown podporovaný portálem GitHub. Zpřísňuje původní formátování a rozšiřuje ho o nové funkce, případně ty existující upravuje, aby lépe zapadaly do ekosystému GitHubu. Jmenovitě tedy přidává podporu pro tabulky, seznamy, přeškrtnutí (strikethrough) a odkazy. Tabulka na obrázku 2.5 má dva sloupce a dva řádky, přičemž první řádek je interpretován jako záhlaví, pod tabulkou pak následuje zápis pro zaškrtávací položky (checkbox) [16].

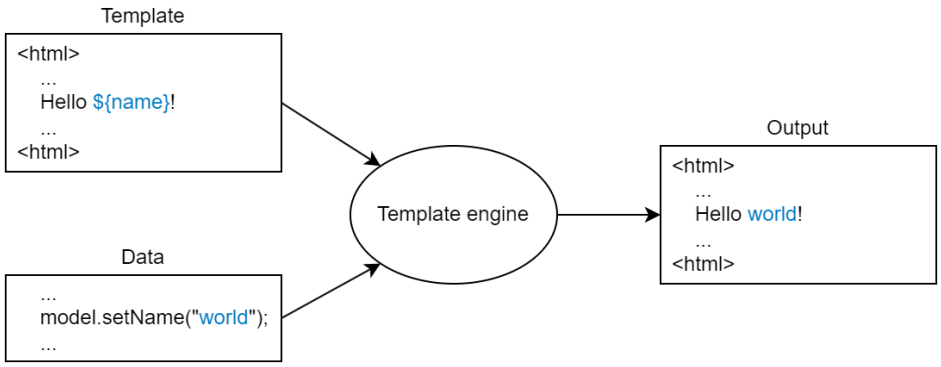
Značná část open source projektů je právě umístěna na portálu GitHub, kam je tedy vhodné umístit i základní dokumentaci včetně popisu konfigurace. Svou jednoduchostí a rozšířenou podporou je Markdown, přesněji GitHub Flavored Markdown (GFM), podporovaný portálem GitHub, vhodným formátem pro výslednou dokumentaci konfiguračních možností aplikace. Portál GitHub také podporuje použití HTML tagů v Markdown souborech.



Obrázek 2.5. Github Flavored Markdown formát

2.7 Template Engine

Template engine, nebo také template processor, je program, často ve formě knihovny, který umožňuje generovat textový výstup na základě šablony a vstupních dat. Šablona je textový soubor, který definuje strukturu a obsah výsledného textu. Obsahuje specifické značky, nebo textové řetězce, které slouží jako instrukce pro template engine. Tyto značky jsou nejčastěji nahrazeny korespondujícími hodnotami ze vstupních dat, ale také mohou nabízet další možnosti a pokročilejší funkce. Obrázek 2.6 znázorňuje základní princip fungování template enginu. Na levé straně je vstupem šablona s naznačenou kostrou HTML dokumentu společně se vstupními daty a na pravé straně výstup, který byl vygenerován vložením dat do šablony [17].



Obrázek 2.6. Princip fungování template enginu, převzato z [17], upraveno

Kapitola 3
Možná řešení

V následujících kapitolách budou rozebrána možná řešení, jejich teoretické možnosti, výhody, nevýhody i případné zábrany pro jejich volbu. Hlavními kritérii pro výběr výsledného řešení jsou složitost návrhu, implementace a náročnost integrace do existujícího projektu.

3.1 Existující nástroje

Navzdory velikosti a rozšířenosti frameworku Spring (resp. Spring Boot) existuje velmi malé množství nástrojů, které by se zabývaly generováním dokumentace zaměřené na konfiguraci aplikace.

3.1.1 Jamal Macro Language

Jamal (Java Macro Language) je nástroj pro automatizaci údržby dokumentace. Podporuje řadu výstupních formátů jako Asciidoc, Markdown, XML a další. Jedná se o metaznačkovací jazyk, který rozšiřuje možnosti původního (výstupního) jazyka. Funguje na principu šablon, které jsou zpracovány a na jejich základě je vygenerována výsledná dokumentace. Nabízí komplexní možnosti zpracování vstupních souborů prostřednictvím celé řady maker, která jsou připravena pro zpracování běžného textu i zdrojových souborů některých konkrétních programovacích jazyků. Nenabízí přímou podporu pro dokumentování Spring Boot konfigurace, ale umožňuje v kódu označit Javadoc komentáře a ty pak využít v šabloně. Nevýhodou je komplikované prvotní nastavení, které vyžaduje vytvoření vlastní šablony a případné úpravy zdrojového kódu dokumentované aplikace, tedy označení komentářů ke zdokumentování [18].

3.1.2 Spring Boot Configuration Processor

Knihovna Spring Boot nabízí jako jednu ze svých součástí nástroj Spring Boot Configuration processor. Není přímo určen ke generování dokumentace, ale pro její popis, přesněji pro generování metadat. Procesor zpracovává konfigurační třídy s anotací `@ConfigurationProperties`[1] a generuje soubor s metadaty ve formátu JSON. Zkrácená ukázka takového výstupu se nachází ve výpisu kódu 3.1. Soubor se skládá ze tří částí. Skupiny – „groups“, které poskytují kontext pro jednotlivé konfigurační parametry. Standardně se tedy jedná o konfigurační třídy. Položky – „properties“, které reprezentují jednotlivé konfigurační parametry, a nápovědy – „hints“, které slouží pro dodatečné informace ke konkrétním hodnotám konfiguračních parametrů [9]. Tato metadata pak využívají například vývojová prostředí pro napovídání při úpravě konfiguračních souborů.

[1] Konfigurace prostřednictvím anotace `@Value` není podporována.

```
{
    "groups": [
        {
            "name": "termit",
            "type": "cz.cvut.kbss.termit.util.Configuration",
            "sourceType": "cz.cvut.kbss.termit.util.Configuration"
        }
    ],
    "properties": [
        {
            "name": "termit.file.storage",
            "type": "java.lang.String",
            "description": "Specifies root directory ...",
            "sourceType": "cz.cvut.kbss.termit.util.Configuration$File"
        },
        {
            "name": "termit.url",
            "type": "java.lang.String",
            "description": "TermIt frontend URL. <p> It is used, ...",
            "sourceType": "cz.cvut.kbss.termit.util.Configuration",
            "defaultValue": "http:\/\/localhost:3000\/#"
        }
    ],
    "hints": []
}
```

Výpis kódu 3.1. Ukázka souboru s metadaty vygenerovaného nástrojem Spring Boot Configuration processor pro projekt TermIt [19], pro účely ukázky byl obsah souboru zkrácen

3.1.3 Spring Configuration Property Documenter

Tento nástroj vznikl za účelem dokumentace konfiguračních tříd Spring Boot projektu. Zpracovává výstupní metadata ze Spring Boot Configuration procesoru a na jejich základě generuje dokumentaci, je tedy na něm závislý. Z této závislosti vyplývá, že také nepodporuje konfiguraci prostřednictvím anotace `@Value`. Výhodou zde je, že zpracování samotné konfigurace probíhá nástrojem, který je součástí Spring Boot knihovny. Naopak tím, že zpracovává pouze text (JSON), ztrácí informace o bližším kontextu, které Spring Boot Configuration processor nezpracovává. Například význam některých Javadoc tagů či další anotace (JSR 303) [20].

Řešení této práce by nemělo být závislé na Spring Boot Configuration procesoru a mělo by tedy zpracovávat původní kód konfiguračních tříd. V budoucnu, v případě zásadních změn v konfiguraci Spring Boot projektů, může dojít ke vzniku nekompatibility těchto změn s výsledným nástrojem této práce. Na druhou stranu bude nástroj schopen zpracovávat kompletní kontext konfigurace včetně ostatních anotací a Javadoc tagů.

3.2 Struktura

První otázkou, která vzniká při návrhu řešení, je struktura programu. Jako možná řešení byla zvážena samostatná aplikace a Maven plugin. Se zvolenou strukturou programu blízce souvisí její způsob použití, tedy způsob instalace, konfigurace a další požadavky pro spuštění a běh.

3.2.1 Samostatná aplikace

Přirozenou možností je vytvoření nové samostatně spustitelné aplikace. Vzhledem k jejímu účelu, tedy použití při vývoji, lze snadno předpokládat přítomnost JDK v cílovém prostředí a tedy volba jazyka Java je vcelku přímočará. Aplikace by však vyžadovala svou specifickou konfiguraci, potřebovala by definovat soubory, se kterými má pracovat, a případné další parametry. Vývoj by probíhal prakticky od samotného začátku, vše by se tvořilo od nuly včetně čtení souborů, tedy zpracování zdrojového kódu a jeho následné reprezentace. Případnému využití již existujících nástrojů, resp. technologií je věnována kapitola 3.3. Hlavním negativem by zde bylo oddělené nastavení specifické pouze pro konkrétní aplikaci a nutnost explicitně ji spouštět.

3.2.2 Maven Plugin

Maven plugin je Java aplikace, která obsahuje jednu nebo více „Mojo“ tříd. Každá taková třída představuje cíl (funkci) pluginu. Cíl je samostatně zařazen do určité fáze životního cyklu projektu. Jeden plugin tedy může tímto způsobem nabízet více cílů (funkcí) a může být zapojen do více fází. Každý takový cíl je reprezentován „Mojo“ třídou rozšiřující `AbstractMojo`[2] a navíc s anotací `@Mojo`[3]. Ukázku minimalistické implementace můžeme nalézt ve výpisu kódu 3.2. Anotací plugin sdělí Mavenu své základní informace jako název, fázi, ve které se má vykonat, a případně dostupné parametry. Maven pak automaticky plugin v příslušné fázi spustí [2].

Druhou možností řešení by tedy bylo vytvoření Maven pluginu, který by provedl analýzu kódu a vygeneroval by patřičnou dokumentaci. Výhodou by byla snadná konfigurace a integrace s ekosystémem existujících Java projektů.

```java
@Mojo(name = "sayhi")
public class GreetingMojo extends AbstractMojo
{
    public void execute() throws MojoExecutionException
    {
        getLog().info("Hello, world.");
    }
}
```

Výpis kódu 3.2. Minimální „Mojo“ třída Maven pluginu, převzato z [2]

[2] `org.apache.maven.plugin.AbstractMojo`
[3] `org.apache.maven.plugins.annotations.Mojo`

3.3 Technologie pro zpracování kódu

Protože by bylo velmi náročné navrhovat a implementovat řešení zcela od začátku, bude vhodné zvolit některou z existujících technologií, která umožní snadné načtení zdrojového kódu, reprezentaci jeho struktury, analýzu a další zpracování.

Jedním z kritérií, které je zde nutné zohlednit, je zdroj dokumentace pro jednotlivé konfigurační hodnoty. Dokumentace se bude čerpat z Javadoc komentářů, které byly popsány v kapitole 2.5. Zvažovanou alternativou bylo použití zvláštních anotací, které by byly určeny pro dokumentaci. Tato možnost je popsána v následující sekci 3.3.1.

Následující kapitoly představí několik technologií, které byly uváženy jako možná řešení. V kapitole 3.4 budou tyto technologie porovnány a zhodnoceny pro jejich použití v rámci této práce.

3.3.1 Runtime reflexe

Jedná se o funkci programovacího jazyka Java, která je úzce spjata s Java Virtual Machine[4]. Umožňuje vykonávanému programu „zkoumat“ vlastní strukturu. Je možné, aby program získal informace o svých třídách, metodách, konstruktorech, atributech a využít je k jejich úpravě [21]. Příkladem může být volání setteru na základě znalosti názvu atributu, jak je vidět ve výpisu kódu 3.3. Samozřejmě se zde předpokládá, že název setteru má konkrétní tvar „set“ + název atributu – `setAttribute` [22]. V ukázce je nejdříve vytvořena instance třídy `Foo`, definován název setteru a hodnota, která má být nastavena. Následně je z definice třídy `Foo` za pomoci Java Reflection API získána metoda setteru s odpovídajícím názvem a typem parametru. Nakonec je metoda pomocí reflexe zavolána na objektu `foo` s argumentem nové hodnoty.

„S velkou mocí přichází i velká zodpovědnost“ – reflexe je mocný nástroj, ve většině případů, kdy není explicitně nutná, se ale nevyužívá. Hlavními problémy používání reflexe jsou:

- Zabezpečení – Reflexe narušuje podstatu zapouzdření a umožňuje přístup k neveřejným prvkům, ke kterým by běžně přístup nebyl možný.
- Komplexita – Značně navyšuje velikost kódu a jeho složitost.
- Výkon – Všechny úkony prováděné pomocí reflexe jsou dynamicky vyhodnocovány za běhu programu. To znemožňuje aplikování některých optimalizací a má tedy negativní dopad na rychlost [23].

Příkladem širokého využití reflexe může být framework Spring, nebo obecně IoC frameworky, které využívají reflexi pro minimalizaci nutných úprav kódu klientských aplikací. Pro představu může posloužit mapování konfigurace na konfigurační třídy na obrázku 2.2, kde je vyžadováno, aby Spring nějakým způsobem konfigurační třídu „prozkoumal“ a naplnil hodnotami.

Využití reflexe pro analýzu kódu a následné generování dokumentace by přineslo možnost integrace s frameworkem Spring, tedy spuštění samotné aplikace, a validování konfigurace, nebo také vyhodnocování SpEL výrazů. Velmi podstatnou nevýhodou je zde přístup k Javadoc komentářům. Reflexe pracuje s třídami za běhu programu, Javadoc komentáře jsou při kompilaci společně se všemi ostatními komentáři odstraněny. Jednoduše reflexí není možné k Javadoc komentářům jakkoli přistoupit.

[4] Java Virtual Machine je virtuální prostředí zodpovědné za vykonávání Java bytecodu, do kterého je zdrojový kód kompilován.

```
class Foo {
    private String attribute;

    public void setAttribute(String value) {
        this.attribute = value;
    }
}

public class Reflection {
    public static void main(String[] args) throws Exception {
        Foo foo = new Foo();

        String setterName = "set" + "Attribute";
        String valueToSet = "value";

        Method m = Foo.class.getDeclaredMethod(setterName, String.class);

        m.invoke(foo, valueToSet);
    }
}
```

Výpis kódu 3.3. Ukázka použití Java reflexe k zavolání metody podle názvu a typu parametrů

```
@Configuration
class MyConfiguration {

    @Documentation("Integer value")
    Integer intValue;
}
```

Výpis kódu 3.4. Dokumentace atributu třídy pomocí anotace

Pro možnost dokumentace jednotlivých elementů by bylo tedy nutné zavést nové zvláštní anotace. Takový přístup využívá například knihovna Springdoc, která se zaměřuje na dokumentaci REST API Spring Boot aplikací [24]. Tyto anotace by jako parametr přijímaly text (string), který by byl interpretován jako dokumentace, viz výpis kódu 3.4, kde je použita jako příklad anotace `@Documentation` na atributu třídy.

3.3.2 JavaParser

Nástroj pro čtení a následnou reprezentaci zdrojového kódu Java. Je přístupný v podobě knihovny dostupné v Maven centrálním repozitáři. Umožňuje načtení zdrojového kódu a jeho následné procházení, analýzu, transformaci, úpravy a export. V době psaní této práce podporuje jazyk Java od verze 1.0 až do verze 17. Nabízí naprostou volnost v analýze a manipulaci se zdrojovým kódem. Čte kód ze souborů bez jeho spuštění, bez ohledu na použité knihovny, reprezentuje kód přesně tak, jak je [25].

Ve výpisu kódu 3.5 je znázorněno použití knihovny JavaParser pro invertování bloků podmínek – `if(x !=y) a else b` bude upraveno na `if(x == y) b else a`. Výhodou použití tohoto nástroje je bezpochyby absolutní volnost ve čtení kódu. Naopak nevýhodou je náročnost implementace a závislost řešení na externím nástroji.

```
public class Fix {
  public static void main(String[] args) throws IOException {

    SourceRoot sourceRoot = new SourceRoot(
        CodeGenerationUtils.mavenModuleRoot(Fix.class)
        .resolve("src/main/resources"));

    CompilationUnit cu = sourceRoot.parse("", "Foo.java");

    cu.accept(new ModifierVisitor<Void>() {
      @Override
      public Visitable visit(IfStmt n, Void arg) {

      Expression condExpr = n.getCondition();
      if (condExpr instanceof BinaryExpr) {
        BinaryExpr cond = (BinaryExpr) condExpr;
        if (cond.getOperator() == BinaryExpr.Operator.NOT_EQUALS &&
                n.getElseStmt().isPresent()) {

          Statement thenStmt = n.getThenStmt().clone();
          Statement elseStmt = n.getElseStmt().get().clone();
          n.setThenStmt(elseStmt);
          n.setElseStmt(thenStmt);
          cond.setOperator(BinaryExpr.Operator.EQUALS);
        }
      }
      return super.visit(n, arg);
      }
    }, null);

    sourceRoot.saveAll();
  }
}
```

Výpis kódu 3.5. Ukázka použití knihovny JavaParser, invertuje bloky podmínek v načteném kódu, převzato z [25]

3.3.3 Zpracování HTML

Hyper Text Markup Language (HTML) je značkovací jazyk, který definuje strukturu stránky. Nástroj Javadoc, který byl popsán v kapitole 2.5, generuje ve výchozím nastavení dokumentaci právě ve formátu HTML. V tomto případě se jedná o strojově generovaný kód, který by měl být validní s minimálním, ne-li nulovým množstvím chyb syntaxe jazyka. Možnost použití HTML tagů v Javadoc komentářích může naopak validitu narušit, pokud nejsou tagy v Javadoc komentářích použity správně.

Jedním z možných řešení je tedy zpracování výstupu nástroje Javadoc. V rámci projektu by došlo ke spuštění standardního Javadoc nástroje a vygenerování dokumentace. Následně by se provedla analýza vygenerovaných dokumentů, odkud by se posbíraly informace o konfiguraci. Načtení a reprezentace dokumentace by byla možná za použití

dostupných knihoven. Například pomocí knihovny Jsoup[5]. Toto řešení by se však muselo vázat na konkrétní strukturu generované dokumentace. Jakákoli změna v této struktuře by mohla vést k chybám. Takovou změnu může přinést jiná verze jazyka Java, tedy nástroje Javadoc, nebo také uživatelská přizpůsobení. Nejedná se tedy o příliš spolehlivé řešení.

3.3.4 Javadoc doclet

Doclet je program, který specifikuje obsah a formát výstupu nástroje Javadoc. Výchozí nastavení nástroje Javadoc využívá HTML doclet. Prostřednictvím konfigurace lze poskytnout jiný doclet, který může zajistit výstup v jiném formátu a s jiným obsahem. Doclet API poskytuje strukturovaný model kódu včetně komentářů. Výpis kódu 3.6 znázorňuje vypsání všech názvů elementů, které byly specifikovány při spuštění nástroje[6].

Javadoc stejně jako JavaParser provádí statickou analýzu kódu, při které nedochází k jeho spuštění. Bez spuštění Spring kontextu nebude možné zpracovávanou konfiguraci validovat, nebo vyhodnocovat SpEL výrazy, avšak to není účelem. Hlavním cílem je vygenerovat dokumentaci. Mezi výhody použití docletu patří dostupnost Javadoc nástroje, který je součástí JDK, snadná implementace podpořená oficiální dokumentací i komunitou. Načtení zdrojových souborů a reprezentace kódu je zajištěna Javadoc nástrojem. Integrace docletu je možná prakticky kdekoli, kde je dostupný nástroj Javadoc [14].

```
public class BasicDoclet implements Doclet {
    @Override
    public void init(Locale locale, Reporter reporter) {  }

    @Override
    public String getName() {
        return getClass().getSimpleName();
    }

    @Override
    public Set<? extends Option> getSupportedOptions() {
        return Collections.emptySet();
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latest();
    }

    @Override
    public boolean run(DocletEnvironment environment) {
        environment.getSpecifiedElements().forEach(System.out::println);
        return true;
    }
}
```

Výpis kódu 3.6. Ukázka minimální Javadoc doclet implementace

[5] jsoup.org [vid. 2023-11-04]
[6] Použito aktuální standardní doclet API (od verze Java 9).

3.3.5 Anotační procesor

Anotační procesory byly představeny v kapitole 2.2.1. Zatímco doclet je přímou cestou při zpracovávání Javadoc komentářů, anotační procesor je nástroj pro zpracovávání anotovaných elementů. V případě této práce jde primárně o anotace `@ConfigurationProperties` a `@Value`. Protože je rozhraní anotačních procesorů součástí standardního jazyka Java, je možné procesor využít kdekoli, kde je využit Java kompilátor. V případě příkazové řádky se procesory specifikují prostřednictvím parametrů nástroje `javac` (Java Compiler). V případě Maven projektů, kde je ke kompilaci používán Maven Compiler plugin, je také možné specifikovat anotační procesory prostřednictvím konfigurace pluginu. Zároveň není problém využít procesor v projektech využívajících jiné ekosystémy (například Gradle). Rozhraní kompilátoru umožňuje anotačnímu procesoru přístup k původnímu kódu včetně Javadoc komentářů.

3.4 Shrnutí

V tabulce 3.1 je znázorněno porovnání zvažovaných struktur aplikace, konkrétně tedy náročnost jejich implementace a zapojení do existujícího projektu – náročnost instalace, konfigurace a vyžadovaných změn v projektu. Po porovnání těchto dvou možností lze dojít k závěru, že volba struktury Maven pluginu by byla snazší na implementaci a následné použití v rámci Maven projektů. Možné přístupy ke zpracování zdrojového kódu jsou porovnány v tabulce 3.2. Nejlepšími přístupy se zde jeví využití Javadoc docletu nebo anotačního procesoru.

Obě řešení jsou součástí standardního jazyka, využívají stejnou API a tedy náročnost jejich implementace je téměř totožná. Je možné je integrovat do čistého Java projektu pouze za využití nástroje Javadoc, respektive Java kompilátoru (javac) a tedy i do projektů využívajících libovolné ekosystémy (Maven, Gradle). Hlavním rozdílem je, že nástroj Javadoc zpracuje vždy původní kód, přičemž některé informace vynechá, například výchozí hodnoty atributů – blíže bude popsáno v kapitole 4.5. Oproti tomu anotační procesor má k dispozici kompletní původní kód a zároveň dokáže reagovat na změny z ostatních anotačních procesorů. Vhodným řešením tedy bude implementace anotačního procesoru.

Náročnost:	Implementace	Integrace do existujícího projektu
Samostatná aplikace	náročná	náročná
Maven plugin	středně obtížná	snadná

Tabulka 3.1. Porovnání struktur aplikace

	Náročnost implementace	Vyžaduje změny v existujícím projektu
Runtime reflexe	náročná	ano
Zpracování HTML	náročná	ne
JavaParser	středně obtížná	ne
Javadoc doclet	snadná	ne
Annotation processor	snadná	ne

Tabulka 3.2. Porovnání, jak je možné zpracovat zdrojový kód

Kapitola 4
Návrh řešení

Z předchozí kapitoly vyplývá, že řešením bude anotační procesor, který bude možné integrovat do již existujících projektů díky přímé podpoře Java kompilátorem. Pro ilustraci bude využívána zkrácená konfigurace projektu TermIt[1] [19]. Vstupem pro anotační procesor tedy bude konfigurační třída zobrazená ve výpisu kódu 4.1, očekávaným výstupem je relevantní dokumentace odpovídající obrázku 4.1. V této kapitole bude blíže upřesněno a popsáno, jakým způsobem bude anotační procesor fungovat, jaké nabídne funkce, možnosti konfigurace, formáty a obsah výstupu.

Variable	Description
`TERMIT_URL`	TermIt frontend URL. It is used, for example, for links in emails sent to users.
`TERMIT_REPOSITORY_URL` *	URL of the main application repository.
`TERMIT_REPOSITORY_PUBLICURL`	Public URL of the main application repository. Can be used to provide read-only no authorization access to the underlying data.

Obrázek 4.1. Očekávaný formát dokumentace ke konfiguraci 4.1

4.1 Význam proměnných prostředí

Dokumentace pro konfiguraci bude v tomto případě generována ve formátu proměnných prostředí (environmental variables). Jak bylo již dříve zmíněno, proměnné prostředí se zapisují jako řetězce alfanumerických znaků a znaku podtržítka oddělené rovnítkem od jejich hodnoty, např. `TERMIT_URL=http://localhost:3000` [10]. Proměnné prostředí dostávají zvláštní význam při kontejnerizaci aplikace.

Kontejnerizace může být často zaměňována s virtualizací operačních systémů, ovšem tyto dva přístupy se značně liší. Zatímco virtualizace umožňuje souběžné spuštění více operačních systémů na jednom fyzickém zařízení, kontejnerizace izoluje jednotlivé aplikace v rámci jednoho operačního systému. Součástí kontejneru jsou veškeré závislosti a konfigurace, které aplikace potřebuje pro svůj běh. Navíc kontejner je standardně definován vývojáři konkrétní aplikace, což výrazně zjednodušuje nasazení. Kontejner je tedy zcela samostatná jednotka, je samostatně spustitelný a snadno přenositelný mezi různými prostředími [26].

Aplikace obvykle vyžaduje nějakou konfiguraci, aby byla přizpůsobena provozu pro konkrétní účel a nasazení. Konfigurace se v případě kontejnerů často definuje právě prostřednictvím proměnných prostředí.

[1] https://github.com/kbss-cvut/termit/blob/master/src/main/java/cz/cvut/kbss/termit/util/Configuration.java [vid. 2024-03-13]

```java
@Configuration
@ConfigurationProperties(prefix = "termit")
@Validated
public class TermitConfiguration {
    /**
     * TermIt frontend URL.
     * <p>
     * It is used, for example, for links in emails sent to users.
     */
    private String url = "https://localhost:3000/";

    @Valid
    private Repository repository = new Repository();

    public static class Repository {
        /**
         * URL of the main application repository.
         */
        @NotNull
        String url;

        /**
         * Public URL of the main application repository.
         * <p>
         * Can be used to provide read-only no authorization
         * access to the underlying data.
         */
        Optional<String> publicUrl = Optional.empty();

        // getter & setter
    }

    // getter & setter
}
```

Výpis kódu 4.1. Příklad Spring Boot konfigurační třídy obsahující Javadoc komentáře, převzato z [19]

4.2 Spouštění a integrace do projektu

Protože výsledný nástroj bude implementován jako anotační procesor, bude přímo podporován Java kompilátorem. A tedy není nijak vázán na ekosystém Maven. U přímého použití nástroje javac v příkazové řádce, je možné anotační procesor specifikovat prostřednictvím parametrů `-processor`, `--processor-module-path`, `--processor-path` [27]. V případě Mavenu je možné anotační procesor specifikovat v rámci konfigurace Maven Compiler pluginu, ukázka takové konfigurace se nachází ve výpisu kódu 4.2. Zároveň je v ukázce znázorněno, jakým způsobem je možné definovat konfiguraci, která je kompilátorem předána anotačnímu procesoru.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.11.0</version>
    <configuration>
        <annotationProcessorPaths>
            <path>
                <groupId>cz.lukaskabc.cvut.processor</groupId>
                <artifactId>
                    spring-boot-configuration-docgen-processor
                </artifactId>
                <version>0.0.1</version>
            </path>
        </annotationProcessorPaths>
        <compilerArgs>
            <arg>-Aconfigurationdoc.prepend_required</arg>
            <arg>-Aconfigurationdoc.deprecated_last</arg>
            <arg>-Aconfigurationdoc.order=asc</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

Výpis kódu 4.2. Ukázka konfigurace Maven Compiler pluginu se specifikací anotačního procesoru a jeho konfigurace

4.3 Obsah výstupu

Jak bylo zmíněno v kapitole 4.1, názvy jednotlivých konfigurovatelných hodnot ve výstupní dokumentaci budou v podobě proměnných prostředí, které budou doplněny o jejich popis. Nabízí se vytvoření tabulky se dvěma sloupci. První sloupec bude obsahovat název proměnné prostředí a případná důležitá upozornění, například že se jedná o nezbytnou, nebo „deprecated“ hodnotu. Druhý sloupec bude obsahovat popis dané hodnoty a případná omezení. Je třeba definovat, jaké funkce budou při generování dokumentace podporovány – to zahrnuje části Javadoc komentářů (syntaxe, tagy), JSR 303 anotace a samozřejmě strukturu Spring konfigurace a její anotace.

Název proměnné prostředí je dán buď názvem konfigurační hodnoty uvnitř anotace `@Value`[2], nebo v případě konfiguračních tříd její polohou ve třídě. V druhém případě se začíná u třídy s anotací `@ConfigurationProperties`[3], její název dává začátek názvu proměnné prostředí (případně je využit prefix, pokud je definován). Následuje podtržítko „_“ a název atributu. Pokud má atribut složený datový typ (jedná se o další vnořenou strukturu), pokračuje se stejně – tedy je opět přidáno podtržítko, název atributu atd., dokud se nedojde k atributu jednoduchého datového typu[4]. Z příkladu konfigurace 4.1 lze sestavit název proměnné prostředí pro atribut `String url` uvnitř vnořené třídy Repository.

[2] `org.springframework.beans.factory.annotation.Value`
[3] `org.springframework.boot.context.properties.ConfigurationProperties`
[4] Výjimkou jsou atributy, které mají typ seznamu, mapy, nebo jiného podporovaného datového typu, který umí Spring sestavit z konfiguračních možností.

1. `TERMIT` – Třída s anotací `@ConfigurationPriperties` má název „TermitConfiguration“, v tomto případě je ovšem přítomen parametr „prefix“ s hodnotou „termit“, která je upřednostněna.
2. `TERMIT_REPOSITORY` – Atribut `Repository repository` má název „repository“ (na názvu třídy, tedy datového typu nezáleží).
3. `TERMIT_REPOSITORY_URL` – Datový typ Repository je složený datový typ definovaný vnořenou třídou, její atribut `String url` má název „url“. Zároveň se jedná o jednoduchý datový typ, název proměnné je tedy konečný.

Anotační procesor kromě běžného zdrojového kódu také dokáže zpracovat Javadoc komentáře. I Javadoc komentáře jsou kompilátorem reprezentovány jako struktura a je tedy možné je dále zpracovávat bez nutnosti práce s textem jako takovým. Anotační procesor bude do výsledné dokumentace zahrnovat následující Javadoc tagy ({složené závorky značí inline tag}):

- `{@code text}` – Zformátuje text jako kód.
- `@deprecated` – Přidá upozornění k hodnotě, že by neměla být dále používána.
- `{@literal text}` – Nahradí „<>“ za HTML znaky, aby text nebyl interpretován jako HTML (stejné jako `@code`, ale bez formátování).
- `{@link}` a `{@linkplain}` – Do dokumentace začlení pouze label tohoto tagu („plain“ varianta přistoupí k textu stejně jako tag `@literal`).
- `@param` – Specifikuje dokumentaci ke konkrétnímu parametru metody nebo konstruktoru.
- `@see text/url` – Přidá „See also“ a připojí zadaný text.
- `@since text` – Přidá „Since“ a připojí zadaný text.
- `@hidden` – Zcela skryje hodnotu z dokumentace.

Zpracování tagu `@literal` bude lehce pozměněno, aby byla zajištěna kompatibilita s výstupním formátem. V případě Markdown formátu tedy budou zpracovávány znaky, které jsou využívány k formátování, konkrétně tedy budou zneplatněny tak, že se před ně přidá zpětné lomítko. Obsah tagu `@param` bude ve výstupu využit za podmínky, že je tag přítomen na konstruktoru, jehož odpovídající parametr je spárován s atributem třídy, který je součástí konfigurace. Parametr konstruktoru je spárován, pokud se shoduje jeho název a datový typ s nějakým atributem třídy. Tagy pro odkazy (`@link` a `@see`) budou do dokumentace promítnuty jako pouhý text. V původní Javadoc dokumentaci představují odkazy na jiné části kódu, resp. programátorské dokumentace, která v tomto případě přítomna není a tedy tyto odkazy zde nedávají smysl. Pro jejich zkonstruování by bylo nutné umožnit definovat odkaz/umístění vygenerované Javadoc dokumentace. Stejné funkcionality lze docílit použitím přímého odkazu již v samotném Javadoc komentáři. Ostatní Javadoc tagy[5] nebudou do výstupu zařazeny.

Další podporovanou funkcí bude popis JSR 303 anotací. Tabulka 4.1 představuje seznam aktuálních anotací a jejich vysvětlení [13]. Pokud bude konkrétní anotace použita u některé z konfiguračních hodnot, bude její popis součástí dokumentace. Všechny anotace, kromě `@NotNull`, `@NotBlank` a `@NotEmpty` akceptují prázdnou hodnotu (`null`) jako validní. Dokumentace bude generována v anglickém jazyce. Obrázek 4.2 znázorňuje očekávaný vzhled tabulky se zvýrazněnými elementy.

[5] `author`, `docRoot`, `exception`, `inheritDoc`, `return`, `serial`, `serialData`, `serialField`, `throws`, `value`, `version`

Anotace	Obsah dokumentace
`AssertFalse`	has to be `false`
`AssertTrue`	has to be `true`
`DecimalMax, Max`	value <= `{123}`
`DecimalMin, Min`	`{123}` <= value
`Digits`	maximum of `{123}` digits before the decimal point and `{123}` digits after it
`Email`	valid email
`Future`	time in the future
`FutureOrPresent`	time in the present or in the future
`Negative`	value < 0
`NegativeOrZero`	value <= 0
`NotBlank`	value must be present and contain at least one non-whitespace character
`NotEmpty`	value must be present and not empty
`NotNull`	value must be present
`Null`	no value accepted, leave blank
`Past`	time in the past
`PastOrPresent`	time in the past or present
`Pattern`	value must match regular expression `{expression}`
`Positive`	0 < value
`PositiveOrZero`	0 <= value
`Size`	`{123}` <= value length/size <= `{123}`

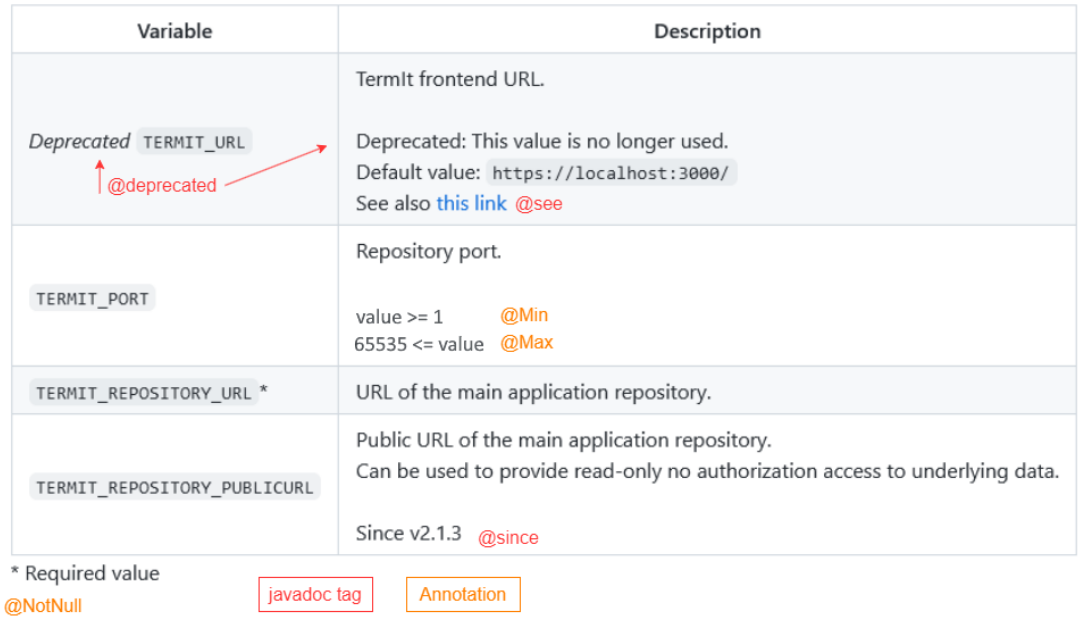
Tabulka 4.1. Dokumentace JSR 303 anotací [13], hodnota v závorkách `{123}` představuje zástupce za číselný parametr anotace

4.4 Šablony dokumentace

Tabulka samozřejmě není jediným možným řešením vzhledu dokumentace a nemusí být vhodná pro všechny případy. Proto bude anotační procesor výstupní dokumentaci generovat pomocí template enginu. Ve výchozím nastavení budou k dispozici interní šablony, které budou generovat dokumentaci v podobě popsané tabulky. Zároveň bude uživateli umožněno nastavit cestu k jeho vlastní šabloně. Existuje řada různých template enginů, které se nejčastěji liší syntaxí a podporovanými funkcemi. Pro účely této práce je klíčové, aby template engine dokázal zpracovávat, respektive generovat formáty HTML, Markdown a aby podporoval syntaxi pro zpracování seznamu objektů (iteraci).

V této práci bude využit template engine FreeMarker [17]. Také byly uváženy template enginy Thymeleaf[6] a Velocity[7], ovšem v jejich případě se jedná o rozsáhlé, komplexní knihovny, které jsou primárně určeny pro generování HTML dokumentů. Dalšími vhodnými kandidáty byli: Handlebars[8], což je template engine používaný projektem Spring Configuration Property Documenter, který byl popsán v kapitole 3.1.3;

[6] https://www.thymeleaf.org/ [vid. 2024-03-13]
[7] https://velocity.apache.org/ [vid. 2024-03-13]
[8] https://jknack.github.io/handlebars.java/ [vid. 2024-03-13]



Obrázek 4.2. Ukázka vzhledu Markdown tabulky s popisem konfigurace

JTE: Java Template Engine[9], Pebble[10] a již zmíněný FreeMarker. Výsledná volba pro FreeMarker byla převážně subjektivní preferencí vzhledu syntaxe, API a dokumentace.

4.4.1 Apache FreeMarker

Jedná se o template engine v podobě Java knihovny. Umožňuje generovat textový výstup na základě definovaných šablon a vstupních dat. Pro vytváření šablon využívá vlastní specializovaný jazyk FreeMarker Template Language (FTL). Výhodou je, že není vázán na konkrétní formát či aplikační strukturu.

Syntaxe FTL obsahuje tři základní typy elementů: nahrazení hodnotou z datového modelu, FTL tagy a komentáře. Syntaxe `${...}` značí přístup ke konkrétní hodnotě ze vstupních dat, tato značka tedy bude nahrazena odpovídající hodnotou. FTL tagy ve výchozím nastavení využívají syntaxi podobnou HTML, tedy `<#if>...</#if>`. Knihovna ovšem nabízí možnost změny syntaxe, čehož bude v této práci využito, aby byly značky template enginu rozdílné od běžného HTML. Bude tedy využita syntaxe hranatých závorek `[#if]...[/#if]`. Komentáře jsou opět podobné HTML `<#-- ... -->`, respektive `[#-- ... --]` [17].

Výpis kódu 4.3 znázorňuje použití šablony s výpisem prvků pole `animals` společně s podmíněným použitím HTML třídy `class="protected"`. Výpis kódu 4.4 znázorňuje zápis stejné šablony, ovšem s použitím syntaxe hranatých závorek. Jednou z výhod FreeMarkeru je podpora maker. Makra umožňují vytvářet vlastní funkce, které lze následně využít v šablonách. Tímto způsobem je možné uživateli, který bude vytvářet vlastní šablonu, předpřipravit sadu funkcí pro usnadnění formátování a práce s dokumentací.

[9] https://jte.gg/ [vid. 2024-03-13]
[10] https://pebbletemplates.io/ [vid. 2024-03-13]

```
<#list animals as animal>
    <div<#if animal.protected> class="protected"</#if>>
        ${animal.name} for ${animal.price} Euros
    </div>
</#list>
```

Výpis kódu 4.3. Ukázka FreeMarker šablony, převzato z [17]

```
[#list animals as animal]
    <div[#if animal.protected] class="protected"[/#if]>
        ${animal.name} for ${animal.price} Euros
    </div>
[/#list]
```

Výpis kódu 4.4. Ukázka FreeMarker šablony s využitím syntaxe hranatých závorek

4.5 Výchozí hodnoty

Konfigurace může obsahovat předdefinované výchozí hodnoty, které jsou použity v případě, že uživatel nespecifikuje vlastní hodnoty. Ty lze v kódu zapsat četnými způsoby.

- `private String url = "https://localhost:3000/";` – přímou deklarací hodnoty
- `private String url = DEFAULT_URL;` – odkázáním na konstantu

Hodnota může být přímo deklarována, může být odkázáno na jinou proměnnou, či statickou konstantu, která může být ze stejné třídy, nebo z jiné. V případě jiné třídy může být identifikována názvem třídy `DefaultValues.URL`, nebo může být staticky importována. Hodnota může být také výsledkem volání metody. Způsobů zápisu a členění kódu je mnoho. Protože Spring Boot umožňuje načítání hodnot z různých zdrojů, mají tyto zdroje různou prioritu. Hodnoty z nadřazených zdrojů jsou upřednostněny a tedy i výchozí hodnoty mohou být definovány v nějakém zdroji, který má nižší prioritu. Příkladem může být zapsání výchozích hodnot do souboru `application.properties`, který je součástí zdrojů aplikace. Následně jsou hodnoty definované v proměnných prostředí upřednostněny před hodnotami ve výchozím souboru.

Tyto různé zdroje a způsoby definice výchozích hodnot je třeba vzít v úvahu při generování dokumentace. Anotační procesor má přístup k původnímu kódu a je tedy možné číst, zpracovávat a do jisté míry určit výchozí hodnoty definované přímo v kódu. Hledání a zpracovávání dalších zdrojů hodnot by bylo složité a v případě změn v knihovně Spring Boot by mohlo přinést nekompatibilitu či nekonzistentní výstupy. Zároveň lze předpokládat, že v některých případech zvláštního zápisu zdrojového kódu nemusí ani implementovaný algoritmus pro hledání výchozích hodnot fungovat správně. Programátor může výchozí hodnoty atributu přiřazovat v těle metod, konstruktorů, či jinými způsoby, které výsledný nástroj nebude podporovat. Proto anotační procesor zavede možnost specifikovat výchozí hodnotu přímo v dokumentaci. Toho bude docíleno vytvořením vlastního Javadoc tagu `@configurationdoc.default`. Doporučením pro vytváření vlastních Javadoc tagů je, aby obsahovaly tečku, čímž bude zaručena budoucí kompatibilita, protože standardní Javadoc tagy nikdy nebudou tečku obsahovat [14]. Javadoc tag bude tedy tvořen prefixem „`@configurationdoc`“ a následně názvem tagu „`default`“. Pokud bude tento tag použit v Javadoc komentáři konfiguračního atributu, bude jeho hodnota použita v dokumentaci jako informace o výchozí hodnotě.

Zde se nachází největší rozdíl mezi možnostmi anotačního procesoru a Javadoc docletu. Doclet pracuje s informacemi, které předtím zpracoval nástroj Javadoc.

Během tohoto zpracování jsou ponechány pouze informace o základních definicích elementů – třídy, atributy, metody, konstruktory. Další informace jsou zahozeny, včetně informací o výchozích hodnotách. Jsou ponechány pouze hodnoty statických konstant, tedy atributů definovaných jako `static final`. V případě docletu by tedy bylo prakticky nezbytné použití Javadoc tagu pro začlenění výchozích hodnot do dokumentace. Oproti tomu anotační procesor má přístup ke kompletnímu modelu původního kódu tak, jak jej načte kompilátor, a tedy může přistupovat k libovolným informacím včetně všech hodnot. Použití Javadoc tagu je zde pouze ponecháno pro zvláštní případy, kdy anotační procesor nedokáže najít správnou výchozí hodnotu.

4.6 Proces generování dokumentace

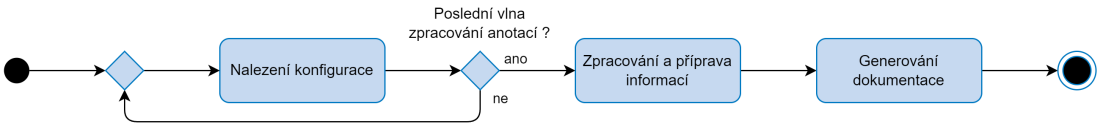
Kompilátor spouští anotační procesory v několika opakujících se vlnách, kdy každý anotační procesor obdrží zpracovávaný kód a může nad ním provést nějaké operace, např. vygenerování dalších zdrojových souborů. Tento proces je opakován, dokud jsou generovány další zdrojové soubory. Protože účelem anotačního procesoru v rámci této práce je generování dokumentace, bude tento procesor pro většinu vln pasivní. Při každé vlně, kdy bude zavolán, posbírá informace o elementech s anotacemi `@ConfigurationProperties` a `@Value`, tyto informace si uloží a vyčká na další vlnu. Poslední vlna je kompilátorem zvlášť označena a umožňuje tak anotačním procesorům oznámit konec zpracovávání. Toho bude využito pro zpracování posbíraných informací a vygenerování výsledné dokumentace. Tím bude zajištěna kompatibilita s ostatními anotačními procesory, protože veškeré zpracovávání bude provedeno až na konci, tedy po tom, co ostatní anotační procesory dokončí svou práci. Samozřejmě pokud jiný anotační procesor bude ke zpracovávání přistupovat stejně a tedy bude vyčkávat až na poslední vlnu, pak nelze kompatibilitu zcela zaručit a bude záležet na konkrétním pořadí, ve kterém kompilátor anotační procesory spustí.

Příkladem, proč je nutné čekat až na poslední vlnu zpracování, může být kompatibilita s knihovnou Lombok [28]. Lombok umožňuje na základě anotací generovat kód. Anotace `@Getter` umožňuje pro anotovaný atribut automaticky vygenerovat getter. Obdobně je možné generovat setter, konstruktory a další části kódu. Tyto části kódu jsou klíčové při rozhodování, zda je daný atribut možné použít ke konfiguraci. Pokud by anotační procesor nevyčkával na ostatní, musel by obsahovat vlastní logiku pro detekování konkrétních anotací. Vyčkáním na ostatní anotační procesory je zajištěno, že budou vygenerovány reálné metody, které mohou být zpracovány již běžným způsobem.

Anotační procesor bude mít tři hlavní fáze:

1. nalezení konfigurace,
2. zpracování a příprava informací,
3. generování dokumentace.

Proces je také znázorněn na obrázku 4.3. Všechny fáze anotačního procesoru probíhají v kompilační fázi „Zpracování anotací“ (viz obrázek 2.1). Během každé vlny anotační procesor obdrží elementy se specifikovanými anotacemi, tyto elementy si pouze uloží pro pozdější zpracování (fáze anotačního procesoru „nalezení konfigurace“). Během poslední vlny zpracovávání anotací proběhnou následující fáze anotačního procesoru. Druhá fáze anotačního procesoru („zpracování a příprava informací“) zahrnuje analýzu anotovaných elementů, například nalezení atributů konfiguračních tříd, a následnou transformaci posbíraných informací do vhodné podoby pro generování dokumentace. Ve třetí fázi („generování dokumentace“) anotační procesor již vygeneruje dokumentaci a zapíše ji do souboru.



Obrázek 4.3. Fáze zpracování informací a generování dokumentace anotačním procesorem

Kapitola 5
Implementace

V této kapitole bude popsána implementace a rozhraní, které využívá. Bude rozebrán návrh jednotlivých komponent programu, jejich vztahů a závislostí. Součástí bude také popis problémů, které se během implementace vyskytly a jejich řešení.

Struktura anotačního procesoru je znázorněna diagramem tříd na obrázku 5.1. Diagram také znázorňuje vytváření instancí tříd pomocí vazeb „instantiate“. Pro lepší přehlednost byly uvedeny pouze některé třídy a jejich metody.

Rámečky ohraničující některé třídy a rozhraní značí jejich umístění v balíčcích v rámci API jazyka. `ConfigurationDocProcessor` je hlavní třídou implementovaného anotačního procesoru (dále jako dokumentační procesor). Třídy `PropertiesClassScanner` a `ValueAnnotationScanner` vyhledávají a extrahují informace o konfigurovatelných parametrech z elementů s anotací `@ConfigurationProperties`, respektive `@Value`. `DefaultValueCollector` zajišťuje sběr výchozích hodnot parametrů. Třídy označené jako „Descriptor“ slouží pro rozhodování, zda je možné na daný atribut mapovat konfiguraci – blíže jsou popsány v kapitole 5.4. `ElementDecorator` reprezentuje konfigurovatelný parametr, který je dále zpracováván třídou `ElementDecoratorDocGenerator`. Tato třída zapouzdří element do `DocumentedElement`, což umožňuje k elementu přidat dokumentaci a pohodlně jej využít v šablonách. Třídy `JavadocTreeVisitor`, `JSR303DocsGenerator` a `JavadocDefaultTagVisitor` slouží pro zpracování Javadoc komentářů, respektive JSR 303 anotací a vlastního `@configurationdoc.default` Javadoc tagu. Třídy označené jako „Formatter“ poskytují syntaxi pro HTML a Markdown formát. Nakonec třída `TemplateDocsGenerator` umožňuje vygenerovat dokumentaci na základě šablony, blíže bude popsáno v kapitole 5.7.

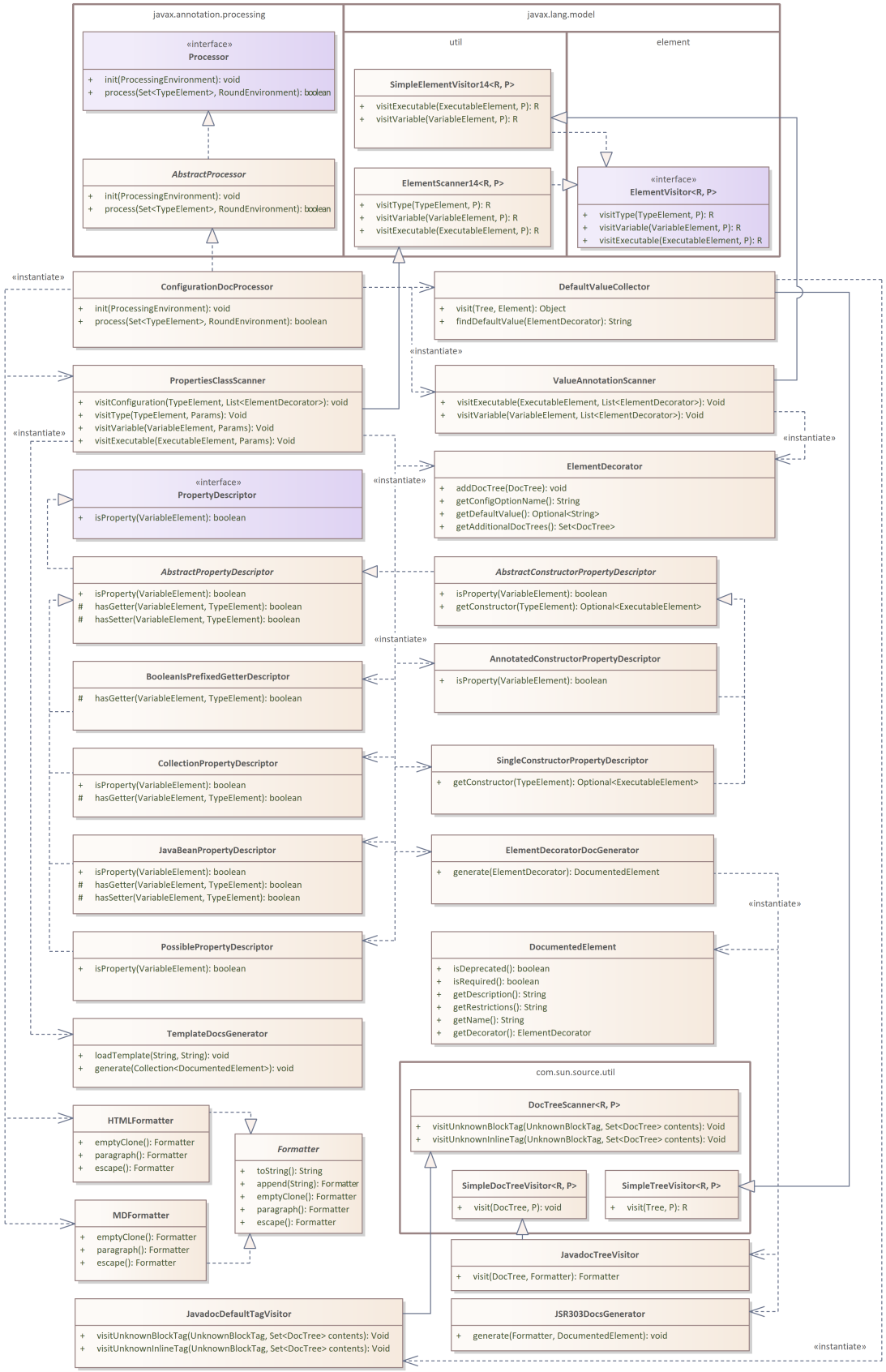
5.1 Verze jazyka Java

Důležitým rozhodnutím před začátkem implementace bylo, jakou verzi jazyka Java zvolit. Z dlouhodobého hlediska je vhodné zvolit verzi s dlouhým obdobím podpory (LTS – Long Term Support). Následuje seznam aktuálně dostupných LTS verzí jazyka Java a jejich datum konce podpory [29]:

- 8 LTS – prosinec 2030,
- 11 LTS – září 2026,
- 17 LTS – září 2029,
- 21 LTS – září 2031.

Java 8 je dle průzkumů JetBrains stále nejpoužívanější verzí [5] a zároveň má nejdelší období podpory mezi nejpoužívanějšími verzemi. Tato verze by byla vhodnou volbou pro zajištění maximální kompatibility s existujícími projekty.

Významným aspektem, který je úzce spjatý s volbou verze jazyka, je podpora verzí knihovny Spring Boot a Spring frameworku. Spring Boot od verze 3 vyžaduje minimálně verzi Java 17. Knihovna Spring Boot zároveň definuje verzi používaného Spring frameworku. Pro Spring Boot 3 se jedná o Spring framework 6. Pro implementaci bude použit Spring Boot 3 a tedy Java 17 a Spring framework 6 [7–9].



Obrázek 5.1. UML diagram tříd anotačního procesoru obohacený o instanční vazby, pro lepší přehlednost byly některé metody a třídy vynechány

5.2 Compiler API

Anotační procesory využívají rozhraní kompilátoru, které umožňuje reprezentovat zpracovávaný zdrojový kód a manipulovat s ním. Toto rozhraní se nachází v modulu `jdk.compiler` a dělí se na dvě hlavní části.

- „Language Model API“[1] – umožňuje modelování a reprezentaci elementů jazyka, jako jsou moduly, balíčky, třídy, rozhraní, konstruktory, metody a atributy ve stromové struktuře. Tedy i jejich datové typy (opět se jedná o třídy) a vztahy dědičnosti mezi nimi.
- „Compiler Tree API“[2] – poskytuje rozhraní pro reprezentaci zdrojového kódu jako stromové struktury (Abstract Syntax Tree – AST).

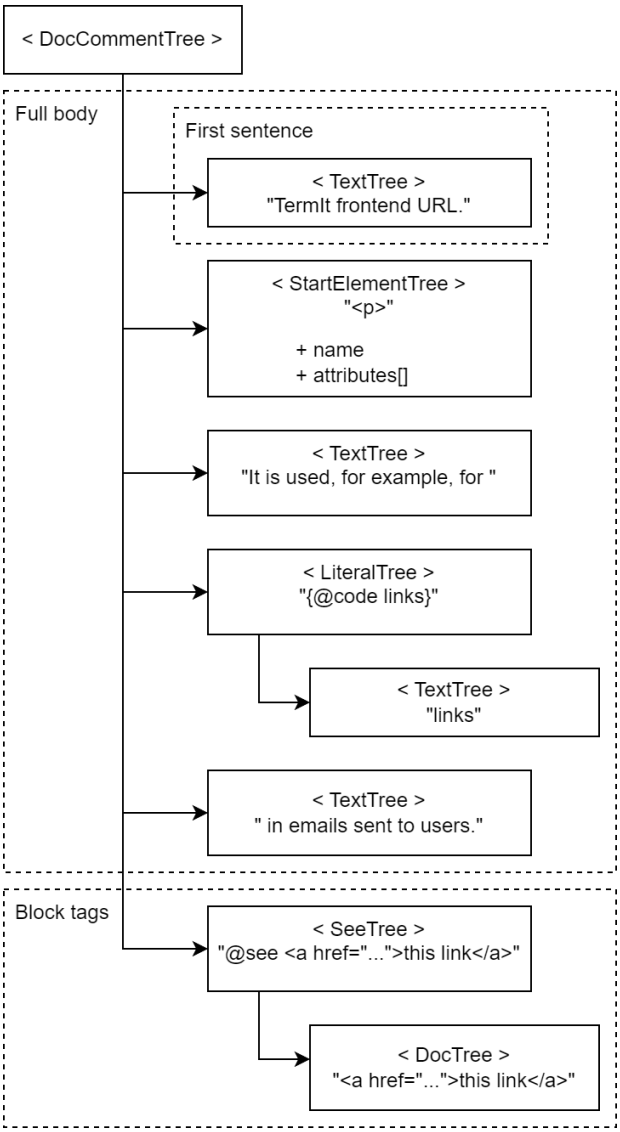
Na rozdíl od „Language Model API“, které řeší datové typy a jejich strukturu, „Compiler Tree API“ již zahrnuje kompletní reprezentaci zdrojového kódu. Reprezentuje například importy, těla metod, ale také Javadoc komentáře [30]. Výpis kódu 5.1 obsahuje příklad Javadoc komentáře, jehož reprezentace stromovou strukturou je znázorněna na obrázku 5.2. Obě rozhraní jsou postavena na návrhovém vzoru Visitor, díky kterému je možné snadno procházet reprezentované stromové struktury [31]. Základními třídami rozhraní kompilátoru jsou:

- `Element`[3] – Představuje prvek kódu, může se jednat o třídu, metodu, atribut a další struktury jazyka.
- `TypeMirror`[4] – Reprezentuje datový typ, tím může být deklarovaný typ (třída, nebo rozhraní), primitivní datový typ, ale také pole nebo null.
- `AnnotationValue`[5] – Drží hodnotu parametru anotace, může obsahovat text, číslo, jiný datový typ (reprezentován třídou `TypeMirror`), jinou anotaci, nebo také seznam hodnot (v případě, že jich je víc).
- `Tree`[6] – Stromová struktura představující kód[7] (AST).
- `DocTree`[8] – Stromová struktura představující Javadoc dokumentaci kódu.

```
/**
 * TermIt frontend URL.
 * <p>
 * It is used, for example, for {@code links} in emails sent to users.
 * @see <a href="...">this link</a>
 */
```

Výpis kódu 5.1. Příklad Javadoc komentáře, stromová struktura je ilustrována na obrázku 5.2, konkrétní odkaz byl vynechán pro lepší čitelnost

[1] `java.compiler`
[2] `com.sun.source.tree`, `com.sun.source.doctree`
[3] `javax.lang.model.element.Element`
[4] `javax.lang.model.type.TypeMirror`
[5] `javax.lang.model.element.AnnotationValue`
[6] `com.sun.source.tree.Tree`
[7] Tato rozhraní jsou implementována JDK kompilátorem a jsou předmětem změn při vývoji jazyka.
[8] `com.sun.source.doctree`



Obrázek 5.2. Ilustrace stromové struktury Javadoc komentáře pro výpis kódu 5.1

5.3 Fáze anotačního procesoru

Základem a zároveň jedinou nezbytnou částí anotačního procesoru je třída implementující rozhraní `Processor`[9]. Prostřednictvím této třídy může kompilátor anotační procesor spustit a získat o něm potřebné informace. Za hlavní lze považovat metody `init` a `process`. Metoda `init(ProcessingEnvironment)` je volána při inicializaci anotačního procesoru a předává mu prostředí, ve kterém bude spuštěn. Metoda `process(Set<? extends TypeElement>, RoundEnvironment)` je volána při každé vlně zpracovávání anotací, pomocí parametrů kompilátor předává anotace, které mají být v této vlně zpracovány a předávají také kontext, ve kterém se aktuální vlna zpracovávání odehrává. Prostřednictvím objektů `ProcessingEnvironment` a `RoundEnvironment` může anotační procesor získat instance nástrojů pro práci s objekty rozhraní kompilátoru. Tyto nástroje umožní například „převod“ objektu `Element` na odpovídající `Tree` [30].

[9] `javax.annotation.processing.Processor`

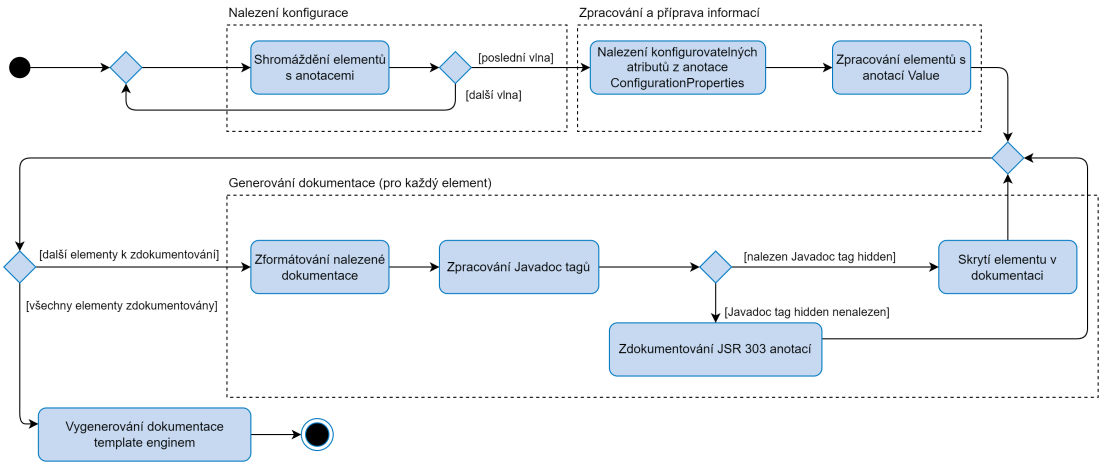
V dokumentačním procesoru je hlavní třídou `ConfigurationDocProcessor`, viz diagram tříd 5.1, která rozšiřuje abstraktní třídu `AbstractProcessor`[10], která implementuje rozhraní `Processor`. Tato třída řídí celý běh anotačního procesoru, přičemž využívá ostatní třídy. Pomocí třídy `AbstractProcessor` je možné popsat procesor, namísto implementování vlastních metod prostřednictvím anotací.

Proces zpracování anotací znázorněný na obrázku 4.3 je podrobněji zobrazen na obrázku 5.3, následující text pak přidává bližší popis jednotlivých fází. Během první fáze bude prakticky kompilátor pouze volat metodu `process` anotačního procesoru. V této metodě procesor vždy projde elementy poskytnuté ke zpracování, rozdělí je dle jejich anotace (`@ConfigurationProperties` a `@Value`) a uloží k nim cesty v AST. Je podstatné, aby byly uloženy „pouze“ cesty k elementům, nikoli objekty, které reprezentují dané elementy, či dokonce jejich kód (Tree). Kompilátor nemusí garantovat, že poskytnuté elementy zůstanou validní pro další vlny – uložením jejich reference by pak v jiné vlně mohlo vyvolat přístup k již nevalidním datům. Obdobně lze problém přirovnat k přístupu k souboru, pokud by jeho obsah byl načten a následně původní obsah změněn jiným procesem (jiným anotačním procesorem). Přistoupením k načtenému obsahu již dochází k přístupu k neaktuálním datům. Uložením cesty k souboru a jeho načtení až ve chvíli, kdy je opravdu jeho obsah potřeba, zaručí přístup k aktuálním datům.

Během každého volání metody `process` musí procesor kontrolovat, zda se již nejedná o poslední vlnu (tato informace je dostupná z objektu `RoundEnvironment`). V případě, že ano, zahájí procesor další zpracovávání shromážděných elementů z předchozích vln. K anotovaným elementům je třeba přistupovat různými způsoby v závislosti na jejich typu (třída, metoda, atribut atd.) a samozřejmě použité anotaci. U elementů s anotací `@ConfigurationProperties` je třeba projít strukturu jejich datového typu a rekurzivně získat všechny atributy, na které by mohla být mapována konfigurace. Zpracovávání jednotlivých atributů je dále popsáno v kapitole 5.4. Při zanořování do struktury konfigurace je třeba průběžně sestavovat název proměnné prostředí, jak bylo popsáno v kapitole 4.3, a sbírat z navštívených elementů dokumentaci. Blíže bude popsáno v kapitole 5.5. Anotace `@Value` neumožňuje mapování vlastních struktur, v jejím případě tedy stačí získat název konfigurační hodnoty z anotace.

Při nalezení všech konfigurovatelných hodnot a jejich názvů, může dokumentační procesor přejít k vygenerování samotné dokumentace. Zde je využita sesbíraná dokumentace z navštívených elementů, respektive z jejich komentářů, ty je zformátovány a jsou z nich zpracovány Javadoc tagy. Nakonec je k dokumentaci připojen popis omezení definovaných JSR 303 anotacemi. Takto zdokumentované elementy jsou předány template enginu, který pomocí uživatelsky definované šablony, nebo výchozí interní šablony, vygeneruje výslednou dokumentaci. Použití šablon bude popsáno v kapitole 5.7.

[10] `javax.annotation.processing.AbstractProcessor`



Obrázek 5.3. Detailnější diagram fází 4.3 anotačního procesoru

5.4 Zpracování konfigurovatelných atributů

Při zpracovávání konfigurační třídy je využíván návrhový vzor Visitor, který umožňuje zpracovávat jednotlivé prvky stromové struktury na základě jejich typu [31]. Elementy jsou strukturovány jako původní kód, tedy element třídy (`TypeElement`[11]) obsahuje vnitřní elementy jako atributy (`VariableElement`[12]), metody a konstruktory (`ExecutableElement`[13]). Při každém navštívení elementu, který reprezentuje atribut třídy, je třeba rozhodnout, zda lze na tento atribut mapovat data z konfigurace. Protože existuje několik pravidel, podle kterých se mapování řídí a nejedná se o zcela primitivní podmínky, byla tato pravidla rozdělena do samostatných tříd zvaných „deskriptory“. Ty jsou navzájem propojeny návrhovým vzorem Chain of Responsibility [31]. Každý deskriptor může rozhodnout, zda je možné na daný atribut mapovat konfiguraci či nikoli a nechat v takovém případě rozhodnout následující deskriptor. Pořadí deskriptorů je následující:

1. Konstruktor s anotací `@ConstructorBinding` – V případě, že třída obsahuje více konstruktorů, definuje Spring Boot anotaci `ConstructorBinding`[14]. Pokud je některý konstruktor označen touto anotací, je použit pro mapování konfigurace. Deskriptor[15] kontroluje, zda takový konstruktor existuje a pokud ano, provádí se mapování konfigurace na parametry tohoto konstruktoru. Pokud konstruktor neexistuje, je zavolán následující deskriptor.
2. Jediný konstruktor s parametry – Pokud třída obsahuje jediný konstruktor, který není výchozí (tedy obsahuje alespoň jeden parametr), je tento konstruktor použit pro mapování konfigurace. Deskriptor[16] kontroluje, zda takový konstruktor existuje a zjišťuje, zda obsahuje parametr shodný s atributem. V případě, že parametr neexistuje, není možné na atribut mapovat konfiguraci. Pokud konstruktor neexistuje, je zavolán následující deskriptor.

[11] `javax.lang.model.element.TypeElement`
[12] `javax.lang.model.element.VariableElement`
[14] `org.springframework.boot.context.properties.bind.ConstructorBinding`
[15] `cz.lukaskabc.cvut.processor.descriptor.ConstructorBindingPropertyDescriptor`
[16] `cz.lukaskabc.cvut.processor.descriptor.SingleConstructorPropertyDescriptor`

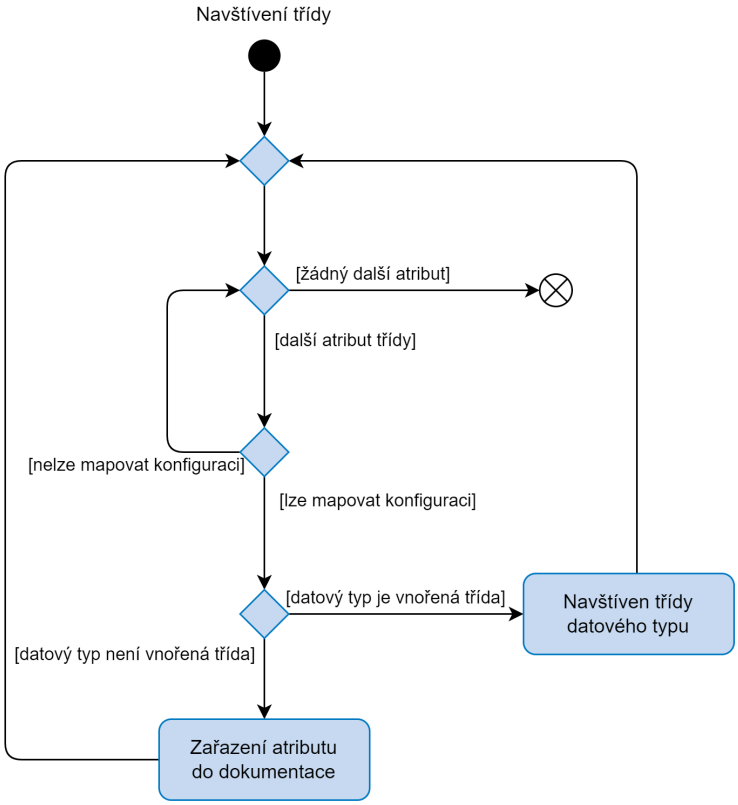
3. Record Component – Pokud je aktuálně procházená struktura záznamem (Record) a navštívený atribut je jeho komponenta, je na něj možné mapovat konfiguraci, v opačném případě deskriptor[17] ověří, že se nejedná o statický atribut a zavolá následující deskriptor. V případě, že je atribut statický, není na něj možné mapovat konfiguraci a další deskriptor již volán není.
4. Kolekce (Collection) – U atributů, které mají datový typ kolekce a mají definovanou výchozí hodnotu (instanci), se předpokládá, že jsou upravitelné. Tedy že stačí, aby Spring získal instanci kolekce pomocí getteru a naplnil ji hodnotami z konfigurace, v takovém případě tedy není potřeba setter. Deskriptor[18] ověří, zda atribut má datový typ kolekce a přiřazenou hodnotu. Pokud ano, stačí ověřit, že pro něj existuje getter. V opačném případě je zavolán následující deskriptor.
5. JavaBean atribut [24] – Aby byl atribut považován za vlastnost (property) JavaBean třídy, musí pro něj existovat getter a setter zároveň. Deskriptor[19] ověří, že takové metody existují. Pokud ne, je zavolán následující deskriptor.
6. Boolean atribut – Tento deskriptor[20] přímo neověřuje, zda je možné na atribut mapovat konfiguraci a na takové „dotazy“ jednoduše odpovídá zavoláním následujícího deskriptoru, ale vzhledem k tomu, že další deskriptor již neexistuje, vrací informaci, že na atribut není možné konfiguraci mapovat. Naopak implementuje metodu `hasGetter`, která je definována v abstraktním deskriptoru. Tato metoda je také řetězena a v případě, že nějaký deskriptor není schopen rozhodnout, zda existuje getter pro daný atribut, zavolá následující deskriptor. Konkrétně tedy tento deskriptor ověřuje, zda pro atribut existuje getter s prefixem „`is`“. Pokud se nějaký z „vyšších“ deskriptorů dotazuje na existenci getteru pro daný atribut, JavaBean deskriptor zkontroluje, zda existuje getter s prefixem „`get`“ a pokud ne, tento deskriptor ještě zkontroluje existenci getteru s prefixem „`is`“ [24].

V tuto chvíli se pomocí deskriptorů rozhodlo, že na atribut lze mapovat konfiguraci. Dalším krokem je prozkoumání datového typu právě navštíveného atributu. Existují jednoduše dvě možnosti:

1. Buď se jedná o složený datový typ (vnořenou třídu), nebo je atribut označen anotací `@NestedConfigurationProperty`. V takovém případě je třeba rekurzivně navštívit atributy tohoto datového typu. Proces rekurzivního procházení atributů tříd je znázorněn na obrázku 5.4.
2. Druhá možnost je, že se nejedná o složený datový typ a je třeba ověřit, že Spring dokáže na tento datový typ mapovat konfiguraci.

Pokud je na atribut možné mapovat konfiguraci, je zařazen do dokumentace, v opačném případě je z ní vyřazen.

[17] `cz.lukaskabc.cvut.processor.descriptor.PossiblePropertyDescriptor`
[18] `cz.lukaskabc.cvut.processor.descriptor.CollectionPropertyDescriptor`
[19] `cz.lukaskabc.cvut.processor.descriptor.JavaBeanPropertyDescriptor`
[20] `cz.lukaskabc.cvut.processor.descriptor.BooleanIsPrefixedGetterDescriptor`



Obrázek 5.4. Proces rekurzivního procházení atributů tříd

5.5 Opakované použití vnořené třídy

Při sběru dokumentace z Javadoc komentářů existuje několik míst, kde by se dokumentace mohla ve struktuře konfigurační třídy nacházet. Výpis kódu 5.2 obsahuje příklad opakovaného použití vnořené třídy pro dva různé konfigurační parametry. V tomto případě se dokumentace nachází na „vnějších“ atributech `term` a `file`, které mají datový typ vnořené třídy `NamespaceDetail`. Vnořená třída `NamespaceDetail` pak obsahuje atribut `separator`, který má svou vlastní dokumentaci. Z této konfigurace vzejdou dva konfigurační parametry (proměnné prostředí):

- `TERM_SEPARATOR`
- `FILE_SEPARATOR`

Je zřejmé, že nelze dokumentaci z „vnějších“ atributů ignorovat, protože by výsledná dokumentace dvou rozdílných konfiguračních parametrů obsahovala totožný popis z „vnořeného“ atributu `separator`. Během rekurzivního navštěvování atributů a jejich datových typů je tedy nezbytné průběžně evidovat Javadoc komentáře z navštívených atributů, u kterých dochází k zanoření do jejich datového typu, aby tato dokumentace mohla být zahrnuta do popisu vnořených atributů.

```
public static class Namespace {
    /**
     * Separator of Term namespace from the parent Vocabulary identifier.
     */
    private NamespaceDetail term = new NamespaceDetail();
    /**
     * Separator of File namespace from the parent Document identifier.
     */
    private NamespaceDetail file = new NamespaceDetail();

    public static class NamespaceDetail {
        /**
         * Separator of namespace and identifier.
         */
        @NotNull
        String separator;

        // getter & setter
    }
    // getter & setter
}
```

Výpis kódu 5.2. Příklad opakovaného použití vnořené třídy

5.6 Přizpůsobení

Výstup anotačního procesoru je možné přizpůsobit. Parametry anotačního procesoru se specifikují jako parametry nástroje javac (kompilátor) začínající velkým písmenem „A“, „`-A`“, které kompilátoru říkají, aby tyto parametry byly předány anotačním procesorům. Anotační procesor bude pro své parametry využívat prefix `configurationdoc`. Celý konfigurační parametr s prefixem a hodnotou vypadá například následovně: `-Aconfigurationdoc.configuration_package=cz.cvut.kbss.termit.util`.
V případě ekosystému Maven je možné konfiguraci specifikovat jako ve výpisu kódu 4.2. Anotační procesor podporuje následující parametry:

- `configuration_package` – Umožňuje specifikovat balíček (package), ve kterém bude anotační procesor vyhledávat konfigurační třídy. Třídy mimo tento balíček budou ignorovány a nebudou zahrnuty do výsledné dokumentace. Budou zahrnuty všechny balíčky, které začínají uvedeným řetězcem.
- `output_file` – Specifikuje cestu k souboru, do kterého bude zapsána výsledná dokumentace. Pokud není specifikován, je výstup uložen do souboru „springboot-configuration.html“, respektive springboot-configuration.md.
- `order` – Nabízí tři možné hodnoty `ASC`, `DESC`, `NONE` a určuje lexikografické řazení konfiguračních parametrů ve výstupní dokumentaci. Pokud není specifikován, je použito řazení `ASC`.
- `format` – Podporuje hodnoty `HTML`, nebo `MD` (Markdown) a určuje formát výstupní dokumentace. Pokud není specifikován, je použit formát `HTML`.
- `template` – Umožňuje specifikovat cestu k souboru obsahující vlastní šablonu. Pokud není specifikován, je využita vnitřní šablona v závislosti na zvoleném formátu.
- `env_prefix` – Umožňuje specifikovat globální prefix pro proměnné prostředí.

- `prepend_required` – Upřednostní parametry nezbytné pro konfiguraci a umístí je na začátek dokumentace.
- `deprecated_last` – Uvede zastaralé (deprecated) parametry až na konci dokumentace.
- `do_not_merge` – Deaktivuje řetězení dokumentace, které bylo popsáno v kapitole 5.5. Anotační procesor použije první dokumentaci, na kterou u daného parametru narazí.
- `no_html` – Deaktivuje generování HTML tagů anotačním procesorem při použití výstupního formátu Markdown. Blíže popsáno v kapitole 5.7.

5.7 Template

Anotační procesor využívá template engine FreeMarker, který byl blíže popsán, včetně základní syntaxe, v kapitole 4.4. Konfigurační parametry a jejich dokumentace jsou shromážděny ze zpracovávaného kódu a každý konfigurovatelný parametr je ve fázi generování dokumentace (viz obrázek 5.3) reprezentován objektem třídy `DocumentedElement`. Zdokumentovaný element poskytuje informace o deprecated stavu, o tom, zda je parametr nezbytný (required), odpovídající název proměnné prostředí, dokumentaci tohoto parametru a popis omezení pro jeho hodnoty (JSR 303 anotací), viz diagram tříd 5.1. Takto zdokumentované parametry jsou uloženy v kolekci, která je předána template enginu.

Anotační procesor obsahuje dvě interní šablony, které jsou předpřipraveny pro podporu formátů HTML a Markdown. Výpis kódu 5.4 obsahuje šablonu pro formát HTML. Anotační procesor také zavádí několik maker, které je možné použít v jakékoli šabloně, přičemž tato makra automaticky detekují používaný formát (HTML a Markdown) a svůj výstup dle něj přizpůsobují. Definovaná makra jsou následující:

- `[@asterisk option /]` – Jako argument přijímá objekt zdokumentovaného parametru[21]. Pokud je daný parametr nezbytný pro konfiguraci, pak je výstupem makra tučný znak hvězdičky. Pokud je u nějakého parametru výstupem hvězdička, zapíše hodnotu `true` do proměnné `areThereRequired`. Tuto proměnnou je pak možné v šabloně dále využít.
- `[@deprecated option /]` – Jako argument přijímá objekt zdokumentovaného parametru. Pokud je parametr označen jako zastaralý, je výstupem makra text „Deprecated“ formátovaný kurzívou. Obdobně jako předchozí makro, zapisuje hodnotu `true` do proměnné `areThereDeprecated`.
- `[@legend /]` – Využívá proměnnou `areThereRequired`, která v případě, že obsahuje hodnotu `true`, je výstupem makra text vysvětlující, že tučná hvězdička značí nezbytný („required“) parametr konfigurace. Pokud proměnná obsahuje hodnotu `false`, makro nevygeneruje žádný výstup.

Šablona pro HTML ve výpisu kódu 5.4 využívá všechna uvedená makra. Definuje tabulku o dvou sloupcích, první řádek obsahuje záhlaví s názvy sloupců – „Variable“ a „Description“. Následující řádky jsou generovány cyklem, který prochází kolekci `options`, ze které postupně dosazuje hodnoty do proměnné `option`. Pro každou hodnotu je vytvořen řádek tabulky. První sloupec obsahuje název proměnné, který předchází upozornění o „deprecated“ stavu, a pokud se jedná o nezbytný parametr, je následován hvězdičkou. Druhý sloupec obsahuje popis parametru, který je následován popisem omezení. U obou částí popisu, bylo třeba ošetřit, aby byly odděleny novým řádkem, a při tom, aby nový řádek nebyl přítomen, pokud jedna část z popisu je

[21] Zdokumentovaným parametrem se rozumí objekt `DocumentedElement`.

prázdná. Po tabulce následuje makro `[@legend /]`, které vypíše vysvětlení, pokud je nějaký parametr označen hvězdičkou. Šablona také využívá makra definovaná template enginem FreeMarker – podmínka `[#if]`, cyklus `[#list]`, a `[#t]`, respektive `[#lt]`, která odebírají prázdné znaky z celého řádku, respektive z jeho začátku (levé strany). Příklad výstupu této šablony je možné vidět ve výpisu kódu 5.3.

Druhá dostupná šablona je pro formát Markdown. V tomto případě je využívána syntaxe tabulek definována v GitHub Flavoured Markdown [16]. Tento zápis ovšem vyžaduje, aby jeden řádek tabulky nebyl rozdělen mezi více řádků v Markdown souboru. To znamená, že je třeba z popisů parametrů a celkového obsahu tabulky odebrat znaky nových řádků. Protože Markdown nenabízí žádnou možnost, jak vynutit nový řádek bez vložení reálného nového řádku, je třeba nové řádky nahradit pouhou mezerou. Ovšem v souvislosti s využitím Markdown formátu na stránkách portálu GitHub, kde je Markdown zpracováván do formátu HTML a tedy jsou některé HTML tagy v Markdown zápisu podporovány, je možné nové řádky vytvořit použitím tagu `<br>`. Samozřejmě použití tohoto tagu nemusí být vždy žádoucí, proto je možné pomocí parametru anotačního procesoru `no_html_tags` vypnout generování těchto tagů.

```
<table>
    <tr>
        <th>Variable</th>
        <th>Description</th>
    </tr>
[#list options as option]
    <tr>
        <td>
[@deprecated option /]<code>${option.name}</code>[@asterisk option /]
        </td>
        <td>
            [#if option.description?trim?length > 0]
                ${option.description}[#t]

            [/#if]
            [#if option.restrictions?trim?length > 0]
                [#if option.description?trim?length > 0]
                    <br>[#lt]
                [/#if]
                ${option.restrictions}[#lt]
            [/#if]
        </td>
    </tr>
[/#list]
</table>

[@legend /]
```

Výpis kódu 5.4. FreeMarker šablona pro formát HTML, upraveno pro potřeby dokumentu

```
<table>
    <tr>
        <th>Variable</th>
        <th>Description</th>
    </tr>
    <tr>
        <td>
<code>TERMIT_ADMIN_CREDENTIALSFILE</code><b>&#42;</b>
        </td>
        <td>
Name of the file in which admin credentials are saved when its account
is generated.
<br>
value must be present
        </td>
    </tr>
    <tr>
        <td>
<i>Deprecated </i><code>TERMIT_TERMASSIGNMENTMINSCORE</code><b>&#42;</b>
        </td>
        <td>
Minimal match score of a term occurrence for which a term assignment
should be automatically generated.
Deprecated: This configuration is currently not used.
<br>
value must be present
        </td>
    </tr>
</table>

<b>&#42; Required</b>
```

Výpis kódu 5.3. Výstup šablony z výpisu kódu 5.4, upraveno a zkráceno pro potřeby dokumentu

5.8 Zajímavosti implementace

Během vývoje a testování anotačního procesoru bylo identifikováno několik funkcí frameworku Spring, respektive Spring Boot, které ovlivnily implementaci. Některé funkce nebyly dostatečně zdokumentovány a jejich chování tedy nebylo zcela očekáváno. V následujících kapitolách budou popsány stěžejní funkce a chování frameworku, které bylo třeba v implementaci zohlednit.

5.8.1 Duplicitní Spring Bean

Framework Spring je IoC kontejner, který spravuje životní cyklus aplikace. Zároveň spravuje životní cyklus některých objektů. Takto spravované objekty jsou označovány jako „bean“. Pokud není nějakým způsobem specifikováno jinak, je k vytvořené bean přiřazen název vygenerovaný z názvu třídy. Prostřednictvím tohoto názvu je pak k bean možné přistupovat z ostatních tříd. V případě konfigurační třídy z výpisu kódu 4.1 je vytvořena bean s názvem `termitConfiguration`. Použití

samostatné anotace `@ConfigurationProperties`[22] není pro vytvoření bean dostatečné, aby byla opravdu bean vytvořena, je buď třeba použít anotaci `@Configuration`[23] přímo na konfigurační třídě, nebo použít anotaci `@ConfigurationPropertiesScan`[24] pro vyhledávání „`@ConfigurationProperties`“ tříd. Zvláštní situace, která může nastat uživatelskou chybou, je použití anotace `@ConfigurationProperties` na třídě, která zároveň slouží jako datový typ atributu jiné konfigurační třídy[25].

Výpis kódu 5.5 znázorňuje příklad konfigurační třídy, která obsahuje vnořenou třídu. Anotace `@ConfigurationProperties` a `@Configuration` se nachází na vnější, ale i vnořené třídě. To znamená, že budou vytvořeny dvě bean s názvy `termitConfiguration` a `repository`. Na každý z těchto objektů bude konfigurace mapována zvlášť. Objekty `termitConfiguration.repository` a `repository` nejsou totožné, ale jedná se o dvě samostatné instance. Pokud nějaký kód bude tuto konfiguraci využívat, bude záležet, jakým způsobem přistoupí k hodnotě cílového atributu `url` uvnitř vnořené třídy `Repository`. Výpis kódu 5.6 obsahuje konfiguraci ve formátu YAML, která je mapována zvlášť na každou z vytvořených bean.

Pokud kód, který chce využít konfiguraci, přistoupí k atributu pomocí bean s názvem `termitConfiguration`, ve které přistoupí k atributu `repository` a následně k atributu `url` („`termitConfiguration.getRepository().getUrl()`“), obdrží hodnotu „`termit repository`“. Pokud použije bean `repository` (přistoupí k hodnotě pomocí „`repository.getUrl()`“), obdrží hodnotu „`another repository`“.

Takovéto použití konfiguračních tříd je zcela validní, ale nemusí být přehledné a ve většině případů jej lze považovat za neúmyslné. Dokumentační procesor dokáže toto použití detekovat a vygenerovat varování, které upozorní na možné nechtěné použití anotací.

5.8.2 Vynucení názvu konfigurační hodnoty

Jazyk Java neumožňuje definovat název proměnné, který by se shodoval s některým klíčovým slovem (např. `class`, `int`, `public`). Aby bylo možné mapovat konfiguraci pod těmito klíčovými slovy, nabízí knihovna Spring Boot anotaci `@Name`[26]. Tuto anotaci je možné použít na parametr metody, v kontextu konfigurační třídy přesněji na parametr konstruktoru, který je používán pro mapování konfigurace. Jako argument anotace je možné specifikovat název, pod kterým bude konfigurace mapována. Tedy například pokud je definován konstruktor s parametrem jako `MyConfiguration(@Name("class") String clazz)`, bude na jeho atribut mapována konfigurace s klíčem `class`. Tuto anotaci bere dokumentační procesor v potaz a adekvátně upraví název konfigurovatelného parametru v dokumentaci.

5.8.3 JSR 303 Validace

Aby mohly JSR 303 anotace správně fungovat, je třeba, aby byla aktivována validace konfigurace. Validaci je možné na konfigurační třídě aktivovat pomocí anotace `@Validated`[27]. Pokud konfigurační třída obsahuje atribut s komplexním datovým typem, u kterého dochází k procházení jeho atributů, je třeba validaci vnitřní struktury vynutit anotací `@Valid`[28] umístěné na atribut v konfigurační třídě.

[22] `org.springframework.boot.context.properties.ConfigurationProperties`
[23] `org.springframework.context.annotation.Configuration`
[24] `org.springframework.boot.context.properties.ConfigurationPropertiesScan`
[25] Zde se konfigurační třídou rozumí opět třída s anotací `@ConfigurationProperties`
[26] `org.springframework.boot.context.properties.bind.Name`
[27] `org.springframework.validation.annotation.Validated`
[28] `javax.validation.Valid`

Dokumentační procesor použití těchto anotací detekuje a v případě jejich záměny, nebo absence, vygeneruje adekvátní varování. Tím je uživatel upozorněn například na situaci, kdy byly použity validační anotace, ale samotná validace nebyla aktivována.

5.8.4 Record třídy

Zvláštním případem konfiguračních tříd je použití „záznamů“ (record). Record může být použit jako samotná hlavní konfigurační třída s anotací `@ConfigurationProperties`, nebo může představovat datový typ atributu jiné konfigurační třídy. Protože record není modifikovatelný, mapování konfigurace na jeho atributy (komponenty) probíhá prostřednictvím konstruktoru. Aplikují se zde stejná pravidla jako při mapování na běžné konfigurační třídy, viz kapitola 5.4.

```
@Configuration
@ConfigurationProperties(prefix = "termit")
@Validated
public class TermitConfiguration {

    @Valid
    private Repository repository = new Repository();

    @Configuration
    @ConfigurationProperties
    public static class Repository {
        /**
        * URL of the main application repository.
        */
        @NotNull
        String url;

        // getter & setter
    }

    // getter & setter
}
```

Výpis kódu 5.5. Příklad „chybného“ použití konfiguračních anotací, převzato z [19], upraveno

```
termit:
repository:
url: "termit repository"

repository:
url: "another repository"
```

Výpis kódu 5.6. Konfigurace ve formátu YAML pro dvě různé bean z výpisu kódu 5.5

Kapitola 6
Vyhodnocení

V této kapitole bude popsán způsob, jakým byl dokumentační procesor testován. A bude také prezentováno porovnání automaticky generované dokumentace procesorem a manuálně vytvořené dokumentace projektu TermIt.

6.1 Testování

Cílem testování bylo ověřit výstupy dokumentačního procesoru a hlídat jejich konzistenci. Pro udržení přehlednosti projektu byl Maven projekt rozdělen do dvou modulů, kde první modul obsahuje samotný dokumentační procesor a druhý modul obsahuje testy. Každý test spouští Java kompilátor s dokumentačním procesorem a jeho specifickým nastavením. Jako vstup definuje konfigurační třídu určenou pro testování a očekává výstupní soubor, jehož obsah je následně porovnán s očekávaným výstupem. Test je úspěšný, pokud výstup dokumentačního procesoru odpovídá očekávanému výstupu. Tímto způsobem jsou testovány různé funkce dokumentačního procesoru jako detekce konfigurovatelných parametrů, popis JS303 anotací, reakce na konfiguraci (změnu pořadí) a tak dále.

Realizace testů byla uskutečněna pomocí abstraktní třídy `AbstractProcessorTest`, která poskytuje metody pro spuštění testu. Ostatní testovací třídy tuto abstraktní třídu rozšiřují a pro spuštění testu vždy jen specifikují vstupní soubor, očekávaný formát a případnou konfiguraci dokumentačního procesoru. `AbstractProcessorTest` již zajistí spuštění kompilátoru a ověření výstupů.

6.2 Výstupy nástroje

Prvním krokem pro zhodnocení výstupů dokumentačního procesoru je jeho integrace do projektu TermIt. Ta již byla částečně naznačena ve výpisu kódu 4.2. Výpis kódu 6.1 obsahuje definici Maven profilu „`configuration-doc`“, který spustí Java kompilátor s dokumentačním procesorem.

Spuštěním tohoto profilu je vygenerován soubor `springboot-configuration.html`, jeho obsah je zobrazen na obrázcích 6.1, 6.2, 6.3. Na obrázku 6.4 je zobrazena manuálně vytvořená dokumentace projektu TermIt. Na první pohled je patrné, že dokumentace vygenerovaná procesorem obsahuje více položek (39) než manuálně vytvořená dokumentace (22). Navzdory tomu, procesorem vygenerovaná dokumentace postrádá některé parametry, které jsou v manuálně vytvořené dokumentaci přítomny: `SPRING_MAIL_HOST`, `SPRING_MAIL_PORT` a `SPRING_MAIL_PASSWORD`. Tyto parametry nejsou procesorem zdokumentovány, protože se explicitně nenacházejí v kódu aplikace. Jedná se o parametry knihovny určené pro odesílání emailů. Přidání takových parametrů s jejich dokumentací je možné použitím vlastní šablony a manuálním doplněním dokumentace, nebo použití těchto parametrů někde v kódu aplikace (například pomocí anotace `@Value`). Naopak, parametry v procesorem vygenerované dokumentaci, které jsou navíc, lze skrýt použitím Javadoc tagu `@hidden`, viz kapitola 4.3.

```
<profile>
  <id>configuration-doc</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <proc>only</proc>
          <annotationProcessorPaths>
            <path>
              <groupId>cz.lukaskabc.cvut.processor</groupId>
              <artifactId>
                spring-boot-configuration-docgen-processor
              </artifactId>
              <version>0.0.1</version>
            </path>
          </annotationProcessorPaths>
          <source>${java.version}</source>
          <target>${java.version}</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
```

Výpis kódu 6.1. Příklad vytvoření Maven profilu pro spuštění Java kompilátoru s dokumentačním procesorem

Variable	Description
`APPLICATION_VERSION`	Default value: `development`
`SPRING_MAIL_USERNAME`	Default value: `#{null}`
`SPRING_SERVLET_MULTIPART_MAXFILESIZE`	
`TERMIT_ACL_DEFAULTEDITORACCESSLEVEL`	Default access level for users in editor role. Default value: `READ`
`TERMIT_ACL_DEFAULTREADERACCESSLEVEL`	Default access level for users in editor role. Default value: `READ`
`TERMIT_ADMIN_CREDENTIALSFILE` *	Name of the file in which admin credentials are saved when its account is generated. value must be present
`TERMIT_ADMIN_CREDENTIALSLOCATION` *	Specifies the folder in which admin credentials are saved when its account is generated. value must be present
`TERMIT_CHANGETRACKING_CONTEXT_EXTENSION` *	Extension appended to asset identifier (presumably a vocabulary ID) to denote its change tracking context identifier. value must be present
`TERMIT_COMMENTS_CONTEXT` *	IRI of the repository context used to store comments (discussion to assets). value must be present
`TERMIT_CORS_ALLOWEDORIGINPATTERNS`	A comma-separated list of allowed origin patterns for CORS. This allows a more dynamic configuration of allowed origins that `#allowedOrigins` which contains exact origin URLs. It is useful, for example, for Netlify preview builds of the frontend which use a generated subdomain URL.
`TERMIT_CORS_ALLOWEDORIGINS` *	A comma-separated list of allowed origins for CORS. Default value: `http://localhost:3000` value must be present
`TERMIT_FILE_STORAGE` *	Specifies root directory in which document files are stored. value must be present
`TERMIT_GLOSSARY_FRAGMENT` *	IRI path to append to vocabulary IRI to get glossary identifier. value must be present
`TERMIT_JMXBEANNAME`	Name of the JMX bean exported by TermIt. Normally should not need to change unless multiple instances of TermIt are running in the same application server. Default value: `TermItAdminBean`
`TERMIT_JWT_SECRETKEY`	Secret key used when hashing a JWT.
`TERMIT_LANGUAGE_STATES_SOURCE`	Path to a file containing definition of the language of states terms can be in with. The file must be in Turtle format. The term definitions must use SKOS terminology for attributes (prefLabel, scopeNote and broader/narrower).
`TERMIT_LANGUAGE_TYPES_SOURCE`	Path to a file containing definition of the language of types terms can be classified with. The file must be in Turtle format. The term definitions must use SKOS terminology for attributes (prefLabel, scopeNote and broader/narrower).
`TERMIT_MAIL_SENDER`	Human-readable name to use as email sender.

Obrázek 6.1. Dokumentace projektu TermIt [19] vygenerovaná dokumentačním procesorem (1/3), zobrazena portálem GitHub

`TERMIT_NAMESPACE_FILE_SEPARATOR` *	Separator of File namespace from the parent Document identifier. Since File identifier is given by the identifier of the Document it belongs to and its own normalized label, this separator is used to (optionally) configure the File identifier namespace. For example, if we have a Document with IRI `http://www.example.org/ontologies/resources/metropolitan-plan/document` and a File with normalized label `main-file`, the resulting IRI will be `http://www.example.org/ontologies/resources/metropolitan-plan/document/SEPARATOR/main-file`, where 'SEPARATOR' is the value of this configuration parameter. value must be present
`TERMIT_NAMESPACE_RESOURCE` *	Namespace for resource identifiers. value must be present
`TERMIT_NAMESPACE_SNAPSHOT_SEPARATOR` *	Separator of snapshot timestamp and original asset identifier. For example, if we have a Vocabulary with IRI `http://www.example.org/ontologies/vocabularies/metropolitan-plan` and the snapshot separator is configured to `version`, a snapshot will IRI will look something like `http://www.example.org/ontologies/vocabularies/metropolitan-plan/version/20220530T202317Z`. value must be present
`TERMIT_NAMESPACE_TERM_SEPARATOR` *	Separator of Term namespace from the parent Vocabulary identifier. Since Term identifier is given by the identifier of the Vocabulary it belongs to and its own normalized label, this separator is used to (optionally) configure the Term identifier namespace. For example, if we have a Vocabulary with IRI `http://www.example.org/ontologies/vocabularies/metropolitan-plan` and a Term with normalized label `inhabited-area`, the resulting IRI will be `http://www.example.org/ontologies/vocabularies/metropolitan-plan/SEPARATOR/inhabited-area`, where 'SEPARATOR' is the value of this configuration parameter. value must be present
`TERMIT_NAMESPACE_USER` *	Namespace for user identifiers. value must be present
`TERMIT_NAMESPACE_VOCABULARY` *	Namespace for vocabulary identifiers. value must be present
`TERMIT_PERSISTENCE_DRIVER` *	OntoDriver class for the repository. value must be present
`TERMIT_PERSISTENCE_LANGUAGE` *	Language used to store strings in the repository (persistence unit language). value must be present
`TERMIT_PUBLICVIEW_WHITELISTPROPERTIES` *	Unmapped properties allowed to appear in the SKOS export. value must be present
`TERMIT_REPOSITORY_PASSWORD`	Password for connecting to the application repository.
`TERMIT_REPOSITORY_PUBLICURL`	Public URL of the main application repository. Can be used to provide read-only no authorization access to the underlying data.
`TERMIT_REPOSITORY_URL` *	URL of the main application repository. value must be present

Obrázek 6.2. Dokumentace projektu TermIt [19] vygenerovaná dokumentačním procesorem (2/3), zobrazena portálem GitHub

`TERMIT_REPOSITORY_USERNAME`	Username for connecting to the application repository.
`TERMIT_SCHEDULE_CRON_NOTIFICATION_COMMENTS`	CRON expression configuring when to send notifications of changes in comments to admins and vocabulary authors. Defaults to '-' which disables this functionality. Default value: `-`
`TERMIT_SECURITY_PROVIDER`	Determines whether an internal security mechanism or an external OIDC service will be used for authentication. In case na OIDC service is selected, it should be configured using standard Spring Boot OAuth2 properties. Default value: `INTERNAL`
`TERMIT_SECURITY_ROLECLAIM`	Claim in the authentication token provided by the OIDC service containing roles mapped to TermIt user roles. Supports nested objects via dot notation. Default value: `realm_access.roles`
Deprecated `TERMIT_TEXTANALYSIS_TERMASSIGNMENTMINSCORE` *	Minimal match score of a term occurrence for which a term assignment should be automatically generated. More specifically, when annotated file content is being processed, term occurrences with sufficient score will cause creation of corresponding term assignments to the file. Deprecated: This configuration is currently not used. value must be present
`TERMIT_TEXTANALYSIS_TERMOCCURRENCEMINSCORE` *	Score threshold for a term occurrence for it to be saved into the repository. Default value: `0.49` value must be present
`TERMIT_TEXTANALYSIS_URL`	URL of the text analysis service.
`TERMIT_URL`	TermIt frontend URL. It is used, for example, for links in emails sent to users. Default value: `http://localhost:3000/#`
`TERMIT_WORKSPACE_ALLVOCABULARIESEDITABLE` *	Whether all vocabularies in the repository are editable. Allows running TermIt in two modes - one is that all vocabularies represent the current version and can be edited. The other mode is that working copies of vocabularies are created and the user only selects a subset of these working copies to edit (the so-called workspace), while all other vocabularies are read-only for them. Default value: `true` value must be present

*** Required**

Obrázek 6.3. Dokumentace projektu TermIt [19] vygenerovaná dokumentačním procesorem (3/3), zobrazena portálem GitHub

Variable	Explanation
`TERMIT_URL`	TermIt frontend URL. Used, for example, for links in emails sent to users.
`TERMIT_PERSISTENCE_LANGUAGE`	Primary language used to store strings in the repository (persistence unit language).
`TERMIT_CHANGETRACKING_CONTEXT_EXTENSION`	Extension appended to asset identifier (presumably a vocabulary ID) to denote its change tracking context identifier.
`TERMIT_COMMENTS_CONTEXT`	IRI of the repository context used to store comments.
`TERMIT_NAMESPACE_VOCABULARY`	Namespace for vocabulary identifiers.
`TERMIT_NAMESPACE_USER`	Namespace for user identifiers.
`TERMIT_NAMESPACE_RESOURCE`	Namespace for resource identifiers
`TERMIT_NAMESPACE_TERM_SEPARATOR`	Separator of Term namespace from the parent Vocabulary identifier. Since Term identifier is given by the identifier of the Vocabulary it belongs to and its own normalized label, this separator is used to (optionally) configure the Term identifier namespace. For example, if we have a Vocabulary with IRI `http://www.example.org/ontologies/vocabularies/metropolitan-plan` and a Term with normalized label `inhabited-area`, the resulting IRI will be `http://www.example.org/ontologies/vocabularies/metropolitan-plan/SEPARATOR/inhabited-area`, where 'SEPARATOR' is the value of this configuration parameter.
`TERMIT_NAMESPACE_FILE_SEPARATOR`	Separator of File namespace from the parent Document identifier. See above for explanation.
`TERMIT_NAMESPACE_SNAPSHOT_SEPARATOR`	Separator of snapshot timestamp and original asset identifier. For example, if we have a Vocabulary with IRI `http://www.example.org/ontologies/vocabularies/metropolitan-plan` and the snapshot separator is configured to `version`, a snapshot will IRI will look something like `http://www.example.org/ontologies/vocabularies/metropolitan-plan/version/20220530T202317Z`.
`TERMIT_ADMIN_CREDENTIALSLOCATION`	Specifies the folder in which admin credentials are saved when its account is generated.
`TERMIT_ADMIN_CREDENTIALSFILE`	Name of the file in which admin credentials are saved when its account is generated.
`TERMIT_TEXTANALYSIS_TERMOCCURRENCEMINSCORE`	Score threshold for a term occurrence for it to be saved into the repository.
`TERMIT_GLOSSARY_FRAGMENT`	IRI path to append to vocabulary IRI to get glossary identifier.
`TERMIT_PUBLICVIEW_WHITELISTPROPERTIES`	A comma-separated set of unmapped properties allowed to appear in the base SKOS export.
`TERMIT_WORKSPACE_ALLVOCABULARIESEDITABLE`	Whether all vocabularies in the repository are editable. Allows running TermIt in two modes - one is that all vocabularies represent the current version and can be edited. The other mode is that working copies of vocabularies are created and the user only selects a subset of these working copies to edit (the so-called workspace), while all other vocabularies are read-only for them.
`TERMIT_SCHEDULE_CRON_NOTIFICATION_COMMENTS`	CRON expression configuring when to send notifications of changes in comments to admins and vocabulary authors. Defaults to `-` which disables this functionality.
`TERMIT_MAIL_SENDER`	Human-readable name to use as email sender.
`SPRING_MAIL_HOST`	Email server hostname.
`SPRING_MAIL_PORT`	Email server port.
`SPRING_MAIL_USERNAME`	Email server username.
`SPRING_MAIL_PASSWORD`	Email server password.

Obrázek 6.4. Manuální dokumentace projektu TermIt [19] zobrazena portálem GitHub

Kapitola 7
Závěr

Při vývoji aplikace je třeba patřičně zdokumentovat její možnosti konfigurace. Nástroj implementovaný v rámci této práce umožňuje automatické generování dokumentace ve formátu HTML či Markdown na základě Javadoc komentářů v konfiguračních třídách. Tím zjednodušuje proces údržby a aktualizace dokumentované konfigurace a snižuje riziko vzniku chyb a nekonzistencí mezi kódem a dokumentací.

V rámci této práce byla představena problematika konfigurace aplikací využívajících knihovnu Spring Boot. Byly představeny související technologie a možná řešení pro zpracování konfigurace a následné vygenerování dokumentace. Možná řešení byla porovnána a jako vhodné řešení byl implementován nástroj v podobě anotačního procesoru. Této práci předcházela implementace Javadoc docletu v rámci semestrálního projektu, od této implementace bylo upuštěno kvůli možnosti přístupu k výchozím hodnotám, kterou doclet na rozdíl od anotačního procesoru neposkytoval. Protože Javadoc doclety sdílí využívané rozhraní kompilátoru s anotačními procesory, byl možný převod implementace ze struktury docletu na anotační procesor bez větších obtíží.

Implementovaný anotační procesor umožňuje přizpůsobení obsahu a formátu generované dokumentace pomocí argumentů a šablon. Ve výchozím nastavení generuje HTML soubor obsahující tabulku s konfiguračními parametry a jejich popisem. Druhým podporovaným formátem generované dokumentace je Markdown. V dokumentaci automaticky zahrnuje relevantní Javadoc tagy a popisy JSR 303 anotací. Projekt je umístěn na portálu GitHub[1] a pro snadnou integraci do ostatních projektů je výsledný artifakt dostupný na portálu Maven Central.

V budoucnu by bylo možné rozšířit možnosti přizpůsobení výstupu větším využitím šablon. Například popis JSR 303 anotací by bylo možné realizovat pomocí maker, která by mohla šablona přizpůsobit. Bylo by možné přidat podporu dalších výstupních formátů jako např. Asciidoc, či také umožnit dokumentaci konfiguračních souborů různých formátů namísto proměnných prostředí.

[1] https://github.com/lukaskabc/spring-boot-configuration-docgen

Literatura

[1] Sayagh, Mohammed, Noureddine Kerzazi, Bram Adams a Fabio Petrillo. Software Configuration Engineering in Practice Interviews, Survey, and Systematic Literature Review. *IEEE Transactions on Software Engineering* [online]. 2020-6-1, ročník 46, č. 6, s. 646-673. ISSN 0098-5589. Dostupné na DOI 10.1109/TSE.2018.2867847.

[2] *Apache Maven Project* [online]. The Apache Software Foundation. c2002–2023. [vid. 2023-08-12]. Dostupné na `https://maven.apache.org/`.

[3] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith a Gavin Bierman. *The Java Language Specification Java SE 17 Edition* [online]. [vid. 2024-02-12]. Dostupné na `https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf`.

[4] *Compilation Overview* [online]. Oracle and/or its affiliates. c2024. [vid. 2024-03--09]. Dostupné na `https://openjdk.org/groups/compiler/doc/compilation-overview/index.html`.

[5] *JetBrains: Developer Ecosystem* [online]. JetBrains. 2023. [vid. 2024-02-18]. Dostupné na `https://www.jetbrains.com/lp/devecosystem-2023/java/`.

[6] Cosmina, Iuliana, Rob Harrop, Chris Schaefer, Clarence Ho a SpringerLink (online služba) . *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools*. 5. vyd. Berkeley, CA: Apress, 2017. ISBN 9781484228074. Dostupné na DOI 10.1007/978-1-4842-2808-1.

[7] *Spring by VMware Tanzu* [online]. VMware, Inc. or its affiliates. c2023. [vid. 2023--08-11]. Dostupné na `https://spring.io`.

[8] *Spring Framework Documentation* [online]. VMware, Inc. or its affiliates. c2023. [vid. 2023-08-11]. Dostupné na `https://docs.spring.io/spring-framework/reference/6.0.11/index.html`.

[9] Webb, Phillip a kol. *Spring Boot Reference Documentation* [online]. Version 3.1.2. 2023. [vid. 2023-07-20]. Dostupné na `https://docs.spring.io/spring-boot/docs/3.1.0/reference/html/index.html`.

[10] IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(TM)). *IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1--1996 and IEEE Std 1003.2-1992)*. 2001, s. 159-160 [vid. 2023-09-05]. Dostupné na DOI 10.1109/IEEESTD.2001.93364. Dostupné na `https://ieeexplore.ieee.org/servlet/opac?punumber=7683`.

[11] *Spring Boot: javadoc* [online]. Version 3.1.0. VMware, Inc. or its affiliates. [2023]. [vid. 2023-08-11]. Dostupné na `https://docs.spring.io/spring-boot/docs/3.1.0/api/`.

[12] *Spring Framework: javadoc* [online]. Version 6.0.11. VMware, Inc. or its affiliates. [2023]. [vid. 2023-08-11]. Dostupné na `https://docs.spring.io/spring-framework/docs/6.0.11/javadoc-api/`.

[13] Bernard, Emmanuel a kol. *JSR 303: Bean Validation*. Red Hat. 2009. [vid. 14.09.2023]. Dostupné na `https://jcp.org/en/jsr/detail?id=303`.

[14] *Javadoc* [online]. Redwood Shores. Oracle and/or its affiliates. c1994-2023. [vid. 2023-08-11]. Dostupné na `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`.

[15] MacFarlane, John. *CommonMark Spec* [online]. Version 0.30 (2021-06-19). . [vid. 2023-08-11]. Dostupné na `https://spec.commonmark.org/0.30/`.

[16] *GitHub Flavored Markdown Spec* [online]. San Francisco, CA. GitHub, Inc. 2019. [vid. 2023-08-11]. Dostupné na `https://github.github.com/gfm/`.

[17] *FreeMarker Java Template Engine* [online]. The Apache Software Foundation. c1999–2024. [vid. 2024-03-05]. Dostupné na `https://freemarker.apache.org/`.

[18] Verhas, Peter, Mihály Verhás a Rahman Usta. *Jamal Macro Language* [online]. [vid. 2024-02-14]. Dostupné na `https://github.com/verhas/jamal`.

[19] Ledvinka, Martin, Petr Křemen, Lama Saeeda a Miroslav Blaško. TermIt: A Practical Semantic Vocabulary Manager. In: *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 1: ICEIS*. Ithaca: SciTePress, 2020. s. 759-766. ISBN 978-989-758-423-7. ISSN 2184-4992. Dostupné na DOI 10.5220/0009563707590766.

[20] Holozsnyák, Nándor. *Spring Configuration Property Documenter* [online]. [vid. 2024-02-14]. Dostupné na `https://github.com/rodnansol/spring-configuration-property-documenter`.

[21] McCluskey, Glen. *Using Java Reflection* [online]. Oracle and/or its affiliates. 1998. [vid. 2023-09-01]. Dostupné na `https://www.oracle.com/technical-resources/articles/java/javareflection.html`.

[22] *JavaBeans* [online]. 2550 Garcia Avenue, Mountain View, CA 94043. Oracle and/or its affiliates. 1997. [vid. 2024-04-23]. Dostupné na `https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/`.

[23] Forman, Ira R. a Nate Forman. *Java Reflection in action*. 1 vyd. Greenwich: Manning Publications, 2005. ISBN 9781932394184.

[24] *Springdoc*. 2024. [vid. 2024-04-23]. Dostupné na `https://springdoc.org/`.

[25] *JavaParser* [online]. c2019. [vid. 2023-09-11]. Dostupné na `https://javaparser.org`.

[26] *What is a container?* [online]. Microsoft Azure. c2023. [vid. 2023-11-12]. Dostupné na `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container/`.

[27] *The Javac Command* [online]. Redwood Shores. Oracle and/or its affiliates. c1993--2023. [vid. 2024-03-09]. Dostupné na `https://docs.oracle.com/en/java/javase/17/docs/specs/man/javac.html`.

[28] *Project Lombok* [online]. The Project Lombok Authors. c2009-2023. [vid. 2024-03--02]. Dostupné na `https://projectlombok.org/`.

[29] *Oracle Java SE Support Roadmap* [online]. Oracle and/or its affiliates. 2022. [vid. 2023-09-13]. Dostupné na `https://www.oracle.com/java/technologies/java-se-support-roadmap.html`.

[30] *Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification* [online]. Redwood Shores. Oracle and/or its affiliates. c1993-2023. [vid. 2024-

-03-09]. Dostupné na https://docs.oracle.com/en/java/javase/17/docs/api/.

[31] Gamma, Erich, Richard Helm, Ralph Johnson a John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. vyd. Hoboken: Pearson Education, Limited, 1994. ISBN 9780201633610.

Příloha A
Zkratky

API	Application Programming Interface
AST	Abstract Syntax Tree
GFM	GitHub Flavored Markdown
HTML	Hyper Text Markup Language
HTTP	HyperText Transfer Protocol
IoC	Inversion of Control
JAR	Java Archive
JDK	Java Development Kit
POM	Project Object Model
REST	Representational State Transfer
SpEL	Spring Expression Language