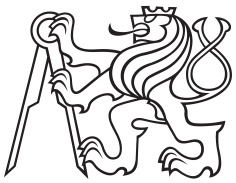


České
vysoké
učení technické
v Praze

Fakulta elektrotechnická
Katedra počítačov

Bakalárska práca

Sémantické Criteria API

Marcel Žec

Vedúci: Ing. Martin Ledvinka
Študijný program: Softvérové inžinierstvo a technológie
Máj 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Žec** Jméno: **Marcel** Osobní číslo: **474414**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Sémantické Criteria API

Název bakalářské práce anglicky:

Semantic Criteria API

Pokyny pro vypracování:

1. Seznamte se s principy sémantického webu (hlavně jazyky RDF, SPARQL) a knihovnou JOPA.
2. Analyzujte programatické dotazovací rozhraní Criteria API a jeho alternativy (např. Querydsl). Zvláštní pozornost věnujte vhodnosti jednotlivých konstruktů pro použití v oblasti sémantického webu.
3. Navrhněte programatické dotazovací rozhraní pro knihovnu JOPA.
4. Naimplementujte vámi navržené dotazovací rozhraní a integrujte jej do knihovny JOPA.
5. Ověřte správnost vaší implementace srovnáním s ekvivalentními dotazy vytvořenými v jazyce SOQL, který je podporován knihovnou JOPA.

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

Poďakovanie

Ďakujem vedúcemu práce, ktorým je **Ing. Martin Ledvinka**, za cenné rady, usmernenia a čas, ktorý mi venoval počas tvorby tejto práce. Poďakovanie patrí aj mojim rodičom za ich neustálu podporu a priateľom, ktorí mi boli oporou počas náročného štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a uviedol všetku použitú literatúru.

V Prahe, 21. mája 2021

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 21, 2021

Abstrakt

Automatizovanie vyhľadávania relevantných informácií na internete je veľmi náročné. Riešením je osvojenie a aplikovanie princípov sémantického webu. Vo svete sémantických dát chýbajú knižnice poskytujúce programovateľné dotazovacie rozhranie, ktorého výhody oceníme hlavne vo veľkých podnikových aplikáciách. Knižnica JOPA sa snaží štandardné postupy a funkcionality zo sveta relačných dát aplikovať vo svete sémantických dát. V práci najprv čitateľa oboznámime s problematikou sémantického webu. Následne analyzujeme existujúce programovateľné dotazovacie rozhrania, využívané vo svete relačných dát. Hlavným prínosom práce je návrh a implementácia programovateľného dotazovacieho rozhrania, ktoré rozširuje knižnicu JOPA. Implementované Semantic Criteria API je inšpirované rozhraním JPA Criteria API. Na preklad využíva objektový dotazovací jazyk SOQL, ktorý je v knižnici podporovaný. Vyhodnotenie správnosti implementácie spočíva v porovnaní výsledkov dotazov vytvorených pomocou Semantic Criteria API a ekvivalentných dotazov vytvorených pomocou SOQL.

Kľúčové slová: sémantický web, sémantické dáta, programovateľné dotazovacie rozhranie, dotazovanie, Java, JOPA, SOQL, JPA, Criteria API, QueryDSL

Vedúci: Ing. Martin Ledvinka

Abstract

Automated search of relevant information on the Internet is complicated. The solution to this problem is to apply semantic web principles. Frameworks that provide programming API for querying, whose benefits are mainly valued in enterprise applications, are missing in the world of semantic data. The JOPA framework strives to apply standard procedures and functionality of the world of relational data in the world of semantic data. At the beginning of this work, we acquaint the reader with the topic of the Semantic Web. Subsequently, we analyze existing programming querying APIs that are used in the world of relational data. The main contribution of this work is the design and implementation of querying API, which expands the JOPA framework. For translation, it uses the object query language SOQL supported by the framework. Evaluation of the correctness of the implementation consists of comparing the results of queries created with Semantic Criteria API and equivalent queries created with SOQL.

Keywords: semantic web, semantic data, programming API, querying, Java, JOPA, SOQL, JPA, Criteria API, QueryDSL

Title translation: Semantic Criteria API

Obsah

Výpisy

Tabuľky

Obrázky

Kapitola 1
Úvod

V dnešnej dobe je život väčšiny ľudí spojený s internetom. World Wide Web sa od svojho vzniku v roku 1990 stal obrovským úložiskom dát rôznych formátov, ktoré sa každým dňom rozširuje. To spôsobuje horšie vyhľadávanie relevantných informácií. Väčšina informácie na internete je navrhnutá tak, aby bola čitateľná pre ľudí. Spracovanie týchto informácií počítačovými programami je preto veľmi obmedzené. V roku 2001 prišiel Tim Berners-Lee, pôvodný tvorca WWW, s myšlienkou, ktorá by tento problém mohla vyriešiť.

Sémantický web je myšlienka, ktorá by mala rozšíriť webové stránky o dáta, ktoré majú byť určené výlučne pre počítače. Počítače budú schopné nájsť význam sémantických dát, vďaka odkazom na definície kľúčových pojmov a pravidiel pre vytváranie logických dedukcií. Výsledná infraštruktúra, ktorá vznikne, podnieti vývoj automatizovaných webových služieb [1].

Veľké množstvo webových stránok alebo aplikácií pracuje s relačnými dátami uloženými v relačných databázach. Pri vývoji týchto aplikácií je potrebné tieto dáta získať dotazovaním na databázu. Vo svete relačných dát existuje dotazovací jazyk SQL. Existuje však aj množstvo knižníc, pomocou ktorých môžeme dotazovať dáta efektívnejším spôsobom, ako písaním nativného dotazovacieho jazyka. Efektívnejšími spôsobmi sú objektové dotazovacie jazyky a programovateľné dotazovacie rozhrania.

Programovateľné dotazovacie rozhrania umožňujú vytváranie dynamických dotazov a často sa využívajú tam, kde štruktúra dotazu nie je dopredu známa a vzniká až za behu aplikácie. Ich hlavnou výhodou je možnosť vytvárať staticky typované dotazy, čo umožňuje odhaľovanie chýb už v čase kompilácie. Túto typovú bezpečnosť oceníme hlavne pri vývoji veľkých podnikových (enterprise) aplikáciách, ktoré takéto dotazovacie rozhrania často používajú [2].

Sémantický web však pracuje s dátami, ktorých formát je značne odlišný od relačných dát. Hlavným nástrojom na dotazovanie takýchto dát je dotazovací jazyk SPARQL. Existujú však aj nástroje, ktoré poskytujú efektívnejšie spôsoby dotazovania dát a vývoja aplikácií či webových stránok, ktoré pracujú so sémantickými dátami. Jedným z týchto nástrojov je knižnica JOPA [3], ktorá ale zatiaľ nedisponuje programovateľným dotazovacím rozhraním. Ciele tejto práce vedú práve k tomu, aby mohla byť knižnica o takéto rozhranie rozšírená.

Ciele bakalárskej práce sú:

- zoznámiť sa s princípmi sémantického webu a knižnicou JOPA
- analyzovať programovateľné dotazovacie rozhranie Criteria API a jeho alternatívy
- navrhnúť programovateľné dotazovacie rozhranie pre knižnicu JOPA
- naimplementovať navrhnuté dotazovacie rozhranie a integrovať ho do knižnice JOPA
- overiť správnosť implementácie porovnaním s ekvivalentnými dotazmi vytvorenými pomocou objektového dotazovacieho jazyku SOQL, ktorý je podporovaný knižnicou JOPA

Kapitola 2
Relevantné technológie

V tejto kapitole sa pozrieme na pojem sémantický web a technológie, ktoré sú s ním spojené. Spomenieme aj technológie, ktoré so sémantickým webom nesúvisia a sú využívané vo svete relačných dát. Pochopenie ich významu a fungovania je veľmi dôležité pre naplnenie cieľov práce.

2.1 Sémantický web

Web, ako ho dnes poznáme, je médium s veľkým množstvom dokumentov. Tieto dokumenty sú vytvárané tak, aby boli čitateľné pre ľudí, a preto je automatizované získavanie informácií veľmi náročné [1].

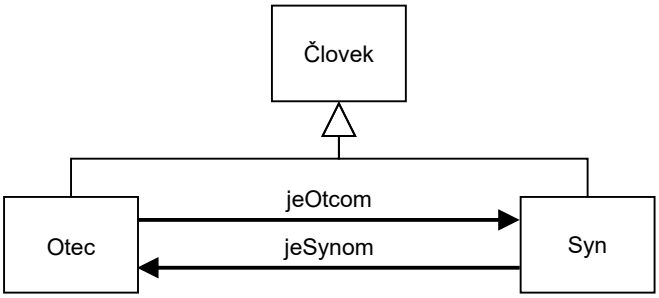
Sémantický web je myšlienka a snaha rozšíriť dokumenty, ktoré tvoria súčasný web o metadáta. Tieto metadáta definujú, o čom informácie v danom dokumente sú, v strojovo spracovateľnej forme. Metadáta spolu s doménovou teóriou v podobe ontológií (viac v sekcii 2.2) umožnia vytváranie oveľa kvalitnejších webových služieb [4]. Budú môcť vznikať veľmi presné automatizované vyhľadávače, ktorým bude stačiť iba zlomok našich poznatkov na nájdenie požadovanej odpovede. A to vďaka schopnosti vyhľadávača pochopiť význam informácií a vzťahy medzi informáciami z rôznych zdrojov [1].

2.2 Ontológia

Výraz ontológia je požičaný z filozofie, kde ontológia systematicky popisuje ľudskú existenciu. V spojený so sémantickým webom, môžeme ontológiu definovať ako explicitnú konceptuálnu špecifikáciu znalostí nejakej záujmovej oblasti [5].

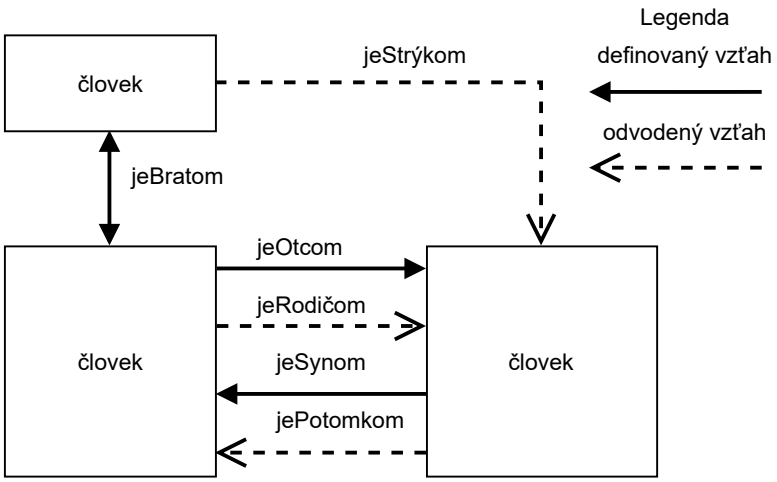
Menej formálne povedané, ontológia definuje množinu reprezentačných primitív, pomocou ktorých modelujeme znalosti o nejakej konkrétnej oblasti. Oblasť znalostí často označujeme ako doména. Pod reprezentačnými primitívami myslíme triedy, atribúty a vzťahy. Definícia primitív obsahuje informácie o ich význame, ale aj obmedzenia a pravidlá na ich používanie a odvodzovanie implicitných znalostí [6].

Pre lepšie pochopenie si ukážeme jednoduchý príklad. Na obrázku 2.1 sme pomocou UML jazyka namodelovali časť znalostí o rodine. Definujeme triedy Otec, Syn a Človek. Definujeme, že triedy Otec, Syn sú podtriedou triedy Človek. Zavedieme vzťah jeOtcom a ten obmedzíme iba na triedy Otec a Syn. Zavedieme aj obrátený vzťah jeSynom a obmedzíme ho v opačnom poradí, teda iba medzi triedami Syn a Otec.



Obrázok 2.1: Vizualizácia znalostí o rodine

Na obrázku 2.2 môžeme vidieť ďalšie obmedzenia ale aj pravidlá slúžiace na odvodzovanie implicitných znalostí. Ak je medzi triedami vzťah jeOtcom, tak je možné logicky odvodiť vzťah jeRodičom. V obrátenom poradí tried to potom platí nasledovné. Ak sú triedy vo vzťahu jeSynom tak môžeme odvodiť vzťah jePotomkom. Ostáva nám posledný vzťah jeStrýkom, ktorý môžeme odvodzovať z existujúcich informácii. Aby sme takýto vzťah mohli logicky odvodiť, je potrebné aby bol daný človek vo vzťahu jeBratom s iným človekom, ktorý je následne vo vzťahu jeOtcom k niekomu ďalšiemu.



Obrázok 2.2: Explicitné a implicitné vzťahy

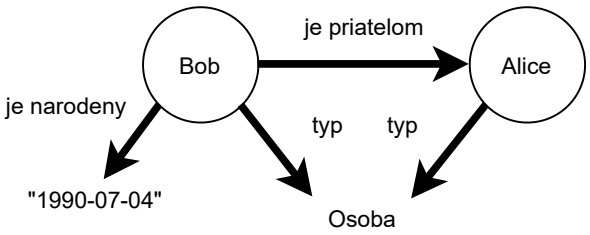
2.3 RDF

V sekcii 2.1 sme hovorili o strojovo spracovateľnom formáte pre metadáta. Pre tieto účely vznikol jazyk RDF (Resource Description Framework), ktorý patrí medzi W3C Recommendations [7] a je teda považovaný za základný štandard sémantického webu.

Jazyk slúži na vytváranie **tvrdení o zdrojoch**. Za zdroj môžeme považovať klasické webové zdroje, teda webové stránky či obrázky, ale aj akúkoľvek fyzickú či abstraktnú vec, o ktorej môžeme vytvoriť nejaké tvrdenie. Zdroj musí byť jednoznačne identifikovateľný pomocou **IRI adresy** [8]. Na vytváranie tvrdení o zdrojoch, sa používajú **RDF trojice** s nasledujúcou štruktúrou:

<subjekt> <predikát> <objekt>

Množinou RDF tvrdení vzniká **RDF graf**. Subjekty a objekty predstavujú uzly grafu, ktoré sú prepojené predikátmi. Subjekty a objekty predstavujú konkrétne zdroje, o ktorých niečo tvrdíme. Predikát označujeme aj ako **vlastnosť** a vyjadruje vzťah, ktorý existuje medzi dvoma zdrojmi [9]. Na obrázku 2.3 môžeme vidieť ukážku RDF grafu, ktorý sa budeme snažiť v ukážkach RDF trojíc namodelovať.



Obrázok 2.3: Graf RDF trojíc [10]

Spomínali sme, že IRI adresou jednoznačne identifikujeme zdroj. Existujú však aj iné možnosti. Zdroj môžeme reprezentovať aj prostredníctvom **literálov**. Literály sa používajú na vyjadrovanie hodnoty. Ide napríklad o čísla, dátumy alebo slová. Za literálom musí nasledovať IRI adresa na jeho dátový typ. Poslednou možnosťou akou vyjadrujeme zdroje je prostredníctvom **prázdnych uzlov**. Ide o lokálne platné identifikátory, a teda používajú sa na lokálne popisovanie zdrojov bez globálneho IRI identifikátora [9].

Na zapisovanie RDF grafov existuje niekoľko rôznych serializačných formátov. Ukážeme si niekoľko formátov zápisu RDF, ktoré odpovedajú grafu na obrázku 2.3. Najstarším formátom je RDF/XML, ktorého ukážku môžeme vidieť na výpise 2.1. V súčasnosti však existujú modernejšie formáty, ktoré umožňujú jednoduchší zápis a zároveň lepšiu čitateľnosť [10]. Na výpise 2.2 vidíme ukážku vo formáte Turtle a na výpise 2.3 vidíme ukážku vo formáte JSON-LD.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
    xmlns:ex="http://example.org/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    <rdf:Description rdf:about="http://example.org/bob#me">
            <rdf:type rdf:resource="http://example.org/classes/Osoba"/>
            <ex:jeNarodeny rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
                1990-07-04
            </ex:jeNarodeny>
            <ex:jePriatelom rdf:resource="http://example.org/alice#me"/>
    </rdf:Description>
</rdf:RDF>
```

Listing 2.1: RDF/XML formát [10]

```
BASE   <http://example.org/>
PREFIX ex: <http://example.org/classes_and_relationships>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

<bob#me>
       a ex:Osoba ;
       ex:jePriatelom <alice#me> ;
       ex:jeNarodeny "1990-07-04"^^xsd:date .
```

Listing 2.2: Turle formát [10]

```
{
      "@context": "example-context.json",
      "@id": "http://example.org/bob#me",
      "@type": "Osoba",
      "jeNarodeny": "1990-07-04",
      "jePriatelom": "http://example.org/alice#me"
}
```

Listing 2.3: JSON-LD formát [10]

2.4 RDF Schema

V sekcii 2.2 sme spomínali, že súčasťou ontológie sú aj obmedzenia a pravidlá. Jazyk RDF je v tomto smere veľmi obmedzený a ponúka len základné stavebné prvky.

RDF Schema je sémantické rozšírenie jazyka RDF. Zavádza mechanizmy na popisovanie skupiny súvisiacich zdrojov a ich vzájomných vzťahov. Ide napríklad o zavedie tried (Class) a podtried (subClassOf) ale aj podvlastnosti (subPropertyOf), rozsahy (range) a domény (domain) [11].

Na výpise môžeme vidieť pseudokód použitia RDFS, v ktorom sme sa snažili čo najlepšie pokryť definíciu tried, vlastností a obmedzení z príkladu o rodine zo sekcie 2.2. Vidíme, že RDFS tvrdenie v skutočnosti odpovedá jednému RDF tvrdeniu.

```
PREFIX ex:<http://example.org/>
PREXIF rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
<!-- define classes -->
    <ex:Osoba><rdf:type><rdfs:Class>
    <ex:Otec><rdf:type><rdfs:Class>
    <ex:Syn><rdf:type><rdfs:Class>
<!-- define subclasses-->
    <ex:Otec><rdfs:subClassOf><ex:Osoba>
    <ex:Syn><rdfs:subClassOf><ex:Osoba>
<!-- define properties -->
    <ex:jeOtcom><rdf:type><rdf:Property>
    <ex:jeSynom><rdf:type><rdf:Property>
<!-- define subproperties -->
    <ex:jeRodicom><rdfs:subPropertyOf><ex:jeOtcom>
    <ex:jePotomkom><rdfs:subPropertyOf><ex:jeSynom>
<!-- define domain and range of property -->
    <ex:jeOtcom><rdfs:domain><ex:Otec>
    <ex:isSynom><rdfs:domain><ex:Syn>
    <ex:jeOtcom><rdfs:range><ex:Syn>
    <ex:jeSynom><rdfs:range><ex:Otec>
```

Listing 2.4: RDF Schema

2.5 OWL2

Aj rozšírenie v podobe RDFS je na vyjadrovanie príliš komplexných vzťahov prikrátke. Príkladom je vzťah jeStrykom zo sekcie 2.2.

OWL 2 Web Ontology Language je sémantický jazyk, navrhnutý na reprezentovanie komplexných znalostí. Ľudia dokážu premýšľaním vyvodzovať nejaké dôsledky zo svojich znalostí a o to isté sa snaží aj OWL2. Naviac sa snaží objavovať informácie, ktoré by človeku mohli uniknúť. Okrem mechanizmov, ktoré už poznáme z RDFS, akými sú triedy a vlastnosti, poskytuje nové mechanizmy. Ide napríklad o disjunkciu a ekvivalenciu tried alebo lepšiu charakterizáciu či špecifickejšie obmedzenia vlastností [12].

Na výpise 2.5 vidíme ukážku pravidiel, zapísanú vo funkcionálnom formáte, pre odvodzovanie implicitných znalostí.

```
SubObjectPropertyOf( :jeRodicom :jePotomkom)

SubObjectPropertyOf(
   ObjectPropertyChain( :jeBratom :jeRodicom)
   :jeStrykom
 )
```

Listing 2.5: OWL2 [12]

Prvé tvrdenie hovorí, že vzťah jeRodicom implikuje opačný vzťah jePotomkom. Druhé tvrdenie na výpise hovorí, že z reťazca vzťahov jeBratom a následne jeRodicom, odvodzujeme vzťah jeStrykom.

Každé tvrdenie vytvorené pomocou OWL2 nazývame **axióm** a v rámci ontológie ho považujeme za pravdivý [12]. Narozdiel od RDFS, sa môže jedno tvrdenie v jazyku OWL2 mapovať na väčšie množstvo RDF tvrdení [13]. Na výpise 2.6 môžeme vidieť mapovanie druhého axiómu, o vzťahu jeStrykom, na RDF tvrdenia.

```
ex:jeStrykom owl:propertyChainAxiom _:stryko.
_:stryko rdf:first ex:jeBratom .
_:stryko rdf:rest _:strykovBrat .
_:strykovBrat rdf:first ex:jeRodicom .
_:strykovBrat rdf:rest rdf:nil .
```

Listing 2.6: OWL2/RDF mapovanie [13]

Identifikátory stryko a strykovBrat predstavujú prázdne uzly a ich názov sme zvolili, aby bolo jednoduchšie pochopiť tento výpis.

Problematika jazyky OWL2 je veľmi rozsiahla a bolo by možné o nej vytvoriť samostatnú prácu, preto nebudeme zachádzať do väčších detailov.

Od vytvárania k dotazovaniu

V predošlých sekciách sme hovorili o technológiách, ktoré slúžia predovšetkým na vytváranie a prácu s dátami. Pri vývoji aplikácií alebo webových stránok je veľmi často dôležitejšie pristupovanie a získavanie dát. Preto sa teraz pozrieme na to, ako sa sémantické dáta dotazujú. Pozrieme sa ale aj na štandardy využívané vo svete relačných dát a na to, ako ich môžeme aplikovať na sémantické dáta.

2.6 SPARQL

Na dotazovanie dát vo formáte RDF existuje dotazovací jazyk SPARQL. Väčšina foriem dotazov jazyka SPARQL obsahuje množinu trojíc, podobnú RDF trojiciam. Nazývame ich **základné vzory grafu** a oproti RDF trojiciam, môže byť každý prvok (teda subjekt, predikát alebo objekt) premennou [14].

Na výpise 2.7 si ukážeme SPARQL dotaz. Budeme používať prefixy, ktoré umožňujú ukryť celú IRI adresu za menný priestor, a tak udržať dotaz prehľadný. Ide o prefix rdf na vyjadrenie typu, xsd na vyjadrenie dátových typov literálov a music, z ktorého čerpáme vymyslené ukážkové triedy.

Prvá časť dotazu predstavuje projekciu a tvorí sa klauzulou SELECT. Následne definujeme premenné, ktoré chceme získať. Premenné sa označujú otáznikom a ich názvom (?album).

Druhá časť dotazu je selekcia a tvorí sa klauzulou WHERE, v ktorej sa nachádzajú základné vzory grafu [14].

```
PREFIX music: <http://stardog.com/tutorial/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?album, ?datum, ?autor, ?price
WHERE {
    ?album rdf:type music:Album .
    ?album music:autor ?autor .
    ?album music:datum ?datum .
    ?autor rdf:type music:SoloUmelec .
    OPTIONAL{
        ?album musc:price ?price .
    }
    FILTER (?datum >= "1970-01-01"^^xsd:date)
}
```

Listing 2.7: SPARQL dotaz [15]

Na prvom riadku selekcie, je predikátom rdf:type, ktorý hovorí, že do premennej ?album hľadáme inštanciu typu music:Album. Na ďalšom riadku požadujeme naplniť premennú ?autor inštanciou, ktorá je s premennou ?album vo vzťahu music:autor. Inými slovami, požadujeme, aby sa okrem albumov vyhľadali aj ich autori. Podobným spôsobom požadujeme získanie dátumu albumov. Pokračujeme požiadavkou, aby bol autor typu music:SoloUmelec, teda sólovy umelec. Klauzula OPTIONAL požaduje získanie informácie, iba ak takáto informácia existuje a aj z prekladu vyplýva, že je nepovinná. Nie všetky výsledky teda musia danú informáciu obsahovať. V našom príklade sa

pokúsime získať ceny albumov. Na poslednom riadku, sme pridali obmedzenie kľúčovým slovom FILTER. To odfiltruje výsledky, ktorých dátum nespĺňal podmienku väčší alebo rovný "1970-01-01". V podmienke vidíme aj zápis literálu dátového typu date. Literály zapisujeme do úvodzoviek, za ktorými následujú symboly ^^ a dátový typ. Dátový typ nie je potrebné uvádzať pre literál typu string. Za tento literál je ale možné definovať jazykovú značku ("word"@en), ktorou definujeme jazyk daného literálu.

Premenná ?album sa nachádza na viacerých riadkoch, a preto môžeme tento zápis skrátiť. Názov premennej necháme len na prvom riadku a všetky ďalšie grafové vzory oddeľujeme bodkočiarkou namiesto bodky. Ďalej môžeme nahradiť zápis rdf:type kľúčovým slovom a. Skrátenú verziu vidíme nižšie.

```
?album a :Album ;
        :autor ?autor ;
        :datum ?datum .
```

Listing 2.8: Skrátenia verzia základných grafových vzorov [15]

2.7 JPA

JPA (Java Persistence API) je štandardná špecifikácia rozhrania, ktoré slúži na presun dát medzi relačnou databázou a aplikáciami písanými objektovo-orientovaným programovacím jazykom Java. Rozhranie ponúka objektovo-relačné mapovanie, ktoré umožňuje mapovať tabuľku relačnej databázy na triedu v jazyku Java. Ďalej poskytuje metódy, ktoré umožňujú vykonávať operácie ako vyhľadávanie, upravovanie a mazanie dát v databáze, ktoré môžu byť zabezpečené transakčným kontextom [16].

2.7.1 JPQL

JPA predpisuje objektový dotazovací jazyk JPQL, ktorý používa syntax podobný štandardnému dotazovaciemu jazyku SQL. Na výpise 2.9 môžeme vidieť rovnaký dotaz v jazyku SQL a jazyku JPQL. Hlavným rozdielom medzi dotazovacím a objektovo dotazovacím jazykom je to, nad čím sa tieto jazyky dotazujú. V SQL ide o tabuľku a v JPQL zasa o entitu, ktorá sa nachádza v aplikačnej doméne a je na tabuľku namapovaná [17].

```
-- SQL
SELECT DISTINCT d.id, d.name
FROM project p, employees_projects ep,
    employee e, department d
WHERE p.id = ep.project_id
    AND ep.employee_id = e.id
    AND e.department_id = d.id
    AND p.name = 'Release1'
```

```
-- JPQL
    SELECT DISTINCT e.department
    FROM Project p JOIN p.employees e
    WHERE p.name = 'Release1'
```

Listing 2.9: SQL a JPQL [17]

2.7.2 Criteria API

Súčasťou JPA špecifikácie je aj programovateľné dotazovacie rozhranie s názvom **Criteria API**. Toto rozhranie slúži ako nástroj na dynamické vytváranie dotazov prostredníctvom vytvárania objektov a volania metód. Pred vznikom objektových dotazovacích jazykov, akým je aj JPQL, bolo najpoužívanejšou metódou dotazovania v mnohých perzistentných poskytovateľoch práve prostredníctvom programovateľných rozhraní. Preto bolo spolu s príchodom špecifikácie JPA 2.0 predstavené aj Criteria API, ktoré štandardizuje funkcie dotazovacích rozhraní [2].

Pomocou Criteria API môžeme vytvárať staticky typované dotazy, aby sme mohli odhaliť chyby, už v čase kompilácie aplikácie. Práve táto typová bezpečnosť je veľkou výhodou, ktorú môžeme oceniť vo veľkých podnikových (enterprise) aplikáciách, kde štruktúra dotazu nemusí byť dopredu známa a vzniká až počas behu aplikácie [2]. Na výpise 2.10 je ukážka dotazu vytvoreného pomocou Criteria API, ktorý je ekvivalentný ku predošlému dotazu na výpise 2.9.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Department> query = cb.createQuery(Department.class);
Root<Project> p = query.from(Project.class);
Join<Project,Employee> join = p.join(Person_.employees);
ParameterExpression<String> nameParam = cb.parameter(String.class);
query.select(join.get(Person_.department))
        .distinct(true)
        .where(cb.equal(p.get(Person_.name), nameParam));
TypedQuery<Department> tq = em.createQuery(query);
List<Department> results = tq.getResultList();
```

Listing 2.10: Criteria API [2]

2.8 JOPA

JPA je štandardom vo svete relačných dát a má veľké využitie. Využívaním týchto štandardov aj vo vývoji aplikácií, ktoré pracujú so sémantickými dátami sa môže tento vývoj výrazne zefektívniť a zkvalitniť.

JOPA (Java OWL Persistence API) je knižnica zameraná na efektívny programovateľný prístup k OWL2 ontológiám a RDF grafom v programovacom jazyku Java [18]. Funkcie tejto knižnice boli inšpirované vyššie spomenutou JPA špecifikáciou. Knižnica JOPA podporuje transakčné spracovanie dotazov, kešovanie (caching) a kaskádovanie (cascading). Taktiež podporuje vykonávanie SPARQL dotazov, ktoré mapuje priamo na entity, teda ponúka objektovo-ontologické mapovanie [19].

Knižnica však zatiaľ nepodporuje programovateľné dotazovacie rozhranie podobné Criteria API. Práve túto chýbajúcu časť knižnice by mali vyplniť výsledky tejto práce.

2.9 SOQL

SOQL je analogický objektový dotazovací jazyk k JPQL (alebo HQL) určený pre knižnicu JOPA. Jeho hlavným účelom je zjednodušiť vytváranie dotazov na objektový model spravovaný knižnicou JOPA. SOQL dotazy sú prekladané do jazyka SPARQL na základe metamodelu. Cieľom jazyka SOQL je podporiť čo najviac funkcií JPQL, s prihliadnutím na špecifiká jazyka SPARQL a ontológií [20].

Ukážka dotazov v objektovom dotazovacom jazyku SOQL:

```
SELECT p FROM Person p
SELECT p FROM Person p WHERE p.username = :username
SELECT p FROM Person p WHERE p.age > :age ORDER BY p.age DESC
```

Listing 2.11: SOQL [20]

Kapitola 3

Analýza programovateľných dotazovacích rozhraní

V tejto kapitole budeme analyzovať štandardné programovateľné dotazovacie rozhranie Criteria API a jednu z jeho alternatív v podobe QueryDSL. Analyzovať budeme trochu odlišný spôsob vytvárania dotazov na jednotlivých konštruktoch s ukážkami kódu.

3.1 Criteria API

Pred tým ako sa pustíme na samotné dotazy je nutné poznamenať, že vo výpisoch kódu používame premennú em, ktorá je typu EntityManager. Táto trieda sa predovšetkým stará o životný cyklus entít. Ponúka však aj metódy, ktoré budeme pri tvorbe dotazou pomocou Criteria API potrebovať.

3.1.1 Základné stavebné prvky

Na príklade jednoduchého dotazu vo výpise 3.1, ktorým chceme získať všetky inštancie jednej triedy si ukážeme základné stavebné prvky, ktoré potrebujeme pri vytváraní dotazov a získavaní výsledkov dotazov.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> query = cb.createQuery(Person.class);

Root<Person> person = query.from(Person.class);
query.select(person);

TypedQuery<Person> typedQuery = em.createQuery(query);
List<Person> people = typedQuery.getResultList();
```

Listing 3.1: Jednoduchý Criteria API dotaz [2]

Pri vytváraní dotazov potrebujeme 3 objekty, nasledujúcich tried:

- CriteriaQuery predstavuje samotný dotaz, ktorý postupne vytvárame.
- CriteriaBuilder slúži na vytváranie inštancie dotazu ale aj mnoho ďalších objektov, pomocou ktorých dotaz tvoríme.
- Root predsavuje koreňový element dotazu. Vzniká po zavolání metódy from na objekte dotazu, ktorá špecifikuje typ dotazovanej entity.

Ak chcem vykonať dotaz a získať dotazované dáta, musíme používať objekt typu TypedQuery, ktorý získame pomocou metódy createQuery(CriteriaQuery) triedy EntityManager.

Na výpise 3.2 môžeme vidieť, že Criteria API sa istým spôsobom snaží byť programovateľným ekvivalentom ku jazyku JPQL.

```
Root<Person> person = query.from(Person.class);     //FROM Person person
query.select(person);      //SELECT person
```

Listing 3.2: Jednoduchý JPQL dotaz [17]

3.1.2 Projekcia

Na ukážke vo výpise 3.1 sme mali dotaz s projekciou celej entity. Na projekciu nám slúži metóda select(Selection<? extends T>), do ktorej môžeme vložiť koreňový element, ale aj atribút či výraz predstavujúci agregačnú funkciu. Predpis argumentu metódy však vyžaduje, aby bol rovnakého typu ako je návratová hodnota dotazu, alebo tento typ dedil. Na výpise 3.3 môžeme vidieť ukážku troch spomínaných projekcií.

```
CriteriaQuery<Person> entityQuery = criteriaBuilder.createQuery(Person.class);
Root<Person> person = entityQuery.from(Person.class);
entityQuery.select(person);

CriteriaQuery<String> stringAttributeQuery = criteriaBuilder.createQuery(String.class);
Root<Person> person = stringAttributeQuery.from(Person.class);
Path<String> attributePath = person.<String>get("name");
stringAttributeQuery.select(attributePath);

CriteriaQuery<Long> aggregMethodQuery = criteriaBuilder.createQuery(Long.class);
Root<Person> person = aggregMethodQuery.from(Person.class);
Expression<Long> aggregExpr = criteriaBuilder.count(person);
aggregMethodQuery.select(aggregExpr);
```

Listing 3.3: Projekcia v Criteria API

3.1.3 Selekcia

Selekciu dotazu vykonávame pomocou metódy where. Na vytváranie výrazov, ktoré predstavujú podmienku dotazu používame CriteriaBuilder metódy. Príklad vyhľadávania človeka podľa konkrétneho mena môžeme vidieť na výpise 3.4.

```
CriteriaQuery<Person> query = criteriaBuilder.createQuery(Person.class);
Root<Person> person = query.from(Person.class);
Path<String> attributePath = person.get("name");
Expression<Boolean> restriction = criteriaBuilder.equal(attributePath, "Milan");
query.select(person).where(restriction);
```

Listing 3.4: Jednoduchá selekcia v Criteria API [2]

Okrem výrazu equal existuje mnoho ďalších výrazov, ktoré pri selekcii dotazu môžeme využívať. V tabuľke 3.1 môžeme vidieť výrazy v jazyku JPQL a k ním odpovedajúce metódy v triede CriteriaBuilder.

JPQL výraz	CriteriaBuilder metóda
AND	and()
OR	or()
NOT	not()
=	equal()
<>	notEqual()
>	greaterThan(), gt()
>=	greaterThanOrEqualTo(), ge()
<	lessThan(), lt()
<=	lessThanOrEqualTo(), le()
BETWEEN	between()
IS NULL	isNull()
IS NOT NULL	isNotNull()
IN	in()
NOT IN	not(in())

Tabuľka 3.1: JPQL vs. CriteriaBuilder [2]

Pre vkladanie väčšieho množstva výrazov do dotazu existuje preťažená metóda where(Predicate... restrictions), ktorá prijíma variabilné množstvo argumentov typu Predicate. Medzi týmito predikátmi vznikne automaticky konjunkcia. Na vytváranie konjunkcie a disjunkcie môžeme používať aj metódy criteriaBuider.and() a criteriaBuider.or(), ktoré prijímajú práve dva booleanovské výrazy alebo variabilné množstvo predikátov. Pretože Java generika nie je kompatibilná s variabilným množstvom argumentov, CriteriaBuilder obaľuje všetky booleanovské výrazy do triedy Predicate [21]. Použitie konjunkcie a disjunkcie môžeme vidieť na výpise 3.5.

```
CriteriaQuery<Person> query = criteriaBuilder.createQuery(Person.class);
Root<Person> person = query.from(Person.class);

Predicate name = criteriaBuilder.equal(person.get("name"),"Milan");
Predicate anotherName = criteriaBuilder.equal(person.get("name"),"Peter");
Predicate age = criteriaBuilder.greaterThan(person.get("age"), 22);

query.where(age, criteriaBuilder.or(name,anotherName));
// WHERE person.age > 22 AND (person.name = "Milan" OR person.name = "Peter")
```

Listing 3.5: Konjunkcia a disjunkcia [2]

Ak chceme výrazy negovať, môžeme využiť metódu triedy CriteriaBuilder alebo predikát negovať pomocou jeho vlastnej metódy. Obe varianty môžeme vidieť na výpise nižšie.

```
Predicate predicate = criteriaBuilder.equal(person.get("name"),"Milan");
Predicate negatedByFactory = criteriaBuilder.not(predicate);

Predicate negatedByYourself = predicate.not();
```

Listing 3.6: Negácia výrazov v Criteria API

3.1.4 Dynamická selekcia

Niekedy je potrebné vytvárať podmienky dotazu dynamicky. Jednou variantou je prázdna konjunkcia alebo disjunkcia v premennej. Prázdna konjunkcia sa sama o sebe vyhodnocuje na true a prázdna disjunkcia sa vyhodnocuje na false. My však tento predikát obalíme do nového predikátu, ktorý spojí pôvodný prázdny predikát s novým predikátom a prepíšeme premennú. Táto postupne obaľovaná premenná bude nakoniec finálnym predikátom, ktorý naviažeme na dotaz. Na výpise 3.7 vidíme ukážku.

```
public List<Person> findByNameAndAge(String nameVar, Integer ageVar){
    ...
    Predicate restrictions = criteriaBuilder.conjunction();
    if(nameVar != null) {
        Predicate nameRestriction = criteriaBuilder.equals(person.get("name"), nameVar);
        restrictions = criteriaBuilder.and(restrictions, nameRestriction);
    }
    if(ageVar != null){
        Predicate ageRestriction = criteriaBuilder.equals(person.get("age"), ageVar);
        restrictions = criteriaBuilder.and(restrictions, ageRestriction);
    }
    query.where(restrictions);
    ...
```

Listing 3.7: Dynamické podmienky [2]

Ďalšou variantou je vkladanie predikátov do zoznamu, ktorý sa vloží do dotazu. Príklad vidíme na výpise 3.8. Zoznam je však potrebné pred vložením do dotazu upraviť na pole.

```
public List<Person> findByNameAndAge(String nameVar, Integer ageVar){
    ...
    List<Predicate> restrictions = new ArrayList<Predicate>();
    if(nameVar != null) {
        restrictions.add(criteriaBuilder.equals(person.get("name"), nameVar));
    }
    if(ageVar != null){
        restrictions.add(criteriaBuilder.equals(person.get("age"), ageVar));
    }
    query.select(person).where(restrictions.toArray(new Predicate[0]));
    ...
```

Listing 3.8: Dynamické podmienky pomocou zoznamu [2]

3.1.5 Spájanie

V predošlej kapitole sme si už ukazovali dotaz so spájaním. Pripomenieme si ho na výpise 3.9 so zvýraznením dôležitých častí. Chceme získať oddelenia všetkých pracovníkov, ktorí pracovali na konkrétnom projekte. Spájanie robíme pomocou metódy join, na koreňovom elemente dotazu. Metóda nám vráti objekt typu Join.

```
CriteriaQuery<Department> query = criteriaBuilder.createQuery(Department.class);
Root<Project> project = query.from(Project.class);
Join<Project,Employee> joinEmployees = project.join("employee");
query.select(joinEmployees.get("department")).distinct(true)
        .where(criteriaBuilder.equal(project.get("name"),"Release1"));
```

Listing 3.9: Spájanie v Criteria API [2]

Vďaka tomu, že Join implementuje rovnaké rozhrania ako koreňový element Root, môžeme pristupovať ku atribútom spojenej entity alebo pokračovať v spájaní.

Prednastavený typ spájania je inner. V prípade potreby môžeme do metódy join poslať druhý argument, ktorý predstavuje požadavný typ spájania. Okrem JoinType.INNER môžeme používať JoinType.LEFT a JoinType.RIGHT.

3.1.6 Poddotazy

Pozrime sa na ešte komplikovanejší dotaz. V predošlom dotaze sme hľadali oddelenia. Teraz chceme vyhľadať všetkých zamestnancov oddelení, na ktorých sa realizoval projekt s konkrétnym názvom. Zápis takéhoto dotazu v jazyku JPQL môžeme vidieť na výpise 3.10.

```
SELECT employee
FROM Employee employee
WHERE employee.department IN
    (SELECT DISTINCT d
    FROM Department d JOIN d.employees de JOIN de.project p
    WHERE p.name = "Release1")
```

Listing 3.10: Poddotaz v JPQL [2]

Na vytváranie poddotazov slúži metóda subquery, ktorú nám ponúka samotný dotaz. Metóda prijíma návratový typ poddotazu a vracia nám poddotaz (objekt typu Subquery), s ktorým môžeme pracovať rovnako ako s obyčajným dotazom. Na výpise 3.11 môžeme vidieť použitie poddotazu, ale aj použitie výrazu IN.

```
CriteriaQuery<Employee> query = criteriaBuilder.createQuery(Employee.class);
Root<Employee> employee = query.from(Employee.class);

Subquery<Department> subQuery = query.subquery(Department.class);
Root<Department> department = subQuery.from(Department.class);
Join<Employee,Project> joinProjects = deparment.join("employees").join("projects");
subQuery.select(department).distinct(true)
    .where(criteriaBuilder.equal(joinProjects.get("name"), "Release1"));

query.select(employee)
    .where(criteriaBuilder.in(employee.get("department")).value(subQuery));
```

Listing 3.11: Poddotaz v Criteria API [2]

Do hlavného dotazu teda vložíme podmienku, že oddelenie zamestnanca musí byť zo zoznamu oddelení. Zoznam oddelení potom vytvoríme pomocou metóda criteriaBuider.in(), ktorá vracia objekt typu In, na ktorom pomocou volania metódy value vkladáme hodnoty zoznamu. V našom príklade bude hodnotou výsledok poddotazu, ktorý spojí oddelenia so zamestnancami a ich projektami. Potom vyberie tie s konkrétnym názvom projektu.

3.1.7 Typová bezpečnosť

V niektorých doterajších ukážkach sme pristupovali k atribútom entít prostredníctvom názvu, ktorý sme vkladali ako string. Na výpise 3.12 vidíme v prvej časti takéto použitie a nedokážeme overiť, že typ atribútu odpovedá typu dotazu. Táto chyba sa prejavuje, až keď začneme pristupovaní ku výsledkom dotazu, teda dotaz samotný sa vyhodnotí bez chyby.

```
CriteriaQuery<String> query = criteriaBuilder.createQuery(String.class);
Root<Person> person = entityQuery.from(Person.class);
query.select(person.get("name")); //unsafe

CriteriaQuery<String> query = criteriaBuilder.createQuery(String.class);
Root<Person> person = entityQuery.from(Person.class);
query.select(person.<String>get("name")); //safer
```

Listing 3.12: Typ atribútu

V druhej časti výpisu sme to upravili a vyžadujeme, aby mal atribút požadovaný typ. Takýmto riešením sa ale znižuje čitateľnosť dotazu. Príjemnejším, elegantnejším a bezpečnejším spôsobom je používanie metamodelu.

Metamodel

Metamodel popisuje typ perzistentnej jednotky, stav a vzťah medzi entitami. Pomocou neho dokážeme pristupovať ku názvom a typom atribútov a jeho používanie nám vyrieši problémy, ktoré boli popísané vyššie.

Používanie samotného Metamodel API však dotazy značne zneprehľadnilo, a preto spolu s príchodom špecifikácie JPA 2.0 bol predstavený kanonický metamodel. Ten zjednodušil a sprehľadil používanie metamodelu. Trieda kanonického metamodelu obsahuje statické atribúty, ktoré odpovedajú atribútom perzistentnej triedy. Tieto triedy môžu byť generované pomocou rôznych nástrojov alebo manuálne vytvorené. Na výpise 3.13 vidíme manuálne vytvorený metamodel.

```
@StaticMetamodel(Person.class)
public class Person_ {
    public static volatile SingularAttribute<Employee, Integer> id;
    public static volatile SingularAttribute<Employee, String> name;
    public static volatile SingularAttribute<Employee, Integer> age;
}
```

Listing 3.13: Metamodel [2]

Upravíme predošlý dotaz za pomoci metamodelu. Triedy metamodelu majú rovnaký názov ako perzistentná trieda rozšírený o symbol _. Ku atribútom pristupuje cez bodkovú notáciu priamo cez názov triedy metamodelu. Výsledný upravnený dotaz vidíme na výpise nižšie.

```
CriteriaQuery<String> query = criteriaBuilder.createQuery(String.class);
Root<Person> person = query.from(Person.class);
query.select(person.get(Person_.name));
```

Listing 3.14: Použitie kanonického metamodelu

3.1.8 Parametre

Ak v podmienkach dotazov pracujeme s premennými hodnotami, je potrebné do dotazu vkladať parametre namiesto obyčajných premenných, aby sme zabránili SQL útokom.

Parametre v Criteria API sú staticky typované a môžu byť použité na viacerých miestach v dotaze. Vytvárame ich pomocou metódy criteriaBuilder.parameter(), ktorá ako argument prijíma dátový typ parametru, prípadne aj názov parametru. Pred vykonaním dotazu je potom potrebné parametre naplniť [2]. Ukážku použitia parametrov vidíme na výpise 3.15.

```
CriteriaQuery<Person> query = criteriaBuilder.createQuery(Person.class);
Root<Person> person = query.from(Person.class);

ParameterExpression<String> namePar = criteriaBuilder.parameter(String.class);
ParameterExpression<Integer> agePar = criteriaBuilder.parameter(Integer.class, "agePar");
Predicate name = criteriaBuilder.equal(person.get(Person_.name), namePar);
Predicate age = criteriaBuilder.greaterThan(person.get(Person._age), agePar);
query.select(person).where(name,age);

TypedQuery<Person> typedQuery = em.createQuery(query);
typedQuery.setParameter(namePar, "Milan");
typedQuery.setParameter("agePar", 22);
```

Listing 3.15: Parametre v Criteria API [2]

3.2 QueryDSL

QueryDSL je framework, umožňujúci vytvárať staticky typované dotazy pomocou plynulého rozhrania. Zrodil sa kvôli potrebe udržovať HQL dotazy typovo bezpečné. HQL používaný prostredníctvom perzistentého poskytovateľa Hibernate bol teda prvým cieľom knižnice QueryDSL. V súčasnosti už je QueryDSL kompatibilné aj s JPA, JDO, JDBC a MongoDB [24].

Na výpisoch kódu budeme používať QueryDSL integrované pomocou JPA. Preto sa môže v kóde opäť vyskytnúť EntityManager ako premenná em.

3.2.1 Dotazové typy

QueryDSL využíva svoj vlastný metamodelový mechanizmus, v ktorom triedy odpovedajúce perzistentným triedam nazýva **dotazové typy** [24]. Dotazové typy sú generované a ich názov pozostáva z názvu perzistentnej triedy a prefixu "Q". Ku inštancii týchto tried môžeme pristupovať cez staticky atribút triedy (QPerson p = QPerson.person) alebo ju inicializujeme sami (QPerson p = new QPerson("p")).

Dotazový typ ponúka priamy prístup ku atribútom entity. Je akousi kombináciou koreňového elementu a metamodelu v Criteria API. Vďaka tomu nie je možné vytvoriť menej typovo bezpečné dotazy, o ktorých sme hovorili v podsekcii 3.1.7 o typovej bezpečnosti Criteria API.

3.2.2 Základné stavebné prvky

Na samotný dotaz slúži trieda JPAQuery. Dotaz môžeme vytvoriť inicializáciou (new JPAQuery(entityManager)), ale preferovanou variantov je vytváranie dotazu pomocou JPAQueryFactory, ktorú uvidíme v ďalších sekciách.

Pri vytváraní dotazu potom pracujeme hlavne s dotazovým typom, ktorý umožňuje pristupovať ku atribútom a vytvárať podmienky dotazu.

3.2.3 Projekcia

Keďže QueryDSL nepoužíva koreňový element tak je možné vytvorenie celého dotazu na jednom riadku. Na výpise 3.16 môžeme vidieť, že špecifikovanie entity, z ktorej dotazujeme a následná projekcia celej entity je vďaka JPAQueryFactory možné spojiť v jednej metóde. Ďalej môžeme vidieť, že projekcia atribútu či agregačnej funkcie pomocou dotazového typu je veľmi intuitívna.

```
QPerson person = QPerson.person;
JPAQueryFactory queryFactory = new JPAQueryFactory(em);

JPAQuery<Person> entityQuery = queryFactory.selectFrom(person);
JPAQuery<String> stringAttributeQuery = queryFactory.select(person.name).from(person);
JPAQuery<Long> aggregationMethodQuery = queryFactory.select(person.count()).from(person);
```

Listing 3.16: Projekcia v QueryDSL [24]

3.2.4 Selekcia

Selekcia sa vykonáva podobne ako v Criteria API pomocou metódy where, ktorá berie ako argumenty variabilné množstvo predikátov, medzi ktorými vytvára konjunkciu. Hlavným rozdielom je spôsob akým sa vytvárajú výrazy, ktoré predstavujú podmienky selekcie. Atribúty dotazového typu majú verejné metódy, pomocou ktorých vznikajú podmienky. Podmienky sú typu BooleanExpression, ktorý následne ponúka metódy and a or na spájanie výrazov. Ukážku vidíme na výpise nižšie.

```
JPAQueryFactory queryFactory = new JPAQueryFactory(em);
QPerson person = QPerson.person;

queryFactory.selectFrom(person)
            .where(person.age.gt(22), person.name.eq("Milan").or(person.name.eq("Peter")));
```

Listing 3.17: Selekcia v QueryDSL [24]

V tabuľke 3.2 máme JPQL výrazy a metódy, ktoré im odpovedajú. Sú rozdelené podľa tried, ktoré ich ponúkajú. Povieme si teda niečo o týchto triedach. Každý atribút dotazového typu nepriamo dedí triedu SimpleExpression. Napríklad atribút Qperson.person.age je typu NumberPath, ten dedí NumberExpression, a cez niekoľko ďalších predkov aj SimpleExpression. V tabuľke máme ešte triedu ComparableExpression, ktorá je nepriamym predkom triedy StringPath, ktorá je typom atribútu Qperson.person.name. Výsledkom všetkých metód sú booleanovské výrazy, preto máme v tabuľke aj triedu BooleanExpression. Metódy v tabuľke napísané kurzívou sú zdedené metódy.

JPQL výraz	**Boolean Expression**	**Simple Expression**	**NumberExpression ComparableExpression**
AND	and()		
OR	or()		
NOT	not()		
=	*eq()*	eq()	*eq()*
<>	*ne()*	ne()	*ne()*
>	*gt()*		gt()
>=	*goe()*		goe()
<	*lt()*		lt()
<=	*loe()*		loe()
BETWEEN	*between()*		between()
IS NULL	*isNull()*	isNull()	*isNull()*
IS NOT NULL	*isNotNull()*	isNotNull()	*isNotNull()*
IN	*in()*	in()	*in()*
NOT IN	*notIn()*	notIn()	*notIn()*

Tabuľka 3.2: JPQL vs. QueryDSL

3.2.5 Dynamická selekcia

Na dynamické pridávanie podmienok dotazu existuje trieda BooleanBuilder, ktorá ponúka metódy na vytváranie konjunkcie, disjunkcie alebo ich negácií. Ukážku vidíme na výpise 3.18.

```
public List<Person> findByNameAndAge(String nameVar, Integer ageVar){
    ...
    BooleanBuilder restrictions = new BooleanBuilder();
    if(nameVar != null) restrictions.and(person.name.eq(nameVar));
    if(ageVar != null) restrictions.and(person.age.eq(ageVar));
    query.where(restrictions);
    ...
```

Listing 3.18: Dynamická selekcia v QueryDSL [24]

Ak chceme výrazy negovať, môžeme využiť metódu triedy BooleanExpression. Pri dynamickom pridávaní podmienok môžeme využiť negovacie metódy, ktoré poskytuje BooleanBuilder. Obe varianty môžeme vidieť na výpise nižšie.

```
BooleanExpression booleanExpression = person.name.eq("Milan");
BooleanExpression negatedExpression = booleanExpression.not();

BooleanBuilder restrictions = new BooleanBuilder();
restrictions.andNot(booleanExpression);
```

Listing 3.19: Negácia výrazov v QueryDSL [24]

3.2.6 Spájanie

Na ukážku spájania použijeme rovnaký dotaz ako vo výpise 3.9 v sekcii o Criteria API. Jeho podobu v QueryDSL môžeme vidieť na výpise 3.20.

```
QEmployee employee = QEmployee.employee;
QProject project = QProject.project;
queryFactory.select(employee.department).distinct().from(project)
            .join(project.employees, employee)   //innerJoin(project.employees, employee)
            .where(project.name.eq("Release1"));
```

Listing 3.20: Spájanie v QueryDSL [24]

Vidíme, že spájanie vykonávame priamo na samotnom dotaze. Na spájanie typu inner môžeme využiť metódu join() alebo innerJoin(). Na ďalšie typy spájanie existujú metódy s odpovedajúcim názvom typu, teda leftJoin() a rightJoin(). Metódy prijímajú ako prvý argument zdroj spájania a ako druhý argument cieľ spájania. Oproti Criteria API tu nie je potrebné toto spájanie ukladať do premennej.

3.2.7 Poddotaz

Na vytváranie poddotazov slúžia statické metódy triedy JPAExpressions. Poddotaz môžeme vytvárať priamo na mieste, kde ho vkladáme, ako to vidíme v prvej časti výpisu 3.21.Celý dotaz tak môže vyzerať plynulejší ak poddotaz nie je príliš dlhý. Pri ukladaní poddotazu do premennej používame typ JPQLQuery. Obe varianty môžeme vidieť na výpise nižšie.

```
QEmployee employee = QEmployee.employee;
QDepartment department = QDepartment.department;
QProject project = QProject.project;
queryFactory.selectFrom(employee).distinct()
            .where(employee.department.name.in(
                  JPAExpressions.select(department.name).from(department)
                        .join(department.employees, employee)
                        .join(employee.projects, project)
                        .where(project.name.eq("Release1"))
            ));

JPQLQuery<String> subQuery = JPAExpressions.select(department.name).from(department)
                                                    .join(department.employees, employee)
                                                    ...;
queryFactory.selectFrom(employee).distinct()
            .where(employee.department.name.in(subQuery));
```

Listing 3.21: Poddotaz v QueryDSL [24]

3.2.8 Parametre

Parametre vytvárame inicializáciou triedy Param. Keďže QueryDSL nepoužíva TypedQuery, hodnota parametrov sa nastavuje priamo na dotaze pomocou metódy set. Na výpise vidíme, že parametre môžeme pomenovať. Naplnenie parametra podľa mena je ale dosť neintuitívne. Je potrebné inicializovať dva krát parameter s rovnakým názvom. Preto je vhodnejšie naplňovať parameter pomocou referencie na objekt parametra.

```
QPerson person = QPerson.person;

Param<String> namePar = new Param<>(String.class);
Param<Integer> agePar = new Param<>(Integer.class, "agePar");
BooleanExpression name = person.name.eq(namePar);
BooleanExpression age = person.age.gt(agePar);
queryFactory.selectFrom(person).where(name, age)
    .set(namePar, "Milan")
    .set(new Param<>(Integer.class, "agePar"), 22);
```

Listing 3.22: Parametre v QueryDSL [24]

3.3 Porovnanie

Na vytváranie kompexných dotazov pomocou rozhrania Criteria API je potreba využívať triedu CriteriaBuilder, čím sa znižuje čitateľnosť dotazov. Niektoré konštrukty nie sú veľmi intuitívne a je potrebné si na nich zvyknúť. Náročnosť používania ale klesá veľmi rýchlo s pribúdajúcimi skúsenosťami. Keďže je Criteria API štandardizované rozhranie, existuje veľké množstvo kvalitných návodov na jeho používanie. Vďaka tomu je oveľa jednoduchšie vytvárať naozaj typovo bezpečné dotazy.

QueryDSL naopak ponúka intuitívnejší a prehľadnejší spôsob vytvárania komplexných dotazov. Dotazové typy, ktoré QueryDSL používa, kombinujú schopnosti JPA metamodelu so schopnosťami triedy CriteriaBuilder, čím sa dotazy stávajú plynulejšími. Neexistuje však veľa kvalitných návodov a pri nesprávnom používaní môžeme veľmi rýchlo prísť o niektoré prvky typovej kontroly.

3.4 Špecifické konštrukty pre sémentické dáta

Väčšina konštruktov, ktoré sme si v tejto kapitole ukázali môžeme preniesť a používať v programovateľnom dotazovacom rozhraní pre sémantické dáta. Niektoré konštrukty však nedávajú význam pre sémantické dáta. Je teda potrebné pridať vkladanie jazykovej značky pre string literály a parametre. Naopak implementovať nebudeme konštrukty, ktoré pracujú s null hodnotami, ako napríklad isNull predikát či null literál.

Najvýraznejším rozdielom je spájanie. Spájanie typu inner v jazyku SQL má rovnakú vyjadrovacú schopnosť, ako niektoré základné grafové vzory v jazyku SPARQL. Spájanie typu left outer v SQL sa môže objaviť a odpovedať základnému grafovému vzoru v klauzule OPTIONAL jazyka SPARQL [22]. V programovateľnom dotazovacom rozhraní pre sémantické dáta teda nemá význam explicitne používať metódy join, leftJoin, rightJoin a rozhranie by malo byť rozšírené o metódu optional.

Kapitola 4
Návrh

Táto kapitola popisuje ako bude vyzerať programovateľné dotazovacie rozhranie, ktoré bude súčasťou knižnice JOPA. V rámci tejto, ale aj ostatných kapitol budeme navrhované a následne implementované rozhranie označovať ako **Semantic Criteria API**.

4.1 Preklad dotazu

Ako prvé je potrebné vybrať jazyk, do ktorého budeme dotazy prekladať. Na výber máme nativný dotazovací jazyk SPARQL a objektový dotazovací jazyk SOQL.

Dotazy vytvárané pomocou Semantic Criteria API budú prekladanú do jazyka SOQL. Schopnosti jazyka sú v súčasnosti obmedzené, ale v jeho vývoji sa plánuje pokračovať. Táto voľba nám zjednoduší konečný preklad dotazu. Vďaka tomu budeme schopní naimplementovať väčšie množstvo dotazovacích konštruktov. Zároveň bude v rámci knižnice iba jedno miesto, teda preklad jazyka SOQL, na ktorom sa generuje SPARQL dotaz.

4.2 Spôsob dotazovania

Po analýze programovacích dotazovacích rozhraní smerovali prvé návrhy k tomu, aby dotazy v Semantic Criteria API pripomínali o trochu viac QueryDSL, než Criteria API. Prvotným návrhom bolo pozmeniť vytváranie koreňového elementu, aby nebolo nutné prerušiť dotaz a jeho vytváranie bolo plynulé. Vytvorili sme teda určitého zástupcu koreňového elementu a pracovne ho nazvali QueryModel. Dotaz už nevytvárame pomocou metódy createQuery, ktorá zároveň určovala návratový typ dotazu. Návratový typ teda určí metóda select, keďže návratový typ dotazu odpovedá typu projekcie.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
QueryModel<Person> person = cb.getQueryModel(Person.class);
CriteriaQuery<Person> simple = cb.selectFrom(person);
CriteriaQuery<Person> anotherSimple = cb.from(person).select(person);
CriteriaQuery<String> complex = cb.from(person)   //return CriteriaQuery<Person>
                        .select(person.get(Person_.name)); // return new CriteriaQuery<String>
```

Listing 4.1: Prvotný návrh

Metóda from musí taktiež vrátiť typovaný dotaz. Preto je potrebné v metóde select kontrolovať zhodnosť typov a v prípade nezhody je treba vytvoriť nový dotaz so správnym typom, do ktorého nakopírujeme pôvodný dotaz.

Nižšie vo výpise vidíme, že rozdiel medzi špecifikáciou a upravenou verziou nie je veľký.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
//Criteria API
CriteriaQuery<Person> query = cb.createQuery(Person.class);
Root<Person> person = query.from(Person.class);
query.select(person.get(Person_.name));
//Semantic Criteria API
QueryModel<Person> person = cb.getQueryModel(Person.class);
CriteriaQuery<String> query = cb.from(person).select(person.get(Person_.name));
```

Listing 4.2: Porovnanie s Criteria API

Výraznou zmenou by bolo vytváranie podmienok dotazu prostredníctvom triedy QueryModel. Jednou z možností je implementovať metódy na tvorbu podmienok z triedy CriteriaBuilder v triede QueryModel. Na výpise nižšie vidíme, ako by taký dotaz výzeral aj s ekvivalentami v QueryDSL a Criteria API.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
JPAQueryFactory qf = new JPAQueryFactory(em);
//Semantic Criteria API
QueryModel<Person> person = cb.getQueryModel(Person.class);
cb.selectFrom(person).where(person.attrEqual(Person_.name, "Milan"));
//QueryDSL
QPerson person = QPerson.person;
qf.selectFrom(person).where(person.name.eq("Milan"));
//Criteria API
CriteriaQuery<Person> query = cb.createQuery(Person.class);
Root<Person> person = query.from(Person.class);
query.select(person).where(cb.equal(person.get(Person_.name)), "Milan");
```

Listing 4.3: Vytváranie podmienok pomocou QueryModel

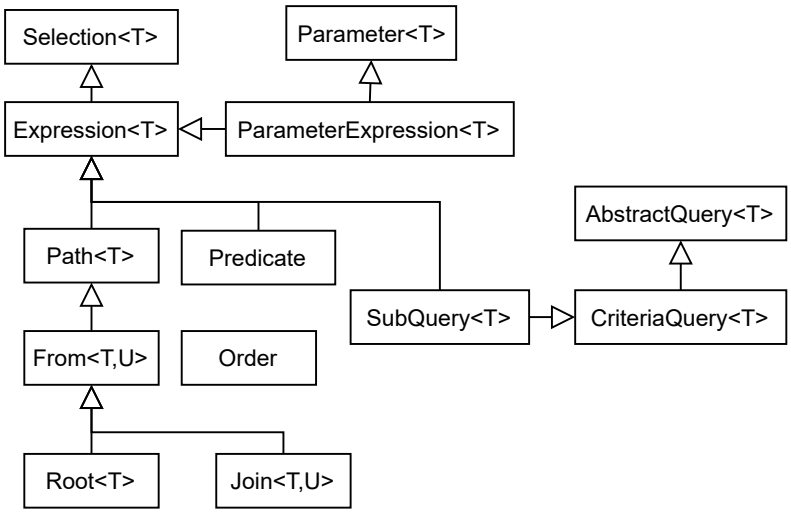
Vidíme, že tento spôsob sa líši aj od Criteria API špecifikácie ale aj od spôsobu v knižnici QueryDSL. Takto implementované rozhranie by teda potrebovalo podrobný návod. Okrem toho práve odlišnosť od zaužívaných spôsobov môže odradiť programátorov od používania nového rozhrania.

Druhou z možností je volanie metód na atribútoch entity, ako sme to videli v QueryDSL. Tento spôsob je ale veľmi ťažko realizovateľný prostredníctvom jednej triedy. Bolo by potrebné generovať triedy podobné dotazovým typom, ktoré sme videli v QueryDSL. Mohli by sme nad existujúcim metamodel urobiť nadstavbu alebo vytvoriť úplne nový metamodel s požadovanou funkcionalitou. Časová náročnosť práce by ale mohla výrazne narásť. Pôvodný cieľ práce, ktorý počíta aj s funkčnou implementáciou rozhrania, by sa mohol stať nesplniteľným. Preto sme pôvodné návrhy prehodnotili.

Semantic Criteria API teda bude inšpirované štandardnou JPA špecifikáciou, tak ako aj celá knižnica JOPA.

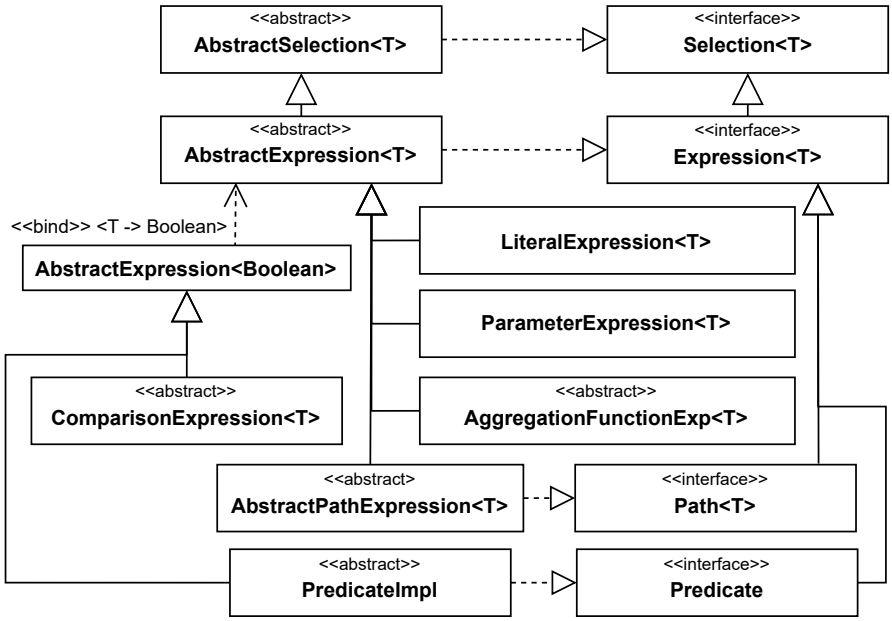
4.3 Objektový návrh

Na obrázku 4.1 vidíme diagram tried, ktorý obsahuje štruktúru dotazu podľa JPA špecifikácie. Diagram je mierne zjednodušený a zachytáva tie najdôležitejšie triedy. Táto štruktúra je verejná a užívateľ pracuje s týmito triedami.



Obrázok 4.1: Štruktúra dotazu [2]

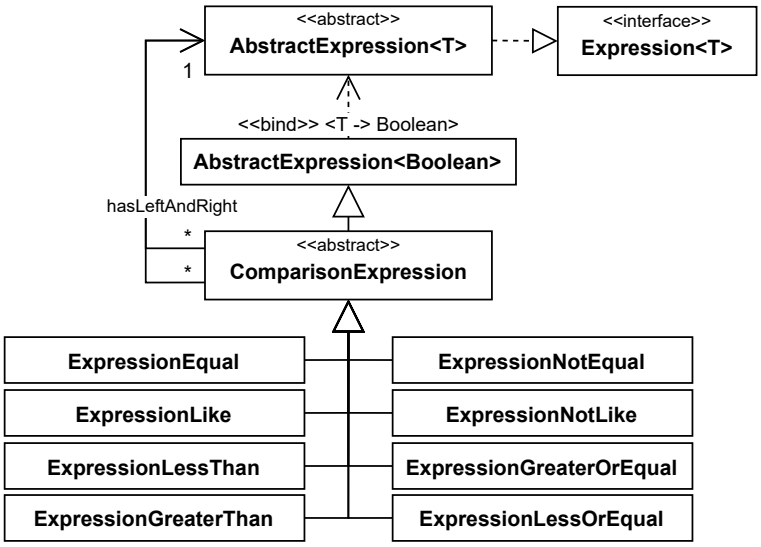
Aby sme však mohli vygenerovať správny dotaz, interná štruktúra bude komplikovanejšie ako verejná. Na obrázku 4.2 vidíme hlavnú časť diagramu.



Obrázok 4.2: UML diagram tried

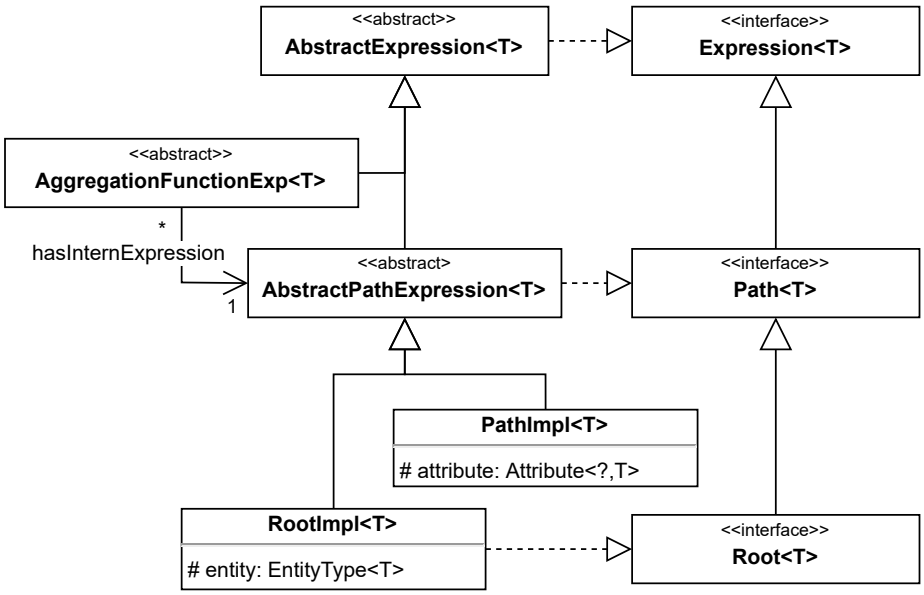
Všetky výrazy budú mať jedného spoločného abstraktného predka AbstractExpression. Niektoré priamo a niektoré nepriamo cez iného predka.

Prvým zložitejším druhom z výrazov, ktoré budeme používať sú porovnávacie výrazy. Ich diagram vidíme nižšie na obrázku 4.3. Tie budú mať spoločného abstraktného predka ComparisonExpression, ktorý bude pozostávať z ľavého a pravého výrazu, ktoré medzi sebou potrebujeme porovnávať. Konkrétne triedy potom budú predstavovať konkrétne porovnávacie výrazy, teda napríklad ExpressionLessOrEqual bude predstavovať výraz <=.



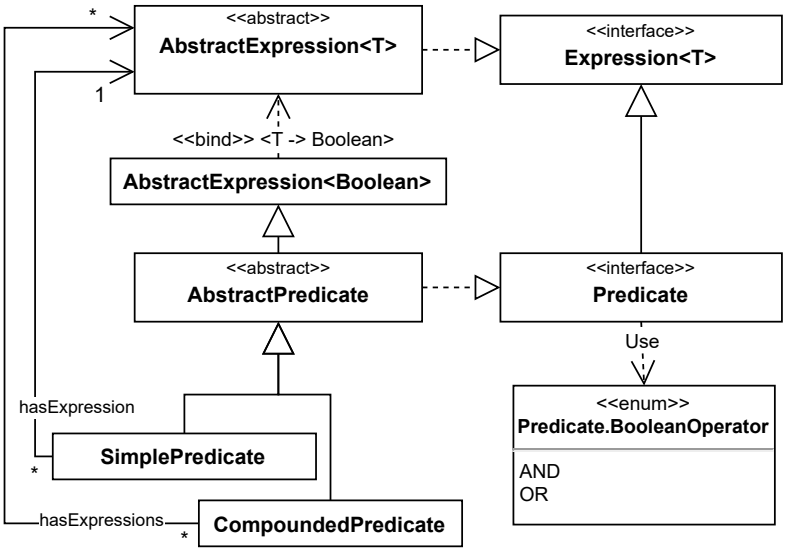
Obrázok 4.3: Porovnávacie výrazy

Ďalšou skupinou výrazov sú agregačné funkcie. Budú fungovať na podobnom princípe. Abstraktný predok AggregationFunctionExp ale bude obsahovať iba jeden vnútorný výraz. Ten bude typu AbstractPathExpression a bude predstavovať cestu k atribútu alebo k entite. Konkrétni potomkovia budú predstavovať konkrétne agregačné funkcie, napríklad CountExpression bude predstavovať výraz COUNT().



Obrázok 4.4: Výrazy cesty

Poslednou komplikovanejšou skupinou výrazov sú predikáty. Booleanovský výraz obaľujeme do jednoduchého predikátu. Na konjunkciu a disjunkciu budeme využívať triedu CompoundedPredicate, ktorá obsahuje zoznam booleanovských výrazov spolu s operátorom AND alebo OR.



Obrázok 4.5: Predikáty

Kapitola 5
Implementácia

V tejto kapitole sa pozrieme na zopár implementačných detailov a problémov, ktoré bolo nutné vyriešil počas implementácie rozhrania.

5.1 Generovanie SOQL dotazu

Z dotazu vytvoreného pomocou Semantic Criteria API potrebujeme vygenerovať správny SOQL dotaz. Používame návrhový vzor "chain of responsibility"[23] a nechávame zodpovednosť za generovanie dotazu na samotné objekty výrazov. Triedy jednotlivých výrazov prepisujú abstraktnú metódu setExpressionToQuery, ktorá prijíma objekt typu StringBuilder. Tento objekt predstavuje SOQL dotaz, do ktorého výrazy vkladajú svoju string podobu.

Pri generovaní aktívne využívame dedičnosť. Na výpise 5.1 vidíme, že predok predpisuje metódu na vpísanie výrazu do dotazu a využíva abstraktnú metódu, ktoré implementujú konkrétne výrazy. Tento spôsob využívajú porovnávacie výrazy a agregačné funkcie.

```
abstract public class AbstractComparisonExpression extends AbstractExpression<Boolean> {
    protected AbstractExpression<?> right;
    protected AbstractExpression<?> left;
    ...
    @Override
    public void setExpressionToQuery(StringBuilder query, CriteriaParameterFiller parameterFiller) {
        this.left.setExpressionToQuery(query, parameterFiller);
        query.append(this.getComparisonOperator());
        this.right.setExpressionToQuery(query, parameterFiller);
    }

    abstract protected String getComparisonOperator();
}

public class ExpressionLessThanImpl extends AbstractComparisonExpression {
    ...
    @Override
    protected String getComparisonOperator() {
        return this.isNegated() ? " >= " : " < ";
    }
}
```

Listing 5.1: Dedičnosť pri generovaní porovnávacích výrazov

Vo výrazoch equal a notEqual sme ale museli dočasne prepísať aj metódu setExpressionToQuery, pretože jazyk SOQL zatiaľ podporuje negáciu len pomocou kľúčové slova NOT a nie pomocou operátora <> alebo !=. Takáto negácia je síce funkčná ale pri komplexnom dotaze môže byť výsledný SOQL neprehľadnejší. To môže zbytočne skomplikovať kontrolu správnosti dotazu, ak programátor kontroluje SOQL namiesto SPARQL dotazu. Na výpise vidíme dočasné riešenie aj s poznámkou na budúcu opravu.

```
public class ExpressionEqualImpl extends AbstractComparisonExpression {
    ...
     //TODO – remove override method when SOQL supports equal negation as != or <>
    @Override
    public void setExpressionToQuery(StringBuilder query, CriteriaParameterFiller parameterFiller) {
        if (this.negated) query.append("NOT ");
        super.setExpressionToQuery(query, parameterFiller);
    }

    @Override
    protected String getComparisonOperator() {
        //TODO – change when SOQL supports equal negation as != or <>
        //return this.isNegated() ? " != " : " = ";
        return " = ";
    }
```

Listing 5.2: Dočasná negácia prispôsobená jazyku SOQL

5.2 Parametre

Na správne používanie parametrov musíme do SOQL dotazu vložiť názov parametru. Generovanie názvu parametra je potrebné robiť mimo triedu parametra, aby bol názov unikátny. Vytvorili sme triedu CriteriaParameterFiller, ktorá okrem iného zabezpečuje aj generovanie názvu parametra.

Pri nastavovaní hodnôt parametrov sme narazili na problém. Pri preklade SOQL dotazu sa vytvárajú objekty typu QueryParameter, ktoré predstavujú parametre. Pri vkladaní hodnôt sa teda porovnávali objekty QueryParameter s objektami ParameterExpression a dochádzalo ku chybe. Bolo teda potrebné prepísať metódy equals a hashCode oboch tried, aby mohli byť objekty správne porovnané a bolo možné parametre naplniť.

5.3 Literály

Pri používaní literálov sme sa rozhodli, že budú do SOQL dotazu vkladané podobne ako parametre a vytváranie dotazu bude o niečo bezpečnejšie. Na vkladanie literálov prostredníctvom parametrov využívame triedu CriteriaParameterFiller. Tá si okrem spomínaného generovania názvu parametru ukladá literály. Výrazy si tento objekt preposielajú v metóde setExpressionToQuery. Volaním metódy registerParameter sa literál uloží do hash mapy. Po vytvorení TypedQuery z CriteriaQuery sa preiteruje hash mapa a nastavia sa hodnoty parametrov.

5.4 Falošný booleanovský výraz

Criteria API umožňuje vytvárať menej typovo bezpečné dotazy, o ktorých sme si hovorili v analýze v podsekcií 3.1.7 Typová bezpečnosť. Špecifikácia predpisuje metóde where prijímať okrem predikátov aj práve jeden Expression<Boolean>. Na výpise 5.3 vidíme, že je do metódy where možné poslat výraz, ktorý v skutočnosti nie je booleanovský výraz.

```
Expression<Boolean> fakeBooleanExpression = root.get("username");
query.select(root).where(fakeBooleanExpression);
```

Listing 5.3: Falošný booleanovský výraz

Problém sme vyriešili metódou v triede CriteriaBuilder, ktorá skontroluje či prichádzajúci Expression<Boolean> nie je inštanciou triedy AbstractPathExpression. Takú inštanciu obalí do porovnávacieho výrazu = true a následne do jednoduchého predikátu.

5.5 Integrácia

Integrácia rozhrania do knižnice JOPA nebola zložitá. Do verejného rozhrania EntityManager a následne aj do implementujúcej triedy sme pridali metódu getCriteriaBuilder() a metódu createQuery(CriteriaQuery<T>).

Prvá zo spomínaných metód vráti užívateľový referenciu na CriteriaBuilder z aktuálneho perzistentného kontextu. Bolo teda ešte potrebné inicializovať tento objekt v konštruktore triedy UnitOfWork, ktorá reprezentuje perzistentný kontext.

Úlohou metódy createQuery je prerobenie CriteriaQuery<T> na TypedQuery<T>. Proces začína inicializovaním triedy CriteriaParameterFiller, o ktorej sme hovorili v predošlých sekciách. Referencia objektu sa posiela do volania metódy translateQuery, ktorá dotaz preloží na SOQL dotaz. Na následný preklad SOQL dotazu využívame schopnosť knižnice prekladať jazyk SOQL. Z perzistentného kontextu získame referenciu na SparqQueryFactory, ktorá zo SOQL dotazu vytvorý TypedQuery. Následne ešte voláme metódu, ktorá do typoveného dotazu vloží hodnoty literálov.

Integráciu je možné vidieť na adrese: github.com/kbss-cvut/jopa/pull/91 Na nej sa nachádza **pull request**, ktorým žiadame o pripojenie našej implementácie ku knižnici JOPA.

Kapitola 6
Vyhodnotenie

V tejto kapitole porovnáme dotazy v jazyku SOQL s dotazmi Semantic Criteria API a vyhodnotíme správnosť implementácie rozhrania. Pred tým si ešte povieme niečo o tom, ako implementáciu testujeme.

6.1 Testy

Prvou kontrolou implementácie sú triedy obsahujúce jednotkové testy. Kontrolujú správnosť metód triedy CriteriaBuilder a správnosť generovania SOQL dotazu, pri jednoduchých a komplexných dotazoch. Ukážku jednotkového testu vidíme na výpise nižšie.

```
public void testTranslateQuerySelectAll() {
    CriteriaQuery<OWLClassA> query = cb.createQuery(OWLClassA.class);
    Root<OWLClassA> root = query.from(OWLClassA.class);
    query.select(root);

    final String generatedJSoqlQuery =
            ((CriteriaQueryImpl<OWLClassA>) query).translateQuery(parameterFiller);
    final String expectedSoqlQuery = "SELECT owlclassa FROM OWLClassA owlclassa";
    assertEquals(expectedSoqlQuery, generatedSoqlQuery);
}
```

Listing 6.1: Jednotkový test

Druhou a dôležitejšou kontrolou je trieda CriteriaRunner s integračnými testami, ktorá opäť obsahuje jednoduché, ale aj komplexné dotazy. Tie prechádzajú od procesu vytvorenia až po porovnávania výsledkov. Ukážka integračného testu vidíme na výpise nižšie.

```
public void testSimpleCount() {
        final List<OWLClassA> expected = QueryTestEnvironment.getData(OWLClassA.class);
        CriteriaBuilder cb = getEntityManager().getCriteriaBuilder();
        CriteriaQuery<Integer> query = cb.createQuery(Integer.class);
        Root<OWLClassA> root = query.from(OWLClassA.class);
        query.select(cb.count(root));
        final Integer result = getEntityManager().createQuery(query).getSingleResult();

        assertEquals(expected.size(), result);
    }
```

Listing 6.2: Integračný test

6.2 Porovnanie

V tabuľke 6.1 vidíme porovnanie SOQL výrazov s odpovedajúcimi metódami v triedach CriteriaBuilder a CriteriaQuery v Semantic Criteria API.

SOQL výraz	CriteriaQuery metóda	CriteriaBuilder metóda
FROM	from()	
SELECT	select()	
DISTINCT	distinct()	
WHERE	where()	
AND		and()
OR		or()
NOT		not()
=		equal()
<>		notEqual()
>		greaterThan()
>=		greaterThanOrEqual()
<		lessThan()
<=		lessThanOrEqual()
LIKE		like()
NOT LIKE		notLike()
COUNT		count()
IN		in()
NOT IN		notIn()
GROUP BY	groupBy()	
ORDER BY	orderBy()	
ASC		asc()
DESC		desc()

Tabuľka 6.1: SOQL vs. Semantic Criteria API

Na vyhodnotenie správnosti implementácie sme vytvorili dotazy v Semantic Criteria API a ekvivalentné dotazy v SOQL. Dotazy pokryli celú funkcionalitu, ktorú SOQL ponúka a boli vykonané na reálnych dátach. Porovnali sme výsledky dotazov a zistili sme, že výsledky sú rovnaké. Nižšie a na ďalšej strane môžeme vidieť štyri zo siedmich spomínaných dotazov.

```
CriteriaQuery<Game> query = cb.createQuery(Game.class);
Root<Game> g = query.from(Game.class);
ParameterExpression<Integer> amount = cb.parameter(Integer.class);
Path<Integer> pathJoin = g.getAttr(Game_.developer).getAttr(Developer_.employeeCount);
query.select(g).where(cb.lessThanOrEqual(pathJoin, amount));
/* ---------------------------------------------------------- */
SELECT g FROM Game g WHERE g.developer.employeeCount <= :amount
```

Listing 6.3: Porovnanie 1

```
CriteriaQuery<Integer> query = cb.createQuery(Integer.class);
Root<Developer> d = query.from(Developer.class);
ParameterExpression<Integer> amount = cb.parameter(Integer.class);
Predicate restriction = cb.greaterThanOrEqual(d.getAttr(Developer_.employeeCount), amount);
query.select(cb.count(d)).where(restriction);
/* ---------------------------------------------------------- */
SELECT COUNT(d) FROM Developer d WHERE d.employeeCount >= :amount
```

Listing 6.4: Porovnanie 2

```
CriteriaQuery<Developer> query = cb.createQuery(Developer.class);
Root<Developer> d = query.from(Developer.class);
ParameterExpression<String> name = cb.parameter(String.class);
ParameterExpression<Integer> amount = cb.parameter(Integer.class);
Predicate nameRestriction = cb.like(d.getAttr(Developer_.name), name);
Predicate amountRestriction = cb.lessThan(d.getAttr(Developer_.employeeCount),amount);
query.select(d).where(nameRestriction,amountRestriction);
/* ---------------------------------------------------------- */
SELECT d FROM Developer d
            WHERE d.employeeCount < :amount AND d.name LIKE :name
```

Listing 6.5: Porovnanie 3

```
CriteriaQuery<Developer> query = cb.createQuery(Developer.class);
Root<Developer> d = query.from(Developer.class);
ParameterExpression<Integer> param1 = cb.parameter(Integer.class);
ParameterExpression<Integer> param2 = cb.parameter(Integer.class);
Predicate firstRestr = cb.greaterThanOrEqual(d.getAttr(Developer_.employeeCount), param1);
Predicate secondRestr = cb.lessThanOrEqual(d.getAttr(Developer_.employeeCount), param2);
Predicate restrictions = cb.or(firstRestr,secondRestr);
Order firstOrder = cb.desc(d.getAttr(Developer_.employeeCount));
Order secondOrder = cb.asc(d.getAttr(Developer_.name));
query.select(d).where(restrictions).orderBy(firstOrder, secondOrder);
/* ---------------------------------------------------------- */
SELECT d FROM Developer d
            WHERE d.employeeCount >= :param1 OR d.employeeCount <= :param2
            ORDER BY d.employeeCount DESC, d.name ASC
```

Listing 6.6: Porovnanie 4

6.3 Výsledok

Implementáciu Semantic Criteria API hodnotíme ako správnu v rozsahu funkcionality SOQL. To sme preukázali porovnaním výsledkov ekvivalentných dotazov. Funkcionalita, ktorú SOQL zatiaľ nepodporuje, generuje dotaz s predpokladanou podobou zhodnou so syntaxou JPQL. Predpokladáme teda, že aj implementácia týchto funkcionalít bude správna ak ich začne SOQL podporovať.

Kapitola 7

Záver

V práci sme sa zoznámili s princípmi sémantického webu. Vyjasnili sme si pojmy a predstavili technológie, ktoré so sémantickým webom súvisia. Pozreli sme sa aj na štandardné a neštandardné technológie zo sveta relačných dát, ktoré sa využívajú pri dotazovaní dát. Predstavili sme si knižnicu JOPA, ktorá sa snaží štandardné postupy a funkcionality zo sveta relačných dát aplikovať aj v svete sémantických dát. Tým sme splnili prvý z cieľov.

Následne sme analyzovali štandardné programovateľné dotazovacie rozhranie Criteria API a jeho neštandardnú alternatívu QueryDSL. Zhodnotili sme silné a slabšie stránky oboch rozhraní. Identifikovali sme konštrukty, ktoré sú aplikovateľné iba v rámci relačných dát a uvedomili si potrebu nových konštruktov aplikovateľných na sémantické dáta.

Po splnení druhého cieľa sme sa pustili do návrhu vlastného dotazovacieho rozhrania, ktoré by sa malo stať súčasťou knižnice JOPA. Navrhli sme, aby sa dotaz vytvorený pomocou rozhrania prekladal do objektového dotazovacieho jazyka SOQL, ktorý knižnica JOPA následne prekladá do nativného dotazovacieho jazyka SPARQL. Pokúsili sme sa navrhnúť štruktúru dotazov podobnú štruktúre QueryDSL. Zistili sme, že takýto návrh vyžaduje prerobenie alebo nanovo vybudovanie metamodelu, ktorý knižnica JOPA využíva. Prehodnotili sme teda prvotný návrh, ktorý by nás mohol odkloniť od pôvodných cieľov práce, a rozhodli sme, že štruktúra dotazu bude inšpirovaná rozhraním Criteria API. Po predstavení verejnej štruktúry dotazu podľa JPA špecifikácie sme navrhli internú štruktúru dotazu budúceho programovateľného dotazovacieho rozhrania.

Podľa návrhu sme potom naimplementovali programovateľné dotazovacie rozhranie a integrovali ho do knižnice JOPA. Ku implementácií sme vytvorili jednotkové testy overujúce správne generovanie prekladu a integračné testy, v ktorých prebieha celý proces dotazovania. Teda od vytvárania dotazu až po získavanie výsledkov. Okrem testov sme overili správnosť implementácie porovnaním výsledkov dotazov Semantic Criteria API a ekvivalentných dotazov v SOQL na reálnych dátach. Splnili sme teda aj posledné ciele našej práce.

Semantic Criteria API sa teda môže stať súčasťou ďalšej verzie knižnice JOPA a umožní tak využívať ďalší efektívny spôsob dotazovania dát užívateľom knižnice. Použiteľnosť rozhrania je síce čiastočne limitované funkcionalitou SOQL, ale vývoj tohto objektového dotazovacieho jazyka je plánovaný. Vývoj SOQL už môže ísť ruka v ruke s vývojom Semantic Criteria API. Rozhranie by v budúcnosti mohlo byť rozšírené o zložitú projekciu, vytváranie poddotazov alebo dotazovanie nepovinnej hodnoty. Budúcnosť by mala samozrejme zahŕňať aj opravu prípadných chýb, ktoré neboli odchytené testami.

Dodatok A

Literatúra

[1] BERNERS-LEE, TIM, JAMES HENDLER a ORA LASSILA. THE SEMANTIC WEB: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*. May 2001, **2001**(5), 34-43. ISSN 0036-8733. Dostupné tiež z: https://www.jstor.org/stable/26059207

[2] Criteria API. KEITH, Mike a Merrick SCHNICARIOL. *Pro JPA 2: Mastering the Java™ persistence API*. New York: Apress, c2009, s. 239-271. The expert's voice in Java technology. ISBN 978-1-4302-1957-6.

[3] KŘEMEN, Petr a Zdeněk KOUBA. Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* [online]. May 2012, **42**(3), 334-344 [cit. 2020-12-18]. ISSN 1094-6977. Dostupné z: doi:10.1109/TSMCC.2011.2163934

[4] Introduction. DAVIES, John, Dieter FENSEL a Frank van HARMES. *TOWARDS THE SEMANTIC WEB: Ontology-driven Knowledge Management*. Chichester: John Wiley, 2003, s. 4. ISBN 0-470-84867-7.

[5] GRUBER, Thomas R. A translation approach to portable ontology specifications. *Knowledge Acquisition*. 1993, **5**(2), 199-220. ISSN 10428143. Dostupné z: doi:10.1006/knac.1993.1008

[6] GRUBER, Tom. Ontology. In: LIU, Ling a M. Tamer ÖZSU, ed. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, 2009, s. 1963-1965. ISBN 978-0-387-35544-3. Dostupné z: doi:10.1007/978-0-387-39940-9_1318

[7] STANDARDS: ABOUT W3C STANDARDS. *W3C* [online]. World Wide Web Consortium [cit. 2020-12-20]. Dostupné z: https://www.w3.org/standards/about.html

[8] WEISS, Michael. Resource Description Framework. In: LIU, Ling a M. Tamer ÖZSU, ed. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, 2009, s. 2423-2425. ISBN 978-0-387-35544-3. Dostupné z: doi:10.1007/978-0-387-39940-9_905

[9] CYGANIAK, Richard, David WOOD a Markus LANTHALER. RDF 1.1 Concepts and Abstract Syntax: W3C Recommendation. *W3C* [online]. World Wide Web Consortium, 25 February 2014 [cit. 2020-11-14]. Dostupné z: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[10] SCHREIBER, Guus a Yves RAIMOND. RDF 1.1 Primer. *W3C* [online]. World Wide Web Consortium, 24 June 2014 [cit. 2020-11-05]. Dostupné z: https://www.w3.org/TR/rdf11-primer/

[11] BRICKLEY, Dan a Ramanathan V. GUHA. RDF Schema 1.1: W3C Recommendation. *W3C* [online]. World Wide Web Consortium, 25 February 2014 [cit. 2020-12-07]. Dostupné z: https://www.w3.org/TR/rdf-schema/

[12] HITZLER, Pascal, Markus KRÖTZSCH, Bijan PARSIA, Peter F. PATEL-SCHNEIDER a Sebastian RUDOLPH. OWL 2 Web Ontology Language Primer: W3C Recommendation. *W3C* [online]. World Wide Web Consortium, 11 December 2012 [cit. 2020-12-07]. Dostupné z: https://www.w3.org/TR/owl2-primer

[13] PATEL-SCHNEIDER, Peter F. a Boris MOTIK. OWL 2 Web Ontology Language Mapping to RDF Graphs: W3C Recommendation. *W3C* [online]. World Wide Web Consortium, 11 December 2012 [cit. 2020-12-12]. Dostupné z: https://www.w3.org/TR/owl2-mapping-to-rdf/

[14] HARRIS, Steve a Andy SEABORNE. SPARQL 1.1 Query Language: W3C Recommendation. *W3C* [online]. World Wide Web Consortium, 21 March 2013 [cit. 2020-11-14]. Dostupné z: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[15] Learn SPARQL: Write Knowledge Graph queries using SPARQL with step-by-step examples. *STARDOG* [online]. Stardog Union, 03 December 2018 [cit. 2020-11-14]. Dostupné z: https://www.stardog.com/tutorials/sparql/

[16] Introduction, Getting Started. KEITH, Mike a Merrick SCHNICARIOL. *Pro JPA 2: Mastering the Java™ persistence API*. New York: Apress, c2009, s. 1-32. The expert's voice in Java technology. ISBN 978-1-4302-1957-6.

[17] Query Language. KEITH, Mike a Merrick SCHNICARIOL. *Pro JPA 2: Mastering the Java™ persistence API*. New York: Apress, c2009, s. 207-219. The expert's voice in Java technology. ISBN 978-1-4302-1957-6.

[18] Kbss-cvut/jopa: JOPA - Java OWL Persistence API. *Github* [online]. Praha: Knowledge Based Software Systems Group, 18 Feb 2016 [cit. 2020-11-16]. Dostupné z: https://github.com/kbss-cvut/jopa/blob/master/README.md

[19] LEDVINKA, Martin, Bogdan KOSTOV a Petr KŘEMEN. JOPA: Efficient Ontology-Based Information System Design. *The Semantic Web: ESWC 2016 Satellite Events*. Cham: Springer International Publishing, 2016, 2016-10-20, (9989), 156-160. Lecture Notes in Computer Science. ISBN 978-3-319-47601-8. Dostupné z: doi:10.1007/978-3-319-47602-5_31

[20] Kbss-cvut/jopa/wiki: Semantic Object Query Language. *Github* [online]. Knowledge Based Software Systems Group, 2018 [cit. 2020-11-11]. Dostupné z: https://github.com/kbss-cvut/jopa/wiki/Semantic-Object-Query-Language

[21] Criteria API. SUN MICROSYSTEMS. *JSR 317: JavaTM Persistence API, Version 2.0* [online]. JCP, 2009, s. 197-284 [cit. 2021-5-5]. Dostupné z: https://download.oracle.com/otn-pub/jcp/persistence-2.0-fr-eval-oth-JSpec/persistence-2_0-final-spec.pdf

[22] PRUD'HOMMEAUX, Eric a Alexandre BERTAILS. *A Mapping of SPARQL Onto Conventional SQL* [online]. World Wide Web Consortium [cit. 2021-5-9]. Dostupné z: https://www.w3.org/2008/07/MappingRules/StemMapping

[23] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, c1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[24] WESTKÄMPER, Timo, Samppa SAARELA, Vesa MARTTILA, Lassi IMMONEN, Ruben DIJKSTRA, John TIMS a Robert BAIN. *Querydsl Reference Guide* [online]. 4.4.0. The Querydsl Team, c2007-2016 [cit. 2020-10-25]. Dostupné z: http://www.querydsl.com/static/querydsl/4.4.0/reference/html_single/

Dodatok B

Zoznam skratiek

- WWW - World Wide Web
- API - Application Programming Interface
- SQL - Structured Query Language
- UML - Unified Modeling Language
- RDF - Resource Description Framework
- IRI - Internationalized Resource Identifier
- RDFS - Resource Description Framework Schema
- OWL - Web Ontology Language
- SPARQL - SPARQL Protocol and RDF Query Language
- JPA - Java Persistence API
- JPQL - Java Persistence Query Language
- JOPA - Java OWL Persistence API
- SOQL - Semantic Object Query Language

Dodatok C

Elektronická príloha

Elektronická príloha a celá práca je zverejnená v *Digitální knihovna ČVUT* na portáli dspace.cvut.cz. Príloha je textový súbor s názvom F3-BP-2021-Zec-Marcel-priloha-odkazy.txt, ktorý obsahuje odkazy na ostatné elektronické prílohy. Odkazy na ostatné elektronické prílohy sú následovné:

- Pull request
  https://github.com/kbss-cvut/jopa/pull/91
- Fork repozitár s knižnicou JOPA obsahujúci implementáciu
  https://github.com/marcel-zec/jopa
- Posledný commit z fork repozitára s implementáciou
  https://github.com/marcel-zec/jopa/tree/11f9a4f5b29f216f7cb0dc2cd2068690665169bf
- Projekt porovnávajúci Semantic Criteria API a SOQL
  https://github.com/marcel-zec/semantic-criteria-api-vs-soql