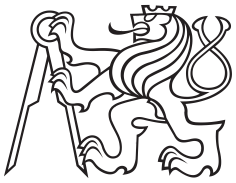
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Mapování atributů založené na SPARQL dotazech v knihovně JOPA

Zdeněk Kotrlý

Vedoucí: Ing. Martin Ledvinka
Obor: Softwarové inženýrství a technologie
Studijní program: Softwarové inženýrství a technologie
Květen 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kotrlý** Jméno: **Zdeněk** Osobní číslo: **483484**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Mapování atributů založené na SPARQL dotazech v knihovně JOPA

Název bakalářské práce anglicky:

SPARQL-based attribute mapping in JOPA

Pokyny pro vypracování:

1. Seznamte se s principy sémantického webu (hlavně RDF, SPARQL) a knihovnou JOPA.
2. Analyzujte, jakým způsobem by v knihovně JOPA bylo možné realizovat podporu pro mapování atributů pomocí SPARQL dotazů.
3. Navrhněte rozšíření knihovny JOPA o podporu mapování atributů pomocí SPARQL dotazů.
4. Implementujte váš návrh a integrujte jej do knihovny JOPA.
5. Ověřte správnost řešení pomocí sady testovacích dotazů a dat poskytnutých vedoucím práce.

Seznam doporučené literatury:

[1] M. Keith and M. Schincariol, Pro JPA 2: Mastering the Java™ Persistence API, Apress, 2009
[2] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C recommendation, W3C, 2013
[3] R. Cyganiak, D. Wood, and M. Lanthaler, RDF 1.1 Concepts andAbstract Syntax, W3C recommendation, W3C, 2014

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL

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

Datum zadání bakalářské práce: **12.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

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

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

Poděkování

Chtěl bych poděkovat především svému vedoucímu Ing. Martinu Ledvinkovi za ochotu a trpělivost při konzultacích a cenné rady při vypracování bakalářské práce.

Prohlášení

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 .....

.....
podpis autora práce

Abstrakt

Knihovna JOPA je zavedeným nástrojem pro přístup k ontologiím a sémantickým datům v programovacím jazyce Java. Tato práce se snaží prozkoumat potenciální rozšíření JOPA, které by umožnilo definici atributů založených pouze na SPARQL dotazech. Následující text se skládá z objasnění fundamentálních sémantických technologií a pojmů perzistentních API, vysokoúrovňové analýzy knihovny JOPA, dvou navržených řešení zmíněného rozšíření této knihovny s detailnějším popisem vlastní implementace jednoho z nich a nakonec z vysvětlení, jak byla správnost nových atributů otestována.

Klíčová slova: Sémantické technologie, Persistence, Objektově-ontologické mapování, SPARQL

Vedoucí: Ing. Martin Ledvinka
Skupina znalostních softwarových systémů,
Karlovo náměstí 13,
Praha 2

Abstract

The JOPA library is an established tool for accessing ontologies and semantic data in the Java programming language. This thesis aims to explore a potential extension of JOPA that would allow for the definition of attributes based purely on SPARQL queries. The following text consists of an explanation of fundamental semantic technologies as well as persistence API terms, a high level analysis of JOPA, two proposed solutions to the aforementioned JOPA extension with a more detailed description on the actual implementation of one of them and finally an account of how the correctness of the new attributes was tested.

Keywords: Semantic technologies, Persistence, Object-ontology mapping, SPARQL

Title translation: SPARQL-based attribute mapping in JOPA

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

Ontologie a sémantická data jsou v informatice již dlouho zavedené pojmy, přesto se zdá, že jejich využití v praxi je rezervováno spíše pro účely vědy a výzkumu, zatímco začínající a existující komerční projekty jsou v této oblasti zastoupeny méně [3]. Rozsáhlé zavedení sémantických technologií by přitom mělo pozitivní dopad na analýzu dat v mnoha oblastech včetně zdravotnictví [20], historie [18] nebo studia a výuky [21].

Dalším příkladem širšího využití sémantických technologií může být myšlenka sémantického webu, jak ji představil Sir Tim Berners-Lee již v roce 1999 [4]. Ačkoliv je potřeba dodat, že web se od té doby v této oblasti výrazně posunul. V roce 2017 již více než třetina URL ze vzorku získaném po vydání služby Common Crawl[1] obsahovala v nějaké formě sémantické trojice. Většinou se však jednalo pouze o technologii HTML Microdata, která se sice podobá RDFa, je ale jednodušší, obecně méně expresivní a nepodporuje stejnou úroveň internacionalizace [19].

Nedostatku nástrojů pro tvorbu informačních systémů v oblasti sémantického webu si všimli doktor P. Křemen a docent Z. Kouba, kteří přišli s metodologií a frameworkem pro návrh a vývoj těchto systémů [12]. Implementací jejich metodologie a frameworku vznikl nástroj JOPA[2] [14]. Ten umožňuje softwarovým vývojářům přistupovat k sémantickým úložištím a datům pomocí dobře známé a rozšířené technologie, kterou Java bezesporu je.

[1] Více informací o této službě lze nalézt na `http://commoncrawl.org/`, navštíveno: 1. 1. 2021.

[2] Tento nástroj bude detailně rozebrán v kapitole Analýza v sekci Knihovna JOPA 3.1

Cílem této práce je rozšířit knihovnu JOPA o novou funkcionalitu: atributy perzistentní entity mohou být definované pomocí SPARQL dotazů. Díky tomu bude možné manuálně ladit mapování ontologií na objekty s velkou přesností.

Práce začíná definováním používaných pojmů (v kapitole 2 - Vysvětlení vybraných pojmů a technologií). Poté bude potřeba zjistit, jak funguje samotná JOPA uvnitř a analyzovat případné existující řešení v podobném projektu (tím se zabývá kapitola 3 - Analýza). S těmito znalostmi bude možné připravit návrh jednoho či více možných řešení (popsaných v kapitole 4 - Návrh řešení). Nakonec bude vybrané řešení v knihovně JOPA implementováno (kapitola 5 - Implementace) a otestováno (kapitola 6 - Testování).

Kapitola 2

Vysvětlení vybraných pojmů a technologií

Tato kapitola obsahuje definice pojmů používaných ve zbytku práce a popis technologií, které s prací úzce souvisí.

2.1 Sémantické technologie

2.1.1 Ontologie

Z knihy A Semantic Web Primer [2] (parafrázováno): Termín *ontologie* pochází z filosofie. V tom kontextu je používán jako název filosofické disciplíny, konkrétně, studium podstaty existence, což je odvětví metafyziky zabývající se identifikací, obecně takových věcí, které skutečně existují a jak tyto věci popsat. Například pozorování, že svět je stvořen z konkrétních objektů, které mohou být uskupeny do abstraktních tříd v závislosti na sdílených vlastnostech[1] je typickým ontologickým závazkem.

V posledních letech však slovo *ontologie* dostalo v počítačové vědě nový, technický význam, který se od toho původního výrazně liší. Všeobecně uznávaná definice od T. R. Grubera [10] zní: *An ontology is an explicit specification of a conceptualization.* (V překladu: *Ontologie je výslovnou specifikací chápání.*)

[1] To je také základní princip celého objektově orientovaného programování.

Další, možná lépe pochopitelná definice od stejného autora se nachází v Encyklopedii databázových systémů [17] (pro potřeby této práce postačí první dvě věty z definice): *Ontologie definuje množinu reprezentačních primitiv, se kterými lze modelovat doménu znalostí či diskursu. Reprezentační primitiva jsou typicky třídy (nebo množiny), atributy (nebo vlastnosti) a vztahy (nebo relace mezi členy tříd).*

2.1.2 RDF

Z knihy A Semantic Web Primer [2] (parafrázováno): RDF (Resource Description Framework) je v podstatě datový model. Aby mohl být abstraktní datový model RDF reprezentován a přenášen, dostal konkrétní syntax v XML, i když jiné syntaktické reprezentace RDF jsou také možné. RDF je nezávislý na doméně a je na uživatelech, aby definovali vlastní terminologii v jazyce RDF Schema (RDFS), který definuje slovní zásobu používanou v RDF data modelech. Kromě toho v RDFS specifikujeme, které vlastnosti patří ke kterým druhům objektů a kterých hodnot mohou nabývat, a popisujeme vztahy mezi objekty.

Základními koncepty RDF jsou tak zvané “zdroje” (resources), “vlastnosti” (properties) a “tvrzení” (statements).

Zdroj je jakýkoliv objekt, nebo věc o které chceme uchovávat nějaké informace. Vždy má unikátní identifikátor nazývaný URI (Uniform Resource Identifier), nebo IRI (Internationalized Resource Identifier). Speciálním typem URI je URL (Uniform Resource Locator), ale URI může být mnohem obecnější v oblastech mimo web.

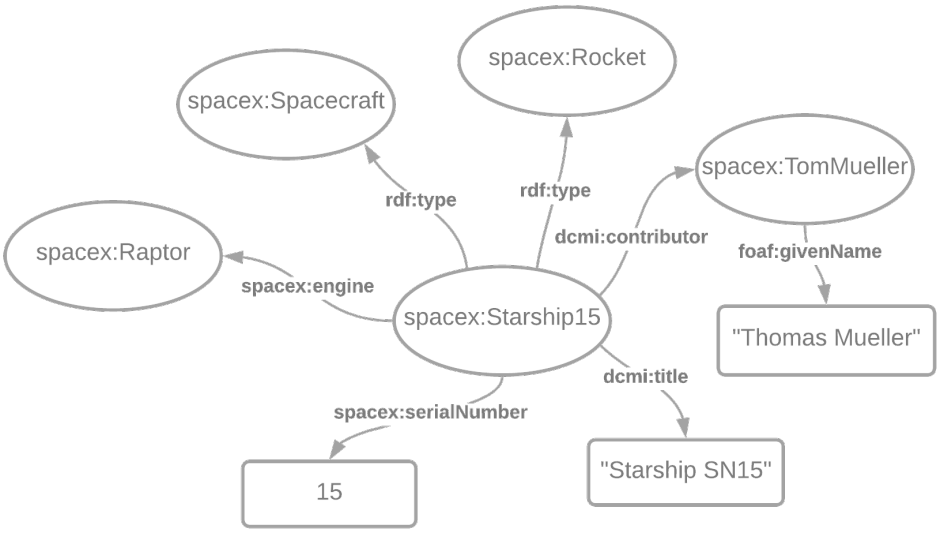
Vlastnost je speciální druh zdroje, který popisuje vztah mezi dvěma zdroji. Také má svůj unikátní identifikátor ve formě URI/IRI, díky čemuž můžeme mít globální schéma unikátně pojmenovaných vlastností, což řeší problém s homonymy.

Tvrzení je trojice subjekt-predikát-objekt (subject-predicate-object), kde subjekt je zdroj, predikát je vlastnost a objekt je buďto zdroj, nebo konkrétní hodnota (např. číslo, řetězec), které se říká literál.

Příklad RDF grafu

Na soubor RDF trojic se lze dívat také jako na orientovaný graf, kterému se potom říká RDF graf. Hrany tohoto grafu představují predikáty. Uzly ze kterých vychází orientovaná hrana jsou subjekty, zatímco uzly do kterých vede orientovaná hrana odpovídají objektům. Můžeme si všimnout, že jeden uzel může být subjekt vzhledem k jednomu predikátu a objekt vzhledem k jinému. To je naprosto v pořádku a stejná situace nastává, pokud použijeme v psaném RDF stejný zdroj ve dvou tvrzeních, jednou jako subjekt a podruhé jako objekt. Uzly obsahující literály se často kreslí jako obdélníky.

Pro účely následujícího příkladu předpokládejme, že společnost SpaceX bude chtít pomocí RDF vytvořit datový model pro informace o svých raketách. Obrázek 2.1 ukazuje, jak by mohla vypadat část tohoto datového modelu reprezentovaná jako RDF graf.



Obrázek 2.1: Příklad RDF grafu

Prefixy používané u zdrojů a vlastností jsou zkratky pro jmenné prostory, ve kterých jsou používané zdroje a vlastnosti definované. V uvedeném příkladě se jedná o prefixy *spacex* = `http://spacex.com/terms/` (toto je hypotetický, neexistující jmenný prostor, kde by byly definované použité zdroje a vlastnosti), *dcmi* = `http://purl.org/dc/terms/` (DCMI je zkratka pro Dublin Core Metadata Initiative[2], tento jmenný prostor nabízí množství předdefinovaných pojmů, které lze při vytváření datového modelu v RDF využít), **foaf** = `http://xmlns.com/foaf/0.1/` (FOAF je zkratka pro Friend

[2]Více na adrese `https://dublincore.org/`, navštíveno 19. 5. 2021.

Of A Friend[3], jenž nabízí předdefinované pojmy sloužící zejména pro popis osob) a nakonec **rdf** = `http://www.w3.org/1999/02/22-rdf-syntax-ns#` (obsahuje nejzákladnější zdroje a vlastnosti v RDF).

2.1.3 SPARQL

SPARQL je rekurzivní akronym pro "SPARQL Protocol and RDF Query Language". Aktuální verze je 1.1.

Z oficiální dokumentace [9]: *SPARQL 1.1 je množina specifikací, které poskytují jazyky a protokoly k dotazování a manipulaci obsahu RDF grafů na webu, nebo v RDF úložišti.*

V této práci budu nadále používat pojem *SPARQL* ve významu *SPARQL 1.1 Query Language for RDF*, tj. dotazovací jazyk pro RDF jehož syntaxe a sémantika je definovaná ve specifikaci [11].

Základní dotazování ve SPARQL

SPARQL funguje na principu porovnávání grafových vzorů. K tomu využívá trojice odpovídající RDF trojicím, ale místo RDF výrazu (zdroje, vlastnosti, nebo hodnoty) se v subjektu, predikátu či objektu může nacházet proměnná, syntakticky značená symbolem `?`.

Následující příklady ve SPARQL jsou vypůjčeny z knihy A Semantic Web Primer [2].

Listing 2.1: Příklad SELECT dotazu ve SPARQL.

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT ?c
4 WHERE {
5    ?c rdf:type rdfs:Class .
6 }
```

[3]Více na adrese `http://xmlns.com/foaf/spec/`, navštíveno 19. 5. 2021.

Výsledkem dotazu ve výpisu 2.1 jsou všechny objekty typu `rdfs:Class`, tj. všechny třídy. Klíčové slovo `PREFIX` slouží k deklaraci zkrácených prefixů pro jmenné prostory.

Struktura `SELECT ... FROM ... WHERE {...}` je obdobná jako u SQL dotazů. `SELECT` specifikuje počet a pořadí hledaných dat. Za `FROM` můžeme specifikovat odkud se mají data brát, tuto část dotazu můžeme vynechat, v takovém případě se předpokládá celá knowledge-base našeho systému. Dovnitř `WHERE` píšeme již zmíněné trojice fungující jako omezení, která musí vrácená data splňovat.

Listing 2.2: Příklad implicitního joinu ve SPARQL

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX uni: <http://www.mydomain.org/uni-ns#>
3 SELECT ?x ?y
4 WHERE {
5    ?x rdf:type uni:Lecturer .
6    ?x uni:phone ?y .
7 }
```

Dotaz ve výpisu 2.2 vrací instance třídy `uni:Lecturer` a jejich telefonní čísla. Skrze proměnnou `x` zde dochází k implicitnímu spojení mezi přednášejícími a telefonními čísly, což má za důsledek, že dotaz nevrátí všechna telefonní čísla, ale pouze ta, která patří přednášejícím.

Více SPARQL ukázek a možností (včetně filtrů, agregačních funkcí a dalších druhů joinů) lze nalézt ve stručné referenční příručce [6].

2.2 Pojmy z oblasti JPA/JOPA

- **Java Persistence API (JPA)** - Java API pro správu dlouhotrvajícího a objektově/relačního mapování s Java EE a Java SE. [5]
- **Entita** - Perzistentní doménový objekt. [5]
- **Entitní třída** - Třída definující entitu.[4]
- **Atribut** - Instanční proměnná entity.

[4] Technické vlastnosti jsou pod pojmem *Entity class* popsány ve specifikaci JPA [5].

- **Objektový model** - Soubor entitních tříd, které jsou určitým způsobem mapovány na databázové entity (např. tabulky v SQL, nebo zdroje v RDF).
- **CRUD operace** - CRUD je anglická zkratka pro čtyři základní operace nad daty. Těmito operacemi jsou: *Create* (vytváření), *Read* (čtení), *Update* (aktualizování) a *Delete* (mazání).
- **Data Access Objects (DAO)** - Soubor objektů obsahujících metody, které implementují CRUD operace nad Objektovým modelem.
- **Named Query** - Uživatelem předem připravený a pojmenovaný dotaz v povoleném dotazovacím jazyce (v případě JOPA se jedná o SPARQL a SOQL), který lze opakovaně použít pro detailnější dotazování do databáze na úrovni DAO.
- **Persistence Context** - Množina spravovaných instancí entit ve kterých pro jakoukoliv perzistentní identitu[5] entity existuje unikátní instance entity. [5]
- **Managed Entity Instance** - Instance s perzistentní identitou, která je aktuálně spojená s perzistentním kontextem. [5]
- **Persistence Provider** - Implementace perzistentního API (např. JPA) typicky poskytovaná třetí stranou. V případě JOPA je implementace již součástí knihovny.

[5]Perzistentní identita není v JPA specifikaci jasně definovaná. Můžeme předpokládat, že se pod tímto pojmem myslí databázová entita.

Kapitola 3

Analýza

Prvním úkolem této analýzy je popsat činnost nejdůležitějších částí knihovny JOPA a její celkovou architekturu z pohledu čtení dat. V další sekci se podíváme, zda by bylo možné se inspirovat existujícím řešením v podobném projektu s názvem AliBaba.

3.1 Knihovna JOPA

JOPA je zkratka pro Java OWL Persistence API. Jedná se o rozhraní a zároveň implementaci objektově-ontologického mapovacího nástroje pro jazyk Java.

3.1.1 Ukázka práce s entitami a atributy

Výpis 3.1 ukazuje definici dvou entitních tříd v JOPA. Entitní třída `Book` představuje entitu knihy v sémantickém úložišti a entitní třída `Author` entitu autora. Tyto dvě entity mají mezi sebou jednosměrný vztah, protože kniha obsahuje referenci na autora. Autor má navíc atribut `name` představující jeho jméno.

Listing 3.1: Ukázka JOPA entity bez mapovaných atributů.

```
1  @OWLClass(iri = "http:\\example.com\ontology\entities#Book")
2  public class Book {
3
4      @Id
5      private URI uri;
6
7      @OWLObjectProperty(iri = "http:\\example.com\ontology\attributes#author")
8      private Author author;
9
10     // gettery a settery
11 }
12
13 @OWLClass(iri = "http:\\example.com\ontology\entities#Author")
14 public class Author {
15
16     @Id
17     private URI uri;
18
19     @OWLDataProperty(iri = "http:\\example.com\ontology\attributes#name")
20     private String name;
21
22     // gettery a settery
23 }
```

Tento jednoduchý příklad bych rád využil k demonstraci výhody zavedení nového atributu založeného na SPARQL dotazu. Datový model definovaný ve výpisu 3.1 umožňuje sice jednoduše zjistit autora libovolné knihy, avšak v případě, že bychom chtěli najít seznam všech knih, které napsal jeden konkrétní autor, by to už tak jednoduchý úkol nebyl. Jedním z řešení tohoto problému je připravit dotaz v jazyce SPARQL, který pro vybraného autora vrátí všechny jeho knihy. Co kdybychom ale zároveň chtěli, aby byl tento seznam knih součástí objektu `Author`, jenomže bychom kvůli tomu nechtěli měnit strukturu datového úložiště (například vytvářet nové tvrzení v RDF)? Odpovědí je již zmíněný koncept atributu založeného na SPARQL dotazu, který budeme dále označovat novým pojmem *query atribut*. Tento atribut bychom mohli jednoduše přidat do entitní třídy `Author` spolu s připraveným SPARQL dotazem a vyřešili bychom tím všechny zmíněné problémy.

3.1.2 Rozhraní a implementace

Rozhraní, definované v modulu `jopa-api`, vychází ze specifikace JPA. Na rozdíl od JPA, která ji tradičně přenechává třetím stranám, je součástí JOPA i implementace zmíněného rozhraní nacházející se v modulu `jopa-impl`.

Jednou z mála nevyužitých funkcionalit JPA je tzv. *property access*, to znamená, že k perzistentnímu stavu entity může být Persistence Providerem

přistupováno pomocí *property accessorů* [5], tj. getterů a setterů. JOPA využívá pouze tzv. *field access* (přístup skrz instanční proměnné entity [5]). Na druhou stranu obsahuje JOPA více vlastností, které se v JPA nevyskytují[1], jedná se především o dodatky v metamodelu potřebné k úspěšnému objektově-ontologickému mapování.

3.1.3 Metamodel

Metamodel je v JOPA i JPA stěžejní částí knihovny. Obsahuje popis objektového modelu, který slouží k mapování na perzistentní entity, kterými jsou SQL tabulky v případě JPA a RDF trojice v případě JOPA, a opětovnému sestavení objektového modelu při čtení z databáze. Primární funkcí metamodelu je tedy držet informace (také můžeme říci metadata) o objektovém modelu, který definoval uživatel. K tomu využívá speciální strukturu, ve které existují pro entity i jejich atributy popisné třídy. V JOPA je to pro entity primárně třída `ManagedType` a pro atributy `FieldSpecification` a od ní dědící `Attribute`. Vytváření a udržování metamodelu je základním požadavkem pro každou implementaci persistentního API.

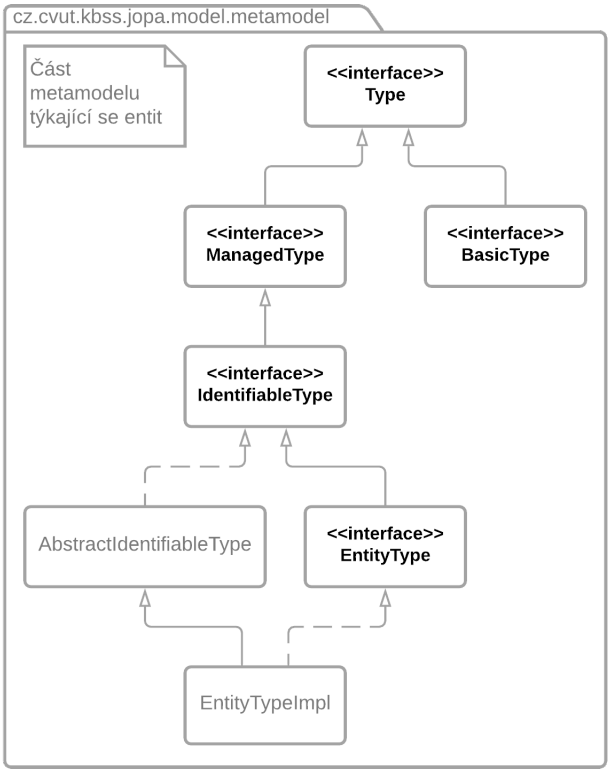
Detailní pohled na to, jakým způsobem metamodel reprezentuje různé datové typy, se kterými JOPA pracuje, nabízí UML class diagram 3.1 (Definice entit v metamodelu). Z diagramu je především patrné rozdělení tzv. spravovaných typů (rozhraní `ManagedType`) a základních typů (rozhraní `BasicType`).

Základním typem se myslí takové Java typy, které JOPA podporuje a nejsou definované uživatelem jako entity. Alternativně můžeme tyto typy hromadně nazývat jako literály.

Příklady základních typů:

- String
- Integer
- Double
- Boolean

[1] Ty jsou pak označeny anotací `@NonJPA`, respektive `@UnusedJPA` pro nevyužívané části JPA.



Obrázek 3.1: Definice entit v metamodelu

Spravované typy jsou naopak uživatelem definované třídy, jejich účelem je reprezentovat entity. V principu si tyto typy lze také představit jako reprezentace RDF zdrojů v objektově orientovaném světě.

Příklady spravovaných typů:

- Book (Typ pro entitu knihy.)
- Author (Typ pro entitu autora.)

Při vytváření typu entity (respektive spravovaného typu) je možné (a často žádoucí) definovat atributy. V kontextu JPA a relačních databází představují atributy entit sloupečky v databázi, zatímco v případě JOPA a ontologií definovaných pomocí RDF lze atribut přirovnat spíše k RDF vlastnosti. Tyto atributy mohou být deklarované jak pomocí základních typů, tak s využitím již definovaných entitních (spravovaných) typů. V druhém případě se zároveň jedná o definici vztahu mezi entitou, ve které atribut deklarujeme a entitou, jejíž typ jsme použili jako typ atributu.

Toto rozdělení typů atributů bude nutné pohlídat také u nových atributů mapovaných na SPARQL dotazy.

3.1.4 OntoDriver

Připojení k úložišti sémantických dat pro JOPA zajišťuje tzv. OntoDriver, který jeho autoři popsali následujícím způsobem: *"OntoDriver je softwarová vrstva navržená pro sjednocení přístupu k různým ontologickým úložištím. Tohoto cíle dosahuje tím, že pro JOPA prezentuje jedno API a umožňuje implementaci používat jakýkoliv framework je vyžadován úložištěm, které se pod ní skrývá, např. Sesame API pro Sesame úložiště nebo Jena pro SDB*[2]*. V tomto ohledu je OntoDriver podobný JDBC*[3] *driveru známém z relačního světa."*[15].

Rozebírat vnitřní fungování OntoDriveru by bylo mimo rámec této práce, důležité je pouze to, že navenek nabízí právě jedno rozhraní, na které se JOPA může připojit.

Seznam podporovaných OntoDriverů:

- Jena
- OWL API
- Sesame

3.1.5 Architektura

Knihovna JOPA má vrstevnatou architekturu danou tím, jak uložená data prochází jednotlivými vrstvami z databáze k uživateli a naopak.

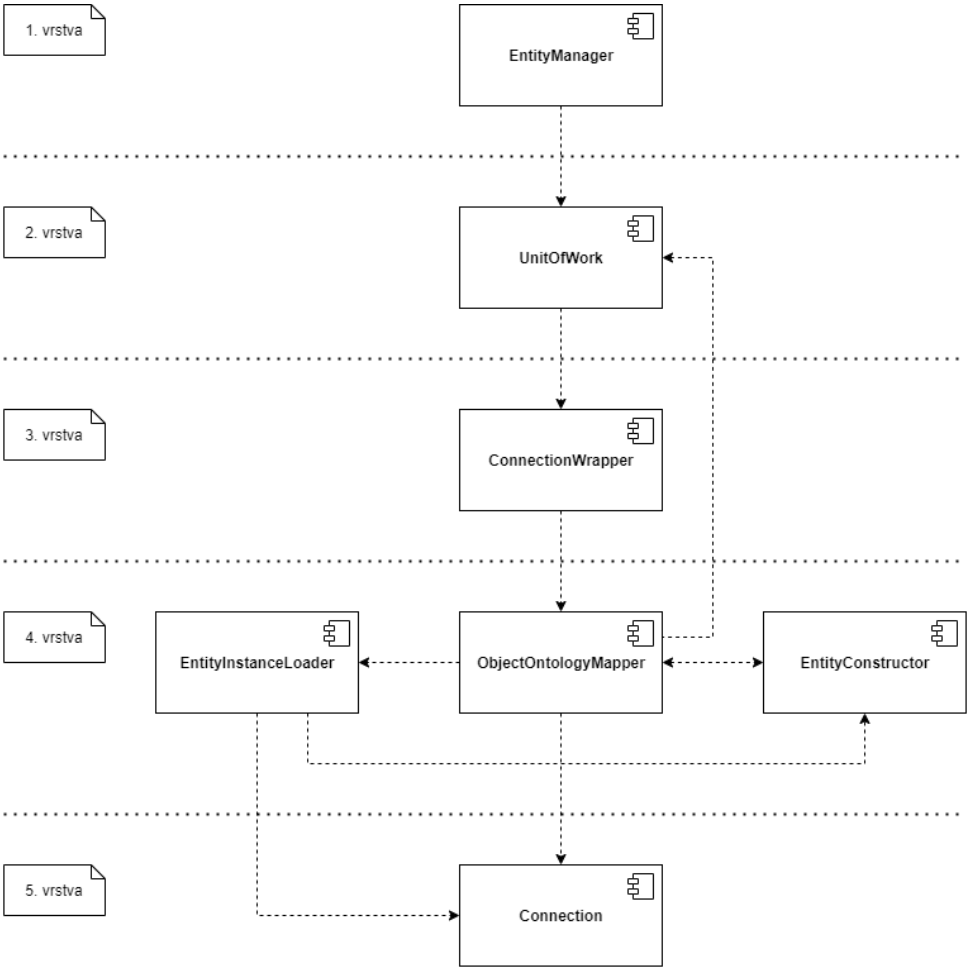
Z pohledu čtení dat z databáze, který je pro potřeby této práce zvlášť vhodný, je nejnižší vrstvou Connection. Ta totiž komunikuje přímo s vybraným OntoDriverem. Data z této vrstvy vychází ve formě Axiomů. Axiom je

[2]Persistentní RDF úložiště vyvinuté firmou Apache.

[3]Java Database Connectivity je API, které poskytuje univerzální přístup k datům z programovacího jazyku Java.[1]

jednoduchá datová struktura reprezentující RDF trojici. Tyto axiomy využívá Entity Constructor ve vyšší vrstvě, který z nich pomocí metamodelu zkonstruuje původní entity objektového modelu. Vytvořené instance entit se dále registrují u komponenty Object Ontology Mapper, která se více využívá při zápisu dat. Nad ní se nachází Unit of Work (princip této komponenty pochází z knihy Patterns of Enterprise Application Architecture [7]). V JOPA je to komplexní vrstva reprezentující aktuální persistence context, který využívá poslední a nejvyšší vrstva: Entity Manager. Ten slouží jako prostředník mezi uživatelem a funkcemi knihovny. Skrz něj provádí uživatel například operace pro ukládání, získávání, aktualizování a mazání dat nebo spravuje transakce.

Obrázek 3.2 (Vrstvy v JOPA) představuje vysokoúrovňový pohled na komponenty a jejich závislosti využívané při čtení dat z databáze v knihovně JOPA a demonstruje její vrstevnatou architekturu.



Obrázek 3.2: Vrstvy v JOPA

3.2 Existující řešení v projektu AliBaba

Podle popisu v projektovém readme se AliBaba definuje jako RDF aplikační knihovna pro vývoj komplexních aplikací s RDF úložištěm [16]. AliBaba nabízí podobnou funkcionalitu se SPARQL dotazy, o kterou usilujeme v knihovně JOPA. Liší se ovšem v tom, že SPARQL dotazy jsou v aplikaci vkládány formou anotace k metodám v tzv. Object Repository (obdoba DAO), nikoliv mapovány přímo na atributy tříd v objektovém modelu, čehož chceme dosáhnout v knihovně JOPA.

Použití anotace se SPARQL dotazem nad metodou ve vyšší vrstvě přináší možnost přidání dynamických parametrů k dotazu pomocí dodatečné anotace `@Bind` aplikované na parametry metody.

Ukázka práce s anotacemi `@Sparql` a `@Bind` definovanými v knihovně AliBaba se nachází ve výpisu 3.2.

Listing 3.2: Příklad metody s anotací Sparql v knihovně AliBaba.

```
1 // vyhleda objekt Person podle jmena za pomoci dotazu
2 @Sparql("PREFIX foo:<http://example.com/resources/>\n" +
3         "SELECT ?person WHERE {?person foo:name $name}")
4 public Person findPersonByName(@Bind("name") String name) {
5     return null;
6 }
```

Od knihovny AliBaba jsme převzali některé syntaktické prvky, konkrétně pojmenování anotace `@Sparql` a parametru `$this`[4], který představuje instanci entity, ve které se atribut nachází.

[4] Znak `$` byl v JOPA zaměněn za `?`, se kterým již pracuje existující implementace rozhraní `QueryParser` pro parsování SPARQL dotazů.

Kapitola 4

Návrh řešení

Z analýzy vnitřní struktury knihovny JOPA vyplynula dvě možná řešení. Tato kapitola nabízí jejich stručný popis a zdůvodnění výběru mezi nimi.

4.1 Řešení 1: Query Manager

Toto řešení je inspirované životním cyklem *named queries* v knihovně JOPA. Pojmenované SPARQL dotazy definované v objektovém modelu pomocí anotací `@NamedNativeQuery` a `@NamedNativeQueries` jsou po přečtení jim určeným procesorem uloženy do objektu `NamedQueryManager`. V něm jsou uchovávány dokud je není potřeba vykonat, v tom případě jsou odtud předány do části knihovny JOPA zabývající se jejich parsováním a vykonáváním (dále pro potřeby této práce označovaná jako továrna na SPARQL dotazy).

V případě SPARQL dotazů mapovaných k atributům by byl postup podobný. Po přečtení specializovaným procesorem by byly uloženy do nového query managera, který by si vedle samotného dotazu držel informaci o připojeném atributu ve formě objektu `java.lang.reflect.Field`. Po vykonání dotazu v továrně na SPARQL dotazy by byl jeho výsledek do připojeného atributu vložen.

Výhodou tohoto řešení je jeho relativní jednoduchost a absence výrazného zásahu do metamodelu. Jedná se však o zkratkovité řešení, kde atributy

založené na SPARQL dotazech se pouze tváří, že jsou součástí metamodelu, ačkoliv ve skutečnosti metamodel obcházejí.

4.2 Řešení 2: Úprava metamodelu

Upravení metamodelu v JOPA znamená provést změny v rozhraní knihovny (modul `jopa-api`) a následně tyto změny implementovat v modulu `jopa-impl`.

Protože atribut definovaný SPARQL dotazem se podstatně liší od stávající definice atributu v metamodelu, je potřeba pro něj vytvořit nové rozhraní `QueryAttribute`. To s klasickým atributem sdílí informace o typu entity, ve které je deklarován a jiné základní informace sloužící pro práci s atributy v jazyce Java.

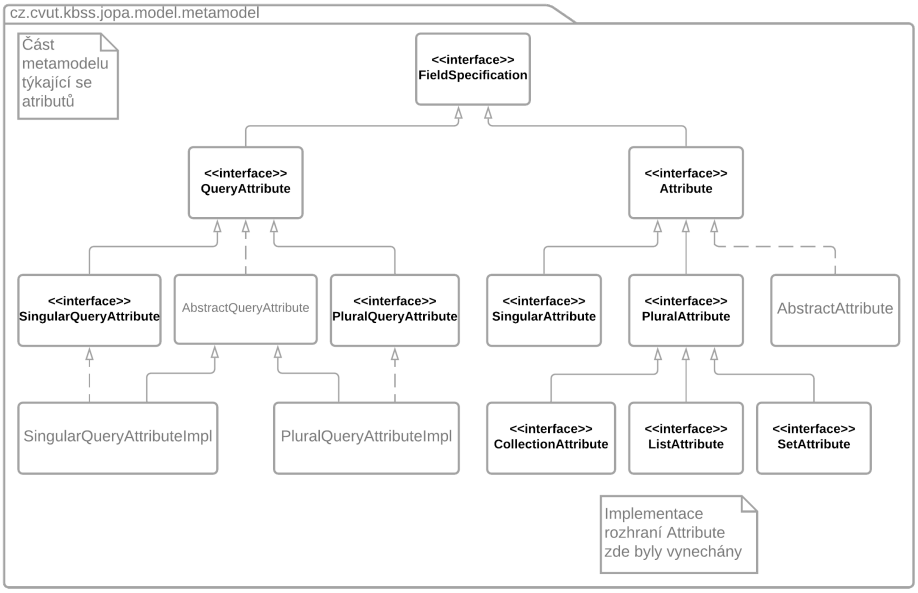
Nejdůležitější částí rozhraní `QueryAttribute` je metoda `getQuery()`, která vrací celý dotaz ve formě řetězce tak, jak byl k atributu zadán uživatelem v objektovém modelu. V rámci této práce jsou uvažovány dva druhy SPARQL dotazů. Prvním jsou dotazy začínající klíčovým slovem `SELECT`, od kterých se očekává, že jejich vykonání vrátí jako hodnotu nějaká data, nebo reference na další entity. Druhým uvažovaným typem SPARQL dotazů jsou dotazy začínající klíčovým slovem `ASK`, které slouží k pokládání takových dotazů, jejichž odpověď je pouze binární, tedy ano/ne[1]. V obou případech SPARQL dotaz jasně definuje obsah atributu, který je tím pádem určen pouze pro čtení (read-only), protože zápis nějaké jiné hodnoty do tohoto atributu by porušil mapování na dotaz a nedává tedy smysl.

Query atribut se také, podobně jako klasický atribut, dále dělí na singular query attribute a plural query attribute. Při implementaci bude navíc pravděpodobně u Plural verze potřeba rozlišovat Java typy `Collection`, `List` a `Set`.

UML class diagram 4.1 (Definice atributů v metamodelu) ukazuje, jak je nový query atribut začleněn do existující definice atributu v metamodelu.

Dále je potřeba zajistit správné vytváření této nové části metamodelu, aby byl dotaz uložen a mohl být z metamodelu při tvorbě entity opětovně přečten a vykonán. K vytváření metamodelu slouží v JOPA podpůrné třídy (nazývaná procesory), kde lze také vyčíst uživatelem vložený dotaz do anotace `@Sparql`.

[1] V jazyce Java se tedy jedná o typ `Boolean` s hodnotami `true` a `false`.



Obrázek 4.1: Definice atributů v metamodelu

V Entity Constructoru a jeho pomocných třídách jsou při rekonstrukci entity u všech query atributů jejich dotazy parsovány a vykonány. Výsledek dotazu je následně vložen do atributu rekonstruované entity.

Malou nevýhodou tohoto řešení je nutnost přidání explicitní závislosti (implicitní závislost zde již existovala skrz komponentu Object Ontology Mapper[2]) mezi komponentami Entity Constructor a Unit of Work, za účelem získání přístupu ke `QueryFactory`.

4.3 Závěr

Druhé řešení, ačkoli složitější na provedení, je jednoznačně nadřazené tomu prvnímu, neboť lépe zapadá do celkového návrhu knihovny JOPA, což zjednoduší její údržbu do budoucna. Kromě toho lze jednodušeji rozlišit různé druhy atributů podle typů daných v jejich deklaraci uvnitř entity, což se může hodit k případnému kvalitnějšímu chybovému výpisu pro uživatele. Další výhodou je možnost jednoduché implementace alternativního typů načítání hodnot nových atributů za účelem optimalizace výkonu. Z těchto důvodů bylo druhé řešení vybráno pro implementační část této práce.

[2]Patrné z diagramu 3.2 (Vrstvy v JOPA).

Kapitola 5

Implementace

V této kapitole se detailněji podíváme na různé části implementace query atributů do knihovny JOPA. Cíle kapitoly jsou podat teoretický základ, který může pomoci při pochopení, jak fungují query atributy v JOPA "pod kapotou" a ukázat detaily vybraných zajímavých částí implementace.

5.1 Přehled query atributů

Než se podíváme na různé implementační detaily mapovaných query atributů, bude potřeba si je nejprve vhodným způsobem kategorizovat a také rozhodnout, jaké vlastnosti a parametry těchto atributů se budou při jejich implementaci používat.

5.1.1 Rozdělení query atributů do kategorií

Pro potřeby implementace query atributů v JOPA se nejlépe hodí jejich rozdělení podle uživatelem přiděleného typu.

Všechny atributy, se kterými dokáže JOPA pracovat, lze podle jejich typu rozdělit do následujících kategorií:

1. Singulární (jednotný) atribut se základním typem
2. Singulární (jednotný) atribut s typem entity
3. Plurální (množný) atribut se základním typem
4. Plurální (množný) atribut s typem entity

Vysvětlení jednotlivých kategorií:

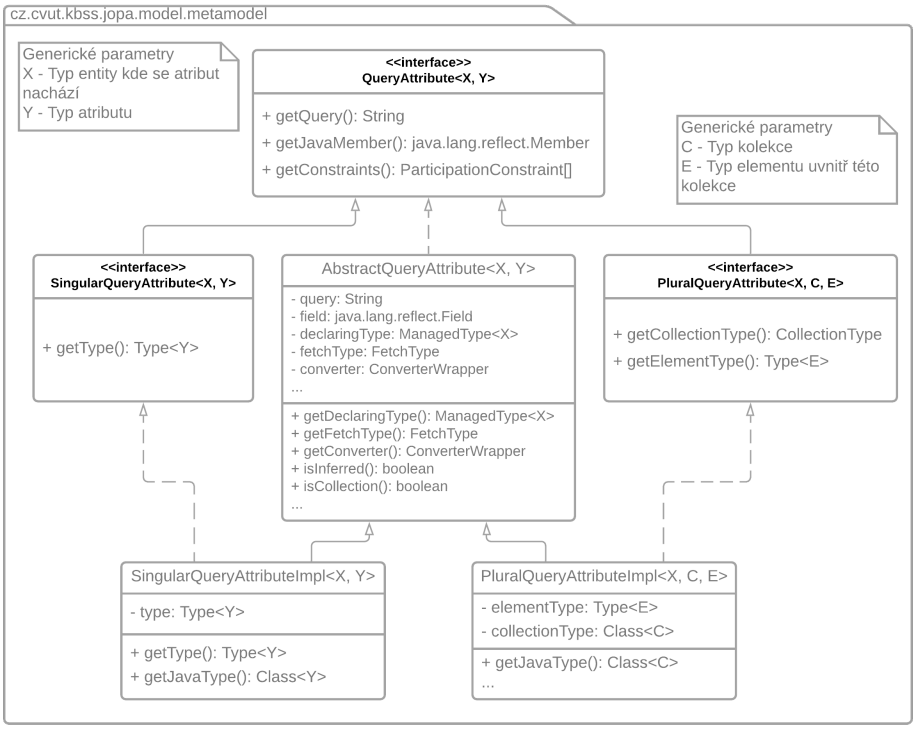
1. Za singulární atribut se v kontextu této práce považuje takový atribut, který obsahuje maximálně jednu jedinou hodnotu[1]. Pod pojmem "základní typ" se myslí především typy definované v balíčku `java.util`, které udržují pouze atomické hodnoty, to znamená například `Integer`, `Double`, `String`[2], `Boolean`.
2. Typ entity je totožný s nějakou entitní třídou, jedná se tedy o referenci na jinou entitu.
3. Plurální atribut obsahuje na rozdíl od singulárního nula až N hodnot. Z toho vyplývá, že takový atribut musí být kolekcí. JOPA dokáže pracovat s následujícími typy kolekcí: `java.util.Collection`, `java.util.List` a `java.util.Set`.
4. Tato kategorie je již pouze kombinací druhé a třetí, to znamená, že se jedná o kolekci entit.

Velká výhoda tohoto rozdělení spočívá v tom, že z pohledu implementace můžeme všechny atributy v jedné kategorii považovat za ekvivalentní. Tyto kategorie tak tvoří třídy ekvivalence nad množinou všech možných atributů.

5.1.2 Vlastnosti query atributů

Nejdůležitější vlastnosti a generické parametry lze vyčíst z UML class diagramu 5.1 (Vlastnosti query atributů). Zajímavostí je například fakt, že query atributy jsou v podstatě vždy *inferred*, což znamená, že mohou být vyvozené z jiných dat. Metoda `isInferred()` je tedy triviální a vrací vždy hodnotu `true`.

[1] Jedna entita se v tomto případě považuje za jednu hodnotu.
[2] Řetězce jsou, stejně jako v relačních databázích, považovány za atomické hodnoty.



Obrázek 5.1: Vlastnosti query atributů

Vysvětlení vybraných deklarovaných metod:

- `getQuery()` - Vrací uživatelem zadaný SPARQL dotaz nad atributem.
- `getConstraints()` - JOPA umožňuje u každého atributu definici tak zvaných *participation constraints*, pomocí kterých lze omezit povolené hodnoty daného atributu. Například nastavit minimální a maximální počet hodnot.
- `getDeclaringType()*` - Vrací entitu, ve které je atribut deklarován.
- `getFetchType()*` - Vrací jednu ze dvou hodnot enumerace `FetchType` (`FetchType.EAGER` nebo `FetchType.LAZY`), která určuje způsob načítání hodnoty do atributu.[3]

Metody označené hvězdičkou (*) pochází z rozhraní `FieldSpecification`, od kterého `QueryAttribute` dědí, jak je vidět na class diagramu 4.1 (Definice atributů v metamodelu)

[3] Těmito způsoby načítání se zabývá sekce 5.4 (Lazy Load v JOPA).

5.2 Úprava rozhraní JOPA

Byla přidána nová anotace `@Sparql`. Příklad jejího použití se nachází ve výpisu 5.1.

Listing 5.1: Příklad atributu s anotací Sparql v knihovně JOPA.

```
1  @OWLClass(iri = SKOS.CONCEPT)
2  public class Term {
3
4      @Id
5      private URI uri;
6
7      @Sparql(query = "PREFIX skos: <http://www.w3.org/2004/02/skos/core#> " +
8              "SELECT ?child WHERE { ?this skos:narrower ?child . }")
9      private Set<TermInfo> children;
10
11     //gettery a settery
12
13 }
```

Vysvětlení kódu ve výpisu 5.1. Řádky 1-2 představují definici entity v Objektovém modelu. Prvním atributem je uri, který plní funkci unikátního identifikátoru entity. Deklarace atributu s anotací `@Sparql` se nachází na řádcích 7-9. Vysvětlení významu tohoto atributu a SPARQL dotazu není pro účely této kapitoly důležité (jedná se o příklad reálného použití query atributu, vytažený z objektového modelu testů na reálných datech). Smyslem této ukázky je pouze prezentovat novou část rozhraní.

Dále bylo v rámci úpravy rozhraní JOPA změněno i vnitřní rozhraní metamodelu podle zvoleného návrhu popsaného v podkapitole 4.2 (Řešení 2: Úprava metamodelu).

5.3 Úprava implementace JOPA

Během implementace nového query atributu do metamodelu trvala snaha o co nejvěrnější zachování návrhu použitého pro klasický atribut. To se týká mimo jiné návrhových vzorů použitých při vývoji. Pěkným příkladem je využití vzoru Strategy [8] pro naplnění různých druhů atributů v Entity Constructoru.

5.3.1 Parsování a spouštění dotazů

Tato část implementace byla vedle úpravy metamodelu bezpochyby tou nejdůležitější, proto zde nabízím poněkud detailnější popis.

K parsování dotazů ve formě řetězce slouží v knihovně JOPA rozhraní `QueryFactory` využívající na pozadí `QueryParser`. Z `QueryFactory` lze získat objekt s rozhraním `TypedQuery`, pomocí kterého je dotaz spuštěn. Výsledek dotazu je následně zkonvertován tak, aby odpovídal typu korespondujícího atributu, a nakonec je do tohoto atributu vložen.

Uvnitř dotazu se ještě může nacházet parametr `?this`, který představuje instanci entity ve které se atribut s dotazem nachází. V takovém případě je tento parametr za využití rozhraní `TypedQuery` před spuštěním detekován a nastaven na identifikátor dané entity.

Sekvenční diagram v UML na obrázku 5.2 ukazuje proces, který provádí Entity Constructor s dotazem u každého atributu (s mapováním na SPARQL dotaz) při vytváření entity, více dopodrobna a ve správné časové souslednosti. Nejdříve získá objekt typu `TypedQuery` a následně zjistí, zda se v tomto dotazu nachází parametr *this*, pokud ano, naplní jej instancí entity, ve které je atribut deklarován a kterou má Entity Constructor k dispozici. Samotné spuštění dotazu a získání výsledku je delegováno na objekty skrývající se pod rozhraním `QueryFieldStrategy`. Pro query atribut existují dvě odlišné základní strategie, jedna pro jednoduchý (singular) atribut a jedna pro plurální (plural) atribut. Za využití stejné strategie je nakonec získaná hodnota vložena do atributu.

Výpis 5.2 ukazuje, jak celý proces naplňování atributů mapovaných na dotazy vypadá v kódu. Metoda `populateAttributes` v Entity Constructoru je volána při každé rekonstrukci entity.

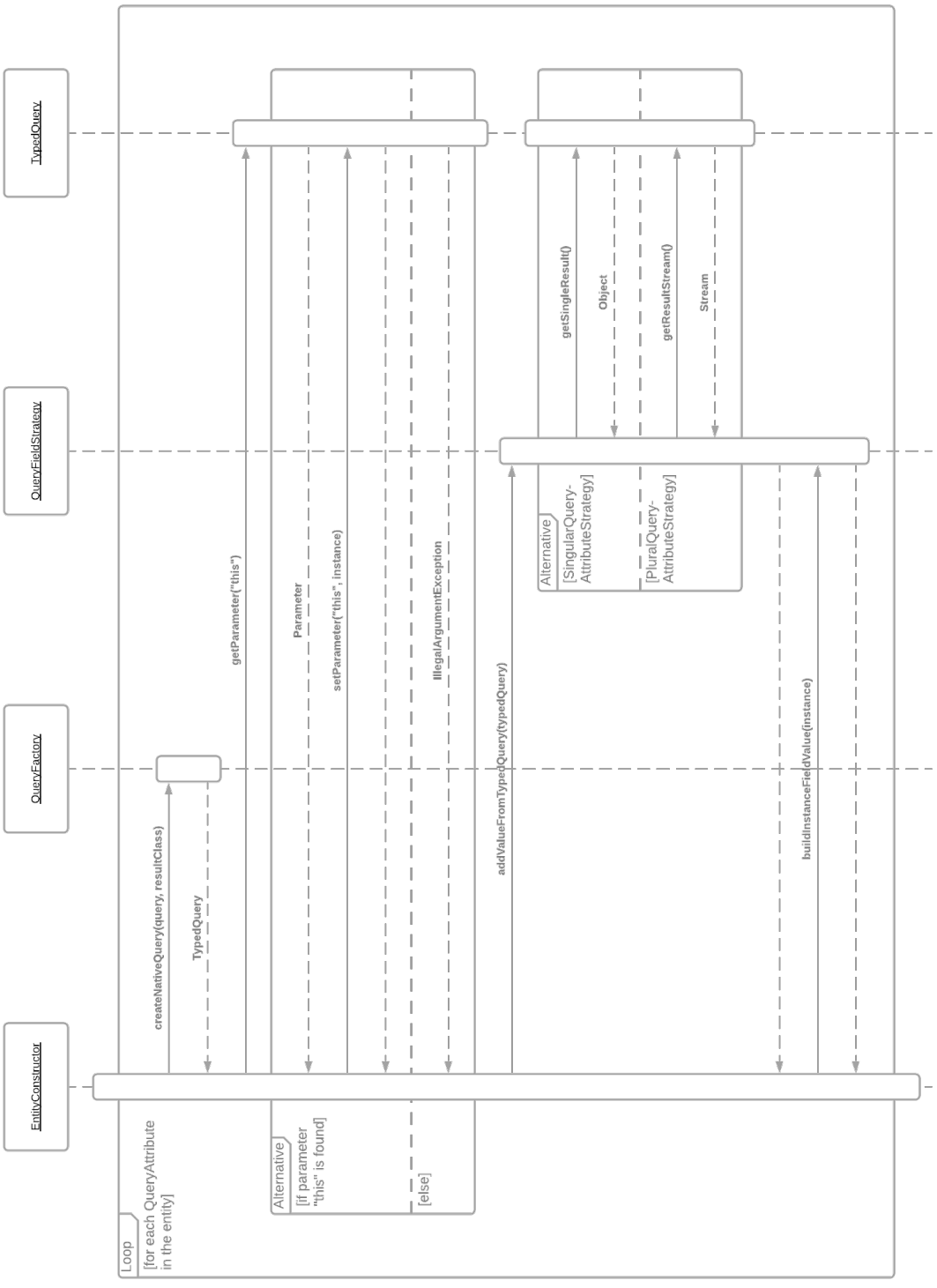
Listing 5.2: Metoda v Entity Constructoru, která naplňuje atributy založené na SPARQL dotazech.

```
1  private <T> void populateQueryAttributes(final T instance, EntityType<T> et)
2              throws IllegalAccessException {
3      final QueryFactory queryFactory = mapper.getUow().getQueryFactory();
4
5      // ziska vsechny query atributy pro danou entitni tridu z metamodelu
6      final Set<QueryAttribute<? super T, ?>> queryAttributes = et.getQueryAttributes();
7
8      for (QueryAttribute<? super T, ?> queryAttribute : queryAttributes) {
9
10         // osetreni vyjimek bylo vypusteno pro lepsi srozumitelnost
11         TypedQuery<?> typedQuery = queryFactory.createNativeQuery(
12                 queryAttribute.getQuery(), queryAttribute.getJavaType());
13
14         // nastavi hodnotu parametru "this", pokud je v dotazu pritomen, instanci entity
15         // osetreni vyjimek bylo vypusteno pro lepsi srozumitelnost
16         typedQuery.setParameter("this", instance);
17
18         QueryFieldStrategy<? extends AbstractQueryAttribute<? super T, ?>, T> qfs =
19                 getQueryFieldLoader(et, queryAttribute); // decides which strategy will be used
20
21         qfs.addValueFromTypedQuery(typedQuery);
22         qfs.buildInstanceFieldValue(instance);
23     }
24 }
```

5.4 Lazy Load v JOPA

Další částí implementace, která stojí za zmínku, bylo přidání alternativního způsobu načítání hodnot atributů, při kterém se hodnota atributu nenačte spolu s entitou, která tento atribut vlastní, ale až ve chvíli, kdy je vyžádána pomocí příslušné getter metody. Tomuto způsobu načítání se obecně říká *Lazy Load* (pojem pochází z knihy Patterns of Enterprise Application Architecture [7]). Hlavní výhodou zmíněného přístupu je, že může být ušetřen čas při načítání entit, zvláště pokud jsou hodnoty vybraných atributů velké či jinak náročné na načtení. V mnoha případech se nám hodí s načtením takových hodnot počkat až budou skutečně potřeba, protože se může dokonce často stát, že nebudou využity vůbec a jejich načítání by tedy bylo úplně zbytečné.

K dosažení výše popsaného efektu se v knihovně JOPA využívá principů aspect-oriented programování. Prakticky to znamená, že se před každým voláním getter metody na nějaký atribut spustí aspekt, který zkontroluje, zda je hodnota daného atributu již načtená. Pokud tomu tak není, provede se její načtení a vložení do pole atributu.



Obrázek 5.2: Práce s dotazy v JOPA

Kapitola 6

Testování

Testování nové funkcionality atributů založených na SPARQL dotazech proběhlo v JOPA na třech úrovních. Na úrovni Unit testů byla otestována pouze nejdůležitější logika, v integračních testech byly otestovány všechny základní případy užití nových atributů a nakonec byla tato nová funkcionalita otestována i na datovém modelu a datech pocházejících z reálného projektu.

6.1 Unit testy

Unit testy slouží primárně k testování netriviální aplikační logiky. Aby bylo v JOPA tohoto cíle dosaženo, využívá se velmi bohatě principu "mockování", což znamená imitace objektů a jejich funkcionality, která tvoří kontext pro testovanou část.

Tato testovací metoda byla využita k otestování správné funkčnosti velmi důležité metody `reconstructEntity` nacházející se ve třídě `EntityConstructor`, část jejíž logiky zabývající se atributy definovanými dotazem již byla diskutována ve výpisu 5.2. V tomto případě bylo konkrétně potřeba namockovat testem využívané části metamodelu a také parsování a spouštění dotazů, protože tyto části knihovny JOPA již nespadaly pod rámec prováděného testu.

6.2 Integrační testy

V integračních testech byla možnost otestovat, zda nový typ atributů definovaných SPARQL dotazy funguje správně ve všech základních případech, ve kterých by je mohl chtít někdo použít. Testování tedy probíhalo podle jednotlivých kategorií query atributů definovaných v 5.1.1 (Rozdělení query atributů do kategorií).

Zde jsou příklady pro každou kategorii, které byly použity v integračních testech:

1. Atribut typu `String`
2. Atribut typu `OWLClassA`, kde třída `OWLClassA` je jiná entita definovaná ve stejném objektovém modelu
3. Atribut typu `Set<String>`
4. Atribut typu `Set<OWLClassA>`

Kromě výše uvedených kategorií definovaných podle typu atributu byly otestovány ještě tyto dva případy:

- SPARQL dotaz typu "ASK".
- Atribut typu `Set<OWLClassA>` jehož parametr `FetchType` je nastavený na hodnotu *Lazy* (výchozí hodnotou testovanou ve všech ostatních testech je typ *Eager*).

Zvláštností SPARQL dotazu typu "ASK" je jeho návratová hodnota, která je vždy typu Boolean, čemuž musí odpovídat i typ atributu, jinak vyhodí JOPA výjimku. V případě *Lazy* atributu bylo otestováno, že jeho hodnota skutečně není nastavena, dokud nebyla zavolána příslušná getter metoda.

Všechny napsané integrační testy se v JOPA spouští celkem třikrát a to jednou na každém z podporovaných OntoDriverů[1].

[1] Seznam podporovaných OntoDriverů lze nalézt v sekci 3.1.4 (OntoDriver)

6.3 Testování na reálných datech

Posledním typem testů jsou příklady poskytnuté vedoucím práce, které reprezentují reálné použití nové funkcionality. Konkrétně se jedná o projekt TermIt[13], který využívá knihovnu JOPA pro přístup k datům v sémantické databázi.

Průběh testování:

1. Pro testovací účely bylo připraveno prostředí, které zajišťuje spojení s databází a poskytuje instance `EntityManager`, pomocí kterých lze přistupovat k testovacím datům s testovacím modelem.

2. Z dodaného slovníku pojmů[2] byly vybrány k otestování 3 pojmy (entity) podle očekávaných hodnot jejich query atributů.

   - Dokumentace
   - Lokalita
   - Stabilizovaná chráněná lokalita

3. Posledním krokem bylo načtení vybraných entit pomocí knihovny JOPA a porovnání hodnot jejich query atributů (získaných řešením načítání query atributů popsaným v této práci) s očekávanými hodnotami.

Deklarace testovaných query atributů se nachází ve výpisu 6.1.

[2] Dostupný z adresy `https://kbss.felk.cvut.cz/termit-dev/#/public/vocabularies/ml-test?namespace=http://onto.fel.cvut.cz/ontologies/slovnik/&includeImported=false`, navštíveno: 30. 4. 2021.

Listing 6.1: SPARQL atributy testované na reálných datech.

```
1  @Namespace(prefix = "pdp", namespace =
     "http://onto.fel.cvut.cz/ontologies/slovn\u00edk/agendov\u00fd/popis-dat/pojem/")
2  @OWLClass(iri = SKOS.CONCEPT)
3  public class Term {
4
5      // ostatni atributy vynechany
6
7      @Sparql(query = "PREFIX skos: <http://www.w3.org/2004/02/skos/core#> " +
8              "SELECT ?child WHERE { ?this skos:narrower ?child . }", fetchType =
                 FetchType.EAGER)
9      private Set<TermInfo> children;
10
11     @Sparql(query = "ASK {
12             ?glossary <http://www.w3.org/2004/02/skos/core#hasTopConcept> ?this .
13         }", fetchType = FetchType.EAGER)
14     private Boolean root;
15
16     // gettery a settery
17
18 }
```

Vysvětlení SPARQL atributů ve výpisu 6.1. Entita `Term` představuje jeden pojem ve slovníku termínů pocházejících z určité ontologie. První mapovaný atribut nazvaný *children* obsahuje množinu pojmů, pro které je tento pojem nadřazeným pojmem. U druhého mapovaného atributu *root* se ptáme, zda je tento pojem nadřazeným pojmem či nikoliv.

Kapitola 7

Závěr

Tato práce představila způsob mapování SPARQL dotazů na atributy entit v knihovně JOPA. Pro lepší porozumění tomuto procesu byly v práci vysvětleny principy fundamentálních sémantických technologií a pojmů spojených s perzistentními API. Za účelem vlastního rozšíření knihovny JOPA o nový druh atributů, jejichž hodnota je závislá na SPARQL dotazu (tzv. *query atributů*), byla provedena analýza této knihovny a existující podobné funkcionality v konkurenční knihovně AliBaba. Dále byla navržena dvě možná řešení, jak tuto funkcionalitu realizovat v JOPA. První navržené řešení "Query Manager" bylo zavrženo ve prospěch druhého řešení "Úprava metamodelu", které umožňuje JOPA metamodelu query atributy rozlišovat od ostatních atributů, díky čemuž je může JOPA naplnit při konstruování entit správnými daty. Toto řešení bylo následně implementováno na úrovních rozhraní JOPA a implementace JOPA. Korektnost implementace byla otestována nejprve pomocí jednotkových a integračních testů a následně i na reálných datech a datovém modelu systému TermIt.

Příloha A

Literatura

[1] The java database connectivity api. `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`. Navštíveno: 1. 5. 2021.

[2] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. MIT press, 2004.

[3] V. R. Benjamins, J. Davies, R. Baeza-Yates, P. Mika, H. Zaragoza, M. Greaves, J. M. Gomez-Perez, J. Contreras, J. Domingue, and D. Fensel. Near-term prospects for semantic technologies. *IEEE Intelligent Systems*, 23(1):76–88, 2008.

[4] Tim Berners-Lee and Mark Fischetti. *Weaving the Web*. HarperCollins, 1999.

[5] L. Demichiel and L. Jungmann et al. Jsr 338: Java persistence 2.2, May 2013.

[6] Lee Feigenbaum. Sparql by example: the cheat sheet. `https://www.iro.umontreal.ca/~lapalme/ift6281/sparql-1_1-cheat-sheet.pdf`, 2008. Navštíveno: 7. 1. 2021.

[7] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[9] W3C SPARQL Working Group. Sparql 1.1 overview. `https://www.w3.org/TR/sparql11-overview/`, March 2013. Navštíveno: 3. 1. 2021.

[10] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[11] Steve Harris and Andy Seaborne. Sparql 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, March 2013. Navštíveno: 3. 1. 2021.

[12] P. Křemen and Z. Kouba. Ontology-driven information system design. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):334–344, 2012.

[13] P. Křemen, M. Ledvinka, M. Med, and L. Saeeda. Termit. `https://github.com/kbss-cvut/termit`, 2021. Navštíveno: 30. 4. 2021.

[14] M. Ledvinka and P. Křemen. Jopa - java owl persistence api. `https://github.com/kbss-cvut/jopa`, 2020. Navštíveno: 3. 1. 2021.

[15] Martin Ledvinka. and Petr Křemen. Jopa: Accessing ontologies in an object-oriented way. In *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, pages 212–221. INSTICC, SciTePress, 2015.

[16] J. Leigh. Alibaba. `https://bitbucket.org/openrdf/alibaba/src/master/README.md`, 2017. Navštíveno: 24. 11. 2020.

[17] Ling Liu and M Tamer Özsu. *Encyclopedia of database systems*, volume 6. Springer New York, NY, USA:, 2009.

[18] Albert et al. Meroño-Peñuela. Semantic technologies for historical research: A survey. *Semantic Web*, 6(6):539–564, 2015.

[19] Chaals McCathie Nevile and Dan Brickley et al. Html microdata - w3c working draft 26 april 2018. `https://www.w3.org/TR/microdata/`, 2018. Navštíveno: 1. 1. 2021.

[20] Robert Piro, Yavor Nenov, Boris Motik, Ian Horrocks, Peter Hendler, Scott Kimberly, and Michael Rossman. Semantic technologies for data analysis in health care. In *International Semantic Web Conference*, pages 400–417. Springer, 2016.

[21] T. Tiropanis, H. Davis, D. Millard, and M. Weal. Semantic technologies for learning and teaching in the web 2.0 era. *IEEE Intelligent Systems*, 24(6):49–53, 2009.

Příloha B

Seznam zkratek

Zkratka	Význam
API	Application Programming Interface
JPA	Java Persistence API
JOPA	Java OWL Persistence API
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
DAO	Database Access Object
URI	Uniform Resource Identifier
IRI	Internationalized Resource Identifier
URL	Uniform Resource Locator
UML	Unified Modeling Language
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JDBC	Java Database Connectivity

Tabulka B.1: Seznam zkratek

Příloha C

Obsah elektronické přílohy

Elektronická příloha této práce má podobu textového souboru s názvem SPARQL_based_attribute_mapping_in_JOPA.txt, který obsahuje dva odkazy. První odkaz[1] vede na větev v GitHub repozitáři, která obsahuje kompletní hotovou implementaci práce uvnitř knihovny JOPA včetně testů na reálných datech. Druhý odkaz[2] vede na pull request vytvořený za účelem přidání této implementace do hlavního projektu JOPA.

[1] `https://github.com/Sidonivs/jopa/tree/thesis-evaluation`, navštíveno: 21. 5. 2021

[2] `https://github.com/kbss-cvut/jopa/pull/88`, navštíveno: 21. 5. 2021