ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kuzevanov** Jméno: **Igor** Osobní číslo: **434798**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Webový klient pro správu katalogů SPARQL dotazů

Název bakalářské práce anglicky:

Web Client for the management of SPARQL Catalogs

Pokyny pro vypracování:

1) Proveďte průzkum existujících řešení:
a) Proveďte rešerše existujících Webových nástrojů pro správu katalogů SPARQL dotazů
b) Popište existující datové modely pro reprezentaci katalogů SPARQL dotazů
c) Popište existujících REST rozhraní pro správu katalogů SPARQL dotazů.
Příklad existujících řešení: RDF4J Workbench, GraphDB, Wikidata.
2) Definujte případy použití nástroje pro správu a prohledávaný katalogů SPARQL dotazů. Formulujte funkční a nefunkční požadavky.
3) Definujte návrh nástroje a prezentujte jej v jazyce UML.
4) Implementujte webového klienta v JavaScript frameworku React. Dále:
a) otestujte tvorbu a správu katalogů,
b) otestujte škálovatelnost na několika větších veřejně dostupných SPARQL katalogů, např. 'Wikidata:SPARQL query service/queries/examples' a porovnejte její výhody ve srovnání s existujícími řešení z bodu 1a.
5) Proveďte validaci použitelnosti pomoci testu bez uživatelů a s uživateli.

Seznam doporučené literatury:

[1] Nielsen, J. Usability Engineering. Morgan Kaufmann, 1993.
[2] Dumas, J., Loring, B. Moderating Usability Tests. Morgan Kaufmann, 2008.
[3] Kuniavsky, M. Observing the User Experience. Morgan Kaufmann, 2003
[4] Courage, C., Baxter, K. Understanding Your User. Morgan Kaufmann, 2005.
[5] SPARQL language specification - konkretní literatura bude upřesněná
[6] Knowledge Organization Systems - konkretní literatura bude upřesněná

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Bogdan Kostov, skupina znalostních softwarových systémů FEL

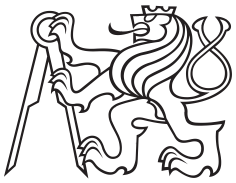
Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **07.01.2020**

Platnost zadání bakalářské práce: **20.09.2020**

Ing. Bogdan Kostov podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická

Webový klient pro správu katalogů SPARQL dotazů

Igor Kuzevanov

Školitel: Ing. Bogdan Kostov
Leden 2020

Poděkování

V první řadě bych chtěl poděkovat vedoucímu této práce panu Ing. Bogdanu Kostovovi za jeho rady, zpětnou vazbu a pravidelnou pomoc během realizace této práce. Dále bych chtěl poděkovat své matce za poskytnutí duševní a finanční podpory a samozřejmě České republice a Českému vysokému učení technickému za poskytnutí možností dosažení vysokoškolského vzdělání.

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.

Abstrakt

Cílem této práce bylo vytvořit webový klient pro správu katalogů SPARQL dotazů. V první polovině jsem se věnoval průzkumu existující řešení, jejich nevýhod a nevýhod. Poté jsem navrhl vylepšenou verzi webového klienta. V druhé části jsem se zabýval implementací webového klienta a jeho testováním včetně testů použitelnosti.

Klíčová slova: semantický web, Linked data, SPARQL, React, webový klient

Školitel: Ing. Bogdan Kostov
Praha, Resslova 307/9, místnost: E-113

Abstract

The goal of this thesis was to implement web client for the management of SPARQL catalogs. In the first part of the thesis I described existing solutions to the problem and analyzed their advantages and disadvantages. After that I designed my own solution. In the second part of the thesis I implemented web client and conducted usability testing.

Keywords: semantic web, Linked data, SPARQL, React, web client

Title translation: Web Client for the management of SPARQL Catalogs

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Již na konci 20. století počet stránek na webu neustále rostl a tím komplikoval vyhledávání relevantní informace. Proto v roce 2001 ředitel konsorcia W3C Tim Berners-Lee přišel s myšlenkou sémantického webu, který je rozšířením současného webu, v němž data mají přidělen dobře definovaný význam lépe umožňující počítačům a lidem spolupracovat. Je ale důležité říct, že sémantický web není jenom o přidělení významu, ale také o propojení dat mezi sebou, což by umožňovalo jednoduché nalezení souvisejících informací.

Pro splnění výše uvedených požadavků byl v roce 2006 zaveden koncept Linked Data [2]. Podrobnější popis bude uveden v další kapitole, prozatím je důležité vědět, že jedním z cílů jeho vytvoření bylo umožnění práce s daty na webu jako s relační databází a dotazování na jejich obsah pomocí jazyků podobných SQL. Jedním z nichž je doporučený konsorciem jazyk SPARQL (SPARQL Protocol and RDF Query Language) [1] .

Nechť Linked Data katalog je uzel v sémantickém webu, obsahující informace ve formátu RDF (Resource Description Framework) [14] na základě příslušné tomuto uzlu ontologie (příkladem je projekt DBpedia[7]). Podobně tomu, jak SQL databáze uchovávají data na základě relačních schémat, Linked Data katalog uchovává data podle některé předem určené ontologie.
Dále zadefinujme pojem katalogu SPARQL dotazů - Linked Data katalog, který obsahuje množinu SPARQL dotazů, kde jednotlivé dotazy mohou mít název, textový popis apod. a mohou být kategorizováne.
Příklady katalogů SPARQL dotazů:

- Katalog vzorových a poučných SPARQL dotazů pro práci s Wikidata [13].
- LSQ (Linked SPARQL Queries) [12] - katalog metadat o SPARQL dotazech získaných z logů různých veřejných SPARQL endpointů.

Cílem této práce je potom vyvinout webový klient pro správu katalogů SPARQL dotazů.

1.1 Aktuální stav

V současné době existují softwarová řešení, která umožňují správu SPARQL dotazů, neumožňují ale současnou správu kategorizaci těchto dotazů.

1.2 Cíle práce

Definujeme cíle práce jako obecné požadavky na webový klient, které později upřesníme v příslušné kapitole. Webový klient bude mít následující vlastnosti:

- CRUD operace nad množinou SPARQL dotazů v katalogu
- CRUD operace nad množinou kategorií SPARQL dotazu
- Třídění a zobrazení dotazů na základě zvolené množiny kategorií

Kapitola 2
Prostředí

V úvodu jsme zmínili Linked data - důležitý pojem pro pochopení problematiky této práce. Nyní jej popíšeme podrobněji a také některé související definice.

2.1 RDF

Resource Description Framework 1.1 (RDF) [14] je grafový datový model, který je součástí koncepce sémantického webu a reprezentuje tvrzení o zdrojích ve formátu "subjekt - predikát - objekt". Důležitou vlastností tohoto formátu je čitelnost jak lidsky, tak i strojově. Pro pojmenování subjektu, predikátu nebo objektu se používá URI. Množina RDF-tvrzení potom tvoří orientovaný graf, kde vrcholy jsou subjekty a objekty a hrany jsou predikáty.

2.2 Linked data

Nevýhoda klasického webu spočívá v tom, že jeho data nemají vedle sebe kontext o informaci, kterou reprezentují. Dva datové objekty totiž můžou mít vlastnosti se stejným názvem, ale odlišným významem. Linked data je způsob řešení tohoto problému. Model Linked Data předpokládá, že data jsou publikována jako datasety ve formátu RDF a jsou strukturována na základě slovníků (ontologií) a jednotlivé datasety jsou propojené pomocí hypertextových odkazů. Samotný dataset může být složen z více RDF grafů.

2.3 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) [1] je sémantický dotazovací jazyk nad daty ve formátu RDF a je hodně podobný jazyku SQL. SPARQL umožňuje dotazovaní pomocí vzorů RDF grafů, kde uzly nebo hrany můžou být nahrazeny proměnnou. Níže uvedený příklad vrátí všechny dvojice (subjekt a objekt) obsahující predikát “rdfs:label”. Neboli vrátí všechny dvojice vrcholů, které jsou spojeny hranou “rdfs:label”.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
WHERE { ?entity rdfs:label ?name . }
```

Pro čtení dat SPARQL podporuje 4 základní typy dotazů:

- SELECT - výstupem je tabulka, obsahující data vyhovující dotazu.

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?nameX ?nameY ?nickY
  WHERE
    { ?x foaf:knows ?y ;
         foaf:name ?nameX .
      ?y foaf:name ?nameY .
      OPTIONAL { ?y foaf:nick ?nickY }
    }
  ```

- CONSTRUCT - výstupem je RDF graf zkonstruovaný na základě šablony grafu.

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
  CONSTRUCT { <http://example.org/person#Alice> vcard:FN ?name }
  WHERE { ?x foaf:name ?name }
  ```

- ASK - výstupem je odpověď (boolean), zda existuje řešení pro dotaz.

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  ASK  { ?x foaf:name  "Alice" }
  ```

- DESCRIBE - výstupem je RDF graf popisující nalezený zdroj.

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  DESCRIBE ?x
  WHERE { ?x foaf:mbox <mailto:alice@org> }
  ```

Existují také dvě důležité klauzule:

- GRAPH - umožňuje specifikovat graf z datasetu, v němž se má vyhodnotit grafový vzor.

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>

  SELECT ?src ?bobNick
  FROM NAMED <http://example.org/foaf/aliceFoaf>
  FROM NAMED <http://example.org/foaf/bobFoaf>
  WHERE
    {
  ```

```
  GRAPH ?src
  { ?x foaf:mbox <mailto:bob@work.example> .
    ?x foaf:nick ?bobNick
  }
}
```

- SERVICE - umožňuje dotazovaní na data v dalším autonomním datasetu.

```
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT *
  {
    SERVICE wikibase:label { ?v wikibase:language "en". }
  }
```

2.4 YASQE

YASQE (Yet Another Sparql Query Editor)[11] je grafický editor pro práce se SPARQL dotazy postavený na komponentě CodeMirror. Podporuje zvýraznění syntaxe a mechanismus našeptávání kódu.

2.5 React Ace

Ace je grafický editor zdrojového kódu. Podporuje zvýraznění syntaxe pro více jazyků a mechanismus našeptávání kódu. Existuje také verze editoru React Ace, která obsahuje stejnou funkcionalitu, ale je dostupná pro použití jako React komponenta.

2.6 Intelligent Tree Data Management Component

Intelligent Tree Data Management Component[8] je React komponenta pro zobrazení stromu objektů. Je schopna zobrazovat velký počet položek aniž by snížila výkonnost aplikace nebo zhoršila UX.

2.7 React

React[4] je knihovna pro vývoj grafických uživatelských rozhraní v jazyce JavaScript. Její základním objektem je komponenta, která je potomkem React třídy Component a může obsahovat další komponenty. Celá aplikace je potom stromem komponent.
Data uvnitř komponenty ukládáme do vnitřní proměnné “state”, kterou za běhu můžeme libovolně měnit. Přenos dat mezi komponenty se realizuje

pomocí “props” [1] - při inicializaci komponenty-potomka ji můžeme předat libovolná data z rodičovské komponenty. “Props” se mění pouze ve chvílí, kdy dojde ke změně “state” rodičovské komponenty. Abychom mohli změnit “state” z komponenty-potomka, musíme do něj předat odkaz na callback-metodu, po jejíž vyvolání se provedou potřebné změny na úrovní předka.
Ve většině případu pro manipulaci se “state” se používají další knihovny: Flux, Redux, Mobx. V této práci jsem zvolil pro tento účel framework Redux.

[1] Zkrátka od “Properties”

Kapitola 3

Rešerše existujících řešení

Cílem této kapitoly bylo prozkoumat existující nástroje a modely pro správu katalogů SPARQL dotazů. Rešerše byla rozdělena do 2 fází:

- Rešerše existujících webových nástrojů pro správu katalogů SPARQL dotazů
- Rešerše existujících datových modelů pro reprezentaci katalogů SPARQL dotazů

3.1 Webové nástroje

První částí této kapitoly byla rešerše existujících webových nástrojů pro správu katalogů SPARQL dotazů. Na začátku jsem uvedl seznam nástrojů, které se dále analyzovaly a jejich stručný popis. Poté byla provedena srovnávací analýza vlastností těchto nástrojů (viz Tabulka 3.6) a to vzhledem k cílem práce:

- CRUD operace nad množinou SPARQL dotazů v katalogu
- CRUD operace nad množinou kategorií SPARQL dotazu
- Třídění a zobrazení dotazů na základě zvolené množiny kategorií

Dále byly u každého nástroje uvedeny vlastnosti, které pozitivně nebo negativně ovlivňují uživatelskou zkušenost (UX, user experience), ale jsou mimo cíle této práce. Účelem analýzy těchto vlastnosti byla snaha implementovat v dalších fázích této práce co nejvíc pozitivních a vyhnout se těm negativním.

3.1.1 YASGUI

YASGUI je webový grafický klient pro práci se SPARQL dotazy. Jádrem tohoto klientu je javascript-komponenta YASQE (Yet Another Sparql Query Editor)[11] postavená na komponentě CodeMirror, podporující zvýraznění syntaxe a mechanismus našeptávání kódu. Pro zovrazování výsledků používá komponentu YASR (Yet Another Sparql Resultset GUI).

Výhody	Nevýhody
■ Podporuje zvýraznění syntaxe. ■ Podporuje našeptávání kódu. ■ Podporuje práci s několika dotazy najednou pomocí záložek. ■ Podporuje spuštění dotazů a zobrazování výsledků v několika formátech (tabulka, RAW data ve formátu JSON, kontingenční tabulka, Google Chart).	

Tabulka 3.1: Další vlastnosti YASGUI

3.1.2 Wikidata

Wikidata je podpůrný projekt Wikipedie pro realizaci společného úložiště dat, které následně využíváno jinými projekty. Wikidata má strukturu velmi podobnou Wikipedii. Pro tuto práci je ale důležité použití stránek Wikidata jako katalogu SPARQL dotazů, což je demonstrováno na stránce s kategorizovaným seznamem poučných SPARQL dotazů[13].

Výhody	Nevýhody
■ Primitivní rozhraní umožňuje použití bez přípravy skoro každým uživatelem. ■ Jednotlivé dotazy mají vedle stručný popis jejich obsahu.	■ Všechna data jsou na jedné webové stránce, což může vést ke snížení doby odezvy aplikace při vysokém počtu dotazů a komplikuje orientaci uvnitř těla stránky.

Tabulka 3.2: Další vlastnosti Wikidata

3.1.3 GraphDB

GraphDB[5] je nástroj pro správu grafových databází, které implementují RDF a SPARQL specifikace. Jádrem editoru SPARQL dotazů jsou YASQE[11] a YASR komponenty, které jsem popisoval v kapitole 2.1.

Výhody	Nevýhody
■ Podporuje zvýraznění syntaxe. ■ Podporuje našeptávání kódu. ■ Podporuje práci s několika dotazy najednou pomocí záložek. ■ Podporuje spuštění dotazů a zobrazování výsledků v několika formátech (tabulka, RAW data ve formátu JSON, kontingenční tabulka, Google Chart). ■ Podporuje uložení a nahrávání dotazů (uvnitř úložiště webového klienta). ■ Podporuje generování URL pro sdílení SPARQL dotazů.	■ Proprietární software.

Tabulka 3.3: Další vlastnosti GraphDB

3.1.4 BASIL

BASIL[3] je cloud platforma pro sdílení a přepoužití SPARQL dotazů, která generuje Web API pro uložené SPARQL dotazy. Specifikace API zahrnuje SPARQL endpoint, SPARQL dotaz a volitelně postprocessingový skript.

Výhody	Nevýhody
■ Generuje Web API pro každý uložený SPARQL dotaz. ■ Umožňuje spuštění dotazu přímo z prohlížeče (pomocí vygenerováného URL).	

Tabulka 3.4: Další vlastnosti BASIL

3.1.5 RDF4J Workbench

RDF4J Workbench je webový klient pro správu úložišť SPARQL dotazů unvitř RDF4J Serveru.

Výhody	Nevýhody
	■ Nelze zkopírovat dotaz bez přechodu do režimu editace.

Tabulka 3.5: Další vlastnosti RDF4J Workbench

	Nástroje				
	YASGUI	Wikidata	GraphDB	BASIL	RDF4J
Přidání dotazu	Ano	Ano	Ano	Ano	Ano
Zobrazení dotazu	Ano	Ano	Ano	Ano	Ano
Změna dotazu	Ano	Ano	Ano	Ano	Ano
Smazání dotazu	Ano	Ano	Ano	Ne	Ano
Přidání kategorie	Ne	Ano	Ne	Ne	Ne
Zobrazení kategorie	Ne	Ano	Ne	Ne	Ne
Změna kategorie	Ne	Ano	Ne	Ne	Ne
Smazání kategorie	Ne	Ne	Ne	Ne	Ne
Třídění podle zvolené množiny kategorií	Ne	Ano	Ne	Ne	Ne

Tabulka 3.6: Výsledky srovnávací analýzy.

3.2 Datové modely

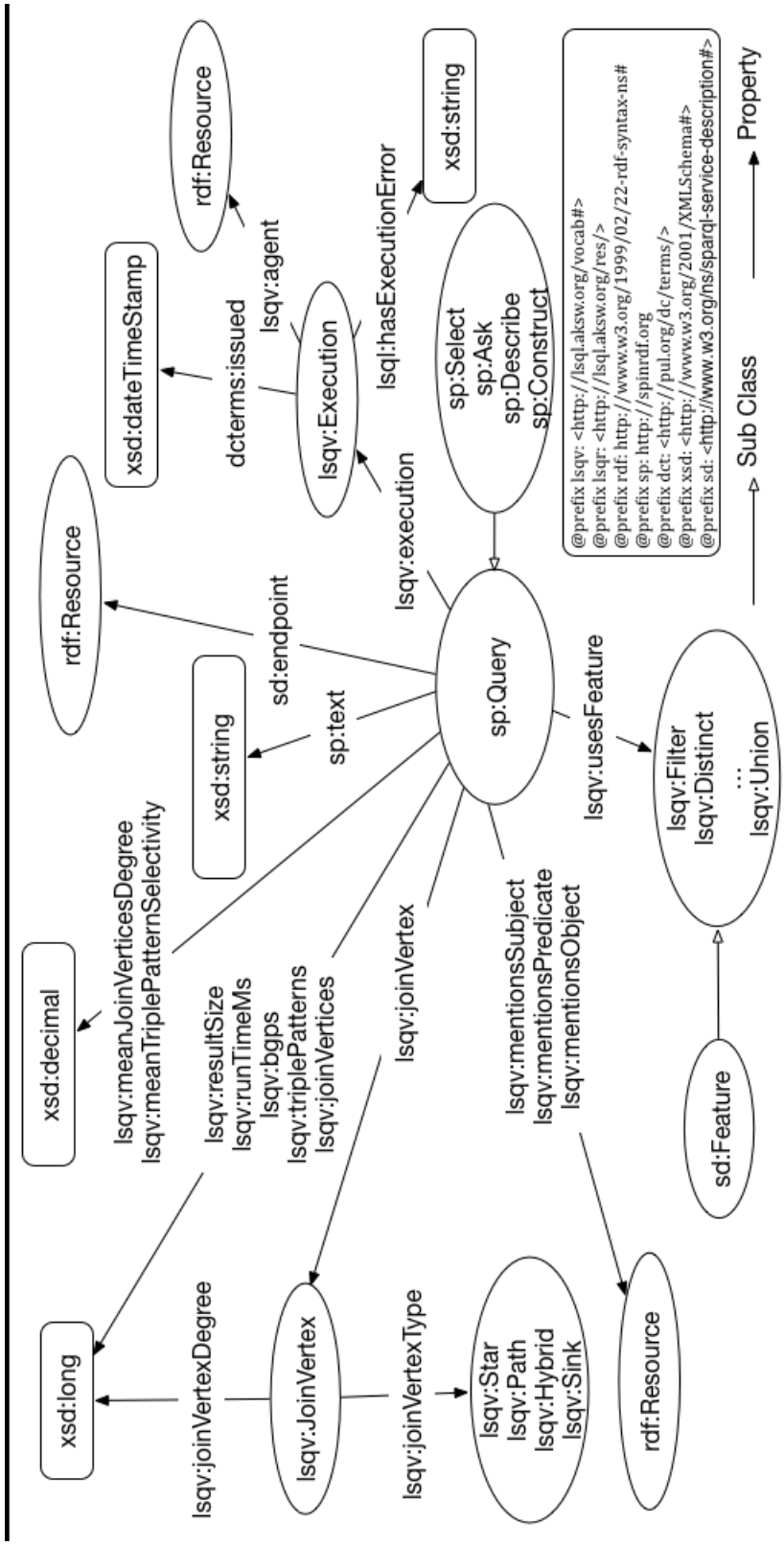
Druhou částí této kapitoly byla rešerše existujících datových modelů pro reprezentaci katalogů SPARQL dotazů.
Protože ve specifikaci SPARQL Query Language[1] vydanou konsorciem W3C neexistuje jediný standard datového modelu pro katalog SPARQL dotazů, zkoumal jsem různorodé modely existujících aplikací a datasetů.

3.2.1 Wikidata

V kapitole 3.1.2 jsem popisoval projekt Wikidata, který obsahuje katalog SPARQL dotazů ve formě wiki stranky. Datový model je v tomto případě text doplněný wiki-syntaxí. Jde o minimální možný model, který nevyhovuje potřebám této práce.

3.2.2 LSQ

LSQ (Linked SPARQL Queries) [12] je Linked Dataset, popisující SPARQL dotazy získané z logů různých veřejných SPARQL endpointů. Data uvnitř tohoto datasetu jsou kategorizované podle modelu na obrázku 3.1. Smyslem této kategorizací je, že každý SPARQL dotaz uložen spolu se svými vlastnostmi, například, jaký má typ (SELECT, ASK, DESCRIBE, CONSTRUCT), jestli obsahuje klíčové slovo Filter, Disctinct, Union a další.



Obrázek 3.1: LSQ model

3.3 Závěr

3.3.1 Webový klient

Ze všech webových klientů, které byly analyzovány v rámci této kapitoly, pouze Wikidata má funkcionalitu pro práci s kategorizací SPARQL dotazů. Nevýhodou Wikidata je ale její primitivní realizace, která není optimální pro práci s velkým počtem dotazů a to jak ve smyslu výkonností aplikace, tak i v pohodlnosti pro uživatele. Každý dotaz ale obsahuje vedle krátký popis svého obsahu, což na jednu stranu dovoluje uživateli vyhledávat dotazy na stránce pomocí přirozeného jazyku, na druhou stranu udává uživateli informaci o tom, k čemu tento dotaz slouží. V případě velkých dotazů (až 45 řádků) je tento popis velmi užitečný.

Dále během analýzy bylo zjištěno, že všechny aplikace kromě Wikidata podporují spuštění dotazů a zobrazení jeho výsledků, proto můžu implikovat, že tato funkcionalita je pro uživatele kritická.

Nástroje YASGUI a GraphDB navíc podporují zvýraznění syntaxe jazyka SPARQL a mechanismus našeptávání kódu, což je podle mého názoru výborná vlastnost těchto webových klientů a v současné době je de facto standardem pro editory zdrojového kódu.

Nakonec byl rozsah funkcionalit webového klientu rozšířen o následující prvky:

- SPARQL dotaz má vedle sebe textový popis svého obsahu.
- Dotaz je možné spustit a zobrazit jeho výsledky.
- Je možné zvolit SPARQL-endpoint, proti kterému bude dotaz spuštěn.
- Editace kódu dotazu podporuje zvýraznění syntaxe (pro jazyk SPARQL) a mechanismus našeptávání kódu.

3.3.2 Datový model

Protože existující datové modely, které se mi podařilo prozkoumat v kapitole 3.2, nevyhovují cílem této práce, mnou byl navrhnut vlastní model v kapitole 4.3.

Kapitola 4

Návrh

V souladu se základy softwarového inženýrství rozlišujeme dvě podstatné skupiny požadavků: funkční a nefunkční. Dále každou z nich lze popsat z pohledů různých členů cílové skupiny uživatelů: vývojář, který tento webový klient nasazuje, a koncový uživatel, jehož cílem je práce se samotnými SPARQL dotazy.

4.1 Požadavky

4.1.1 Funkční

Pro uživatele:

1. Webový klient podporuje následující operace nad SPARQL dotazem:
   a. Přidání
   b. Zobrazení
   c. Změna
   d. Smazání
   e. Spuštění (a zobrazení výsledků)
2. Webový klient podporuje volbu SPARQL-endpointu, proti kterému budou jednotlivé dotazy spouštěny.
3. Webový klient podporuje následující operace nad množinou kategorií jednotlivého SPARQL dotazu:
   a. Přidání kategorie
   b. Zobrazení množiny kategorií
   c. Smazání kategorie
4. Webový klient podporuje filtraci seznamu zobrazených dotazů na základě zvolené množiny kategorií. Filtrovat se bude konjunktivním způsobem - budou zobrazeny pouze dotazy, které sdílejí všechny zvolené kategorie.

Pro vývojáře:

1. Konfigurace adresy REST API

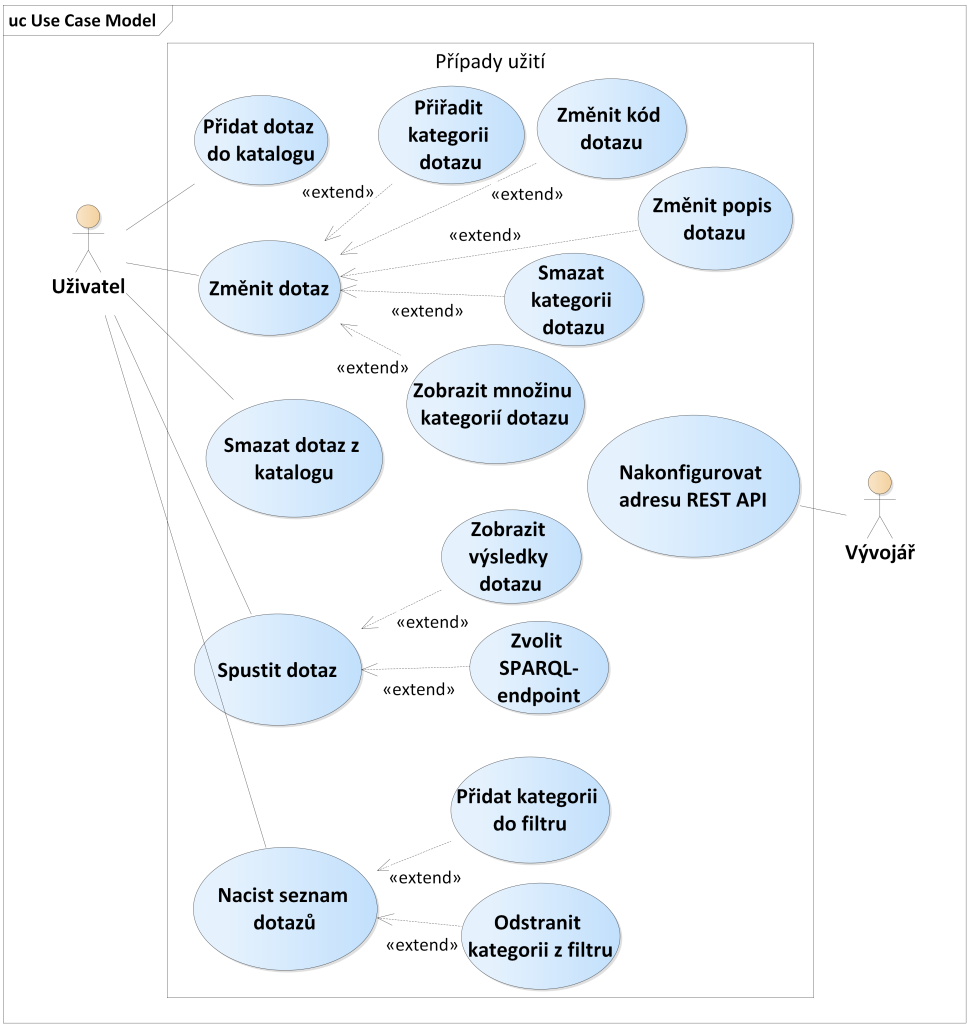
4.1.2 Nefunkční

Pro uživatele:

1. Uživateli je zobrazeno tlačítko pro přechod na stránku přidaní nového dotazu do katalogu.
2. Zobrazený pomocí filtrace dotaz je needitovatelný a obsahuje vedle sebe needitovatelný seznam jeho kategorií.
3. Renderování seznamu dotazů nesmí trvat příliš dlouho ani zpomalovat další práci s webovým klientem.
4. Zobrazený dotaz má vedle sebe tlačítko pro přechod na stránku spuštění dotazu a zobrazení výsledků.
5. Zobrazený dotaz má vedle sebe tlačítko pro smazaní dotazu z katalogu.
6. Zobrazený dotaz má vedle sebe tlačítko pro přechod na stránku editace dotazu a množiny jeho kategorií.
7. Stránka pro editaci dotazů podporuje:
   a. Zobrazení stromu všech kategorií, z něhož uživatel je schopen přiřazovat dotazu další kategorie.
   b. Smazaní existujících kategorií dotazu.
   c. Editaci kódu dotazu včetně zvýrazňování syntaxe a našeptávání.

4.2 Případy užití

Na základě požadavků uvádím i nejdůležitější případy užití webového klientu. Dva aktéry jsou uživatel, u něhož zkoumáme práci se samotným webovým klientem, a vývojář, který je zodpovědný za parametrizaci klientu a jeho nasazení.



Obrázek 4.1: Diagram případů užití

4.3 Datový model

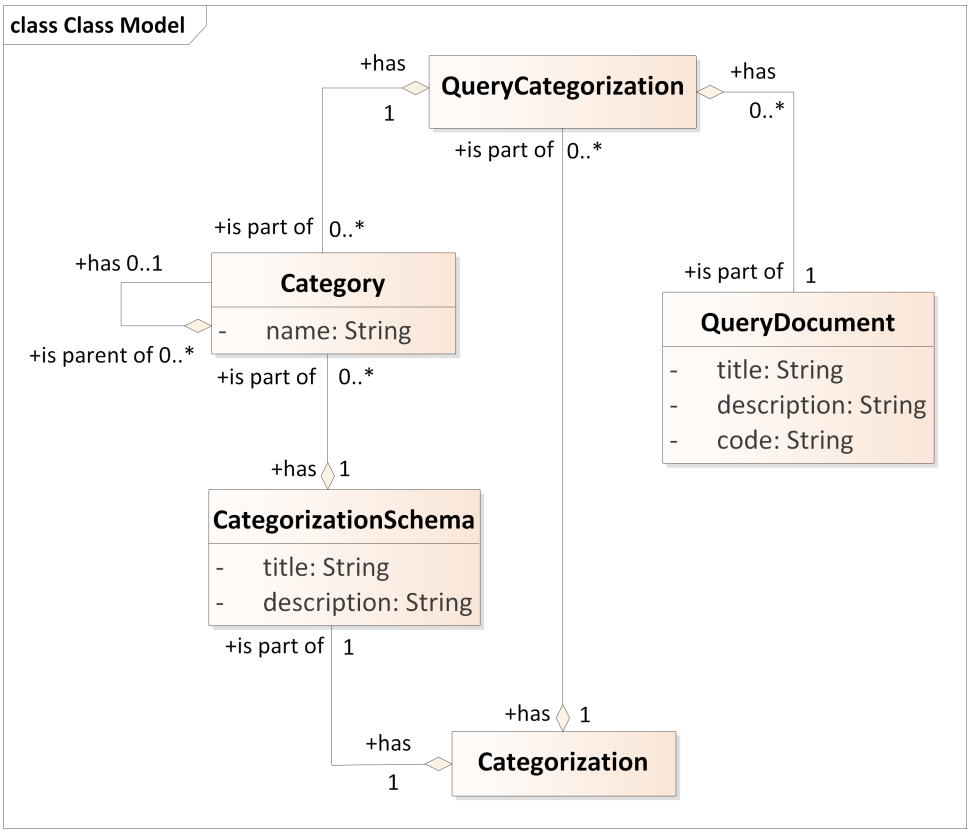
Definuji datový model, který dále budu uvažovat při implementaci jednotlivých částí webového klientu. Protože hlavní účel nástroje je práce s katalogy SPARQL dotazů, datový model bude takový katalog reprezentovat.

- **QueryDocument** reprezentuje SPARQL dotaz - jeho kód - a další informace kolem něj: název a textový popis.
- **Category** reprezentuje kategorii dotazu. Kategorie vždy má název a může, ale nemusí mít rodiče.
- **CategorizationSchema** reprezentuje množinu kategorií typu Category. Má také název a textový popis.
- **QueryCategorization** reprezentuje kategorizaci jednotlivého dotazu.

Skládá se z objektu SPARQL dotazu (QueryDocument) a množiny kategorií typu Category.

- **Categorization** reprezentuje kategorizaci množiny SPARQL dotazů (QueryDocumentList). Pojmem "kategorizace množiny" se myslí množina kategorizací její dotazů (QueryCategorization)

Na základě výše popsáného datového modelu byl mnou sestrojen příslušný diagram, kde jsou přesně definovány vztahy mezi entitami a popsány jejich atributy.



Obrázek 4.2: Diagram datového modelu

4.4 Diagram komponent

V návaznosti na seznam případů užití a datový model z předchozích kapitol definuji rozhraní, které propojí webový klient a REST API. Jsou potřeba následující metody na stráně webového klientu:

- `void createQueryCategoriation(QueryCategoroization)`

  uloží novou kategorizaci dotazu (**QueryCategorization**) v katalogu.

- `void deleteQueryCategorization(long)`

smaže existující kategorizaci dotazu (**QueryCategorization**) z katalogu podle id.

- `void fetchCategories(long)`

  načte seznam kategorií z katalogu podle id kategorizace (**Categorization**).

- `void fetchCategorizations()`

  načte seznam kategorizací(**Categorization**) z katalogu.

- `void fetchQueryCategorizations(long, List<Category>)`

  načte konjunktivně seznam kategorizací dotazů(**QueryCategorization**) podle id kategorizace (**Categorization**) a seznamu kategorií(**Category**).

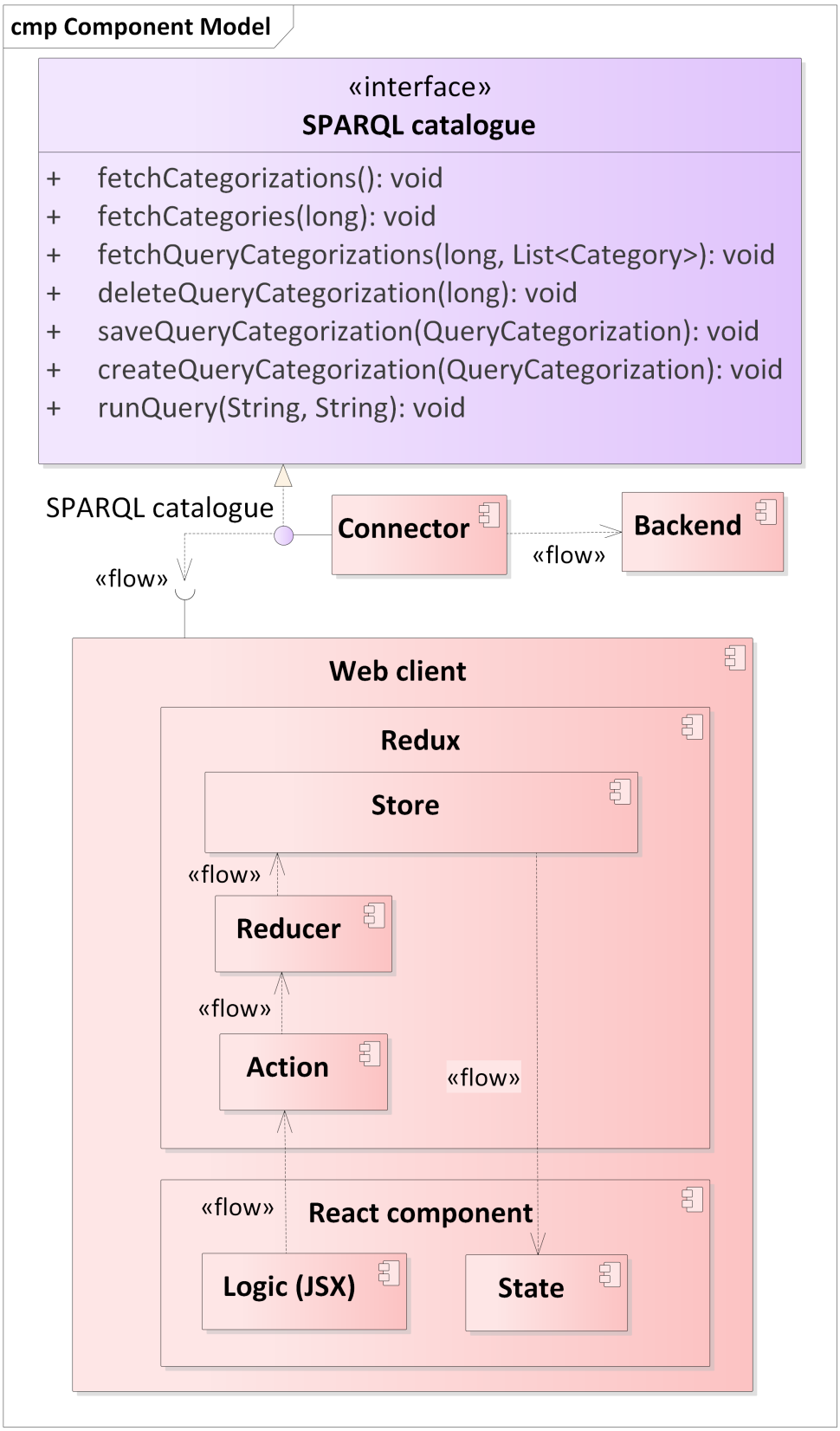
- `void runQuery(String, String)`

  spustí SPARQL dotaz (2. argument) proti SPARQL-endpointu (1. argument).

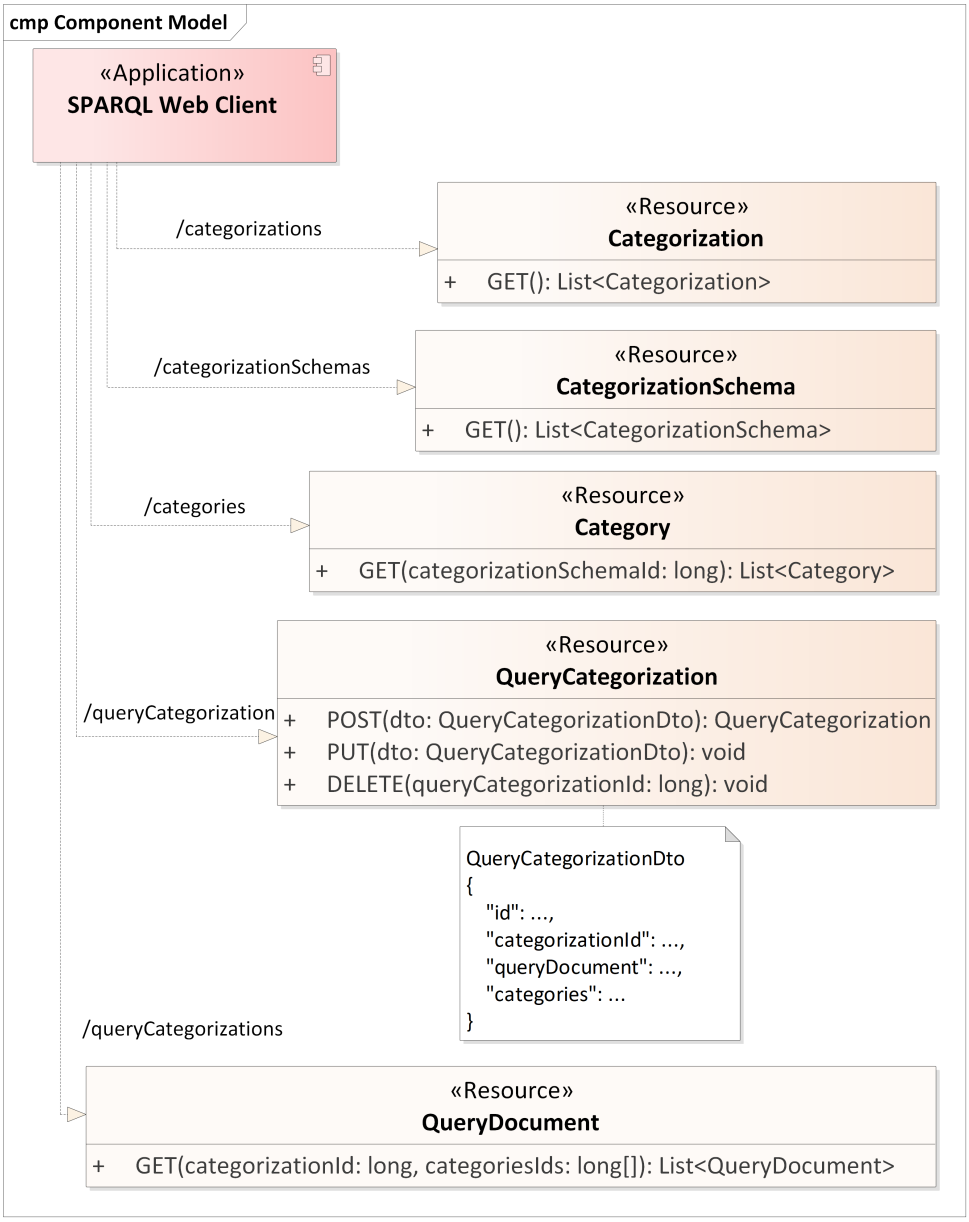
- `void updateQueryCategorization(QueryCategorization)`

  uloží změněnou kategorizaci dotazu (**QueryCategorization**) v katalogu.

Každá z výše uvedených metod modifikuje "state"aplikace a zasílá požadavek na REST API. Podrobný popis REST API rozhraní je na diagramu 4.4. Princip toku dat webového klientu je zobrazen na diagramu 4.3.



Obrázek 4.3: Diagram komponent



Obrázek 4.4: REST API model

4.5 Sekvenční diagramy

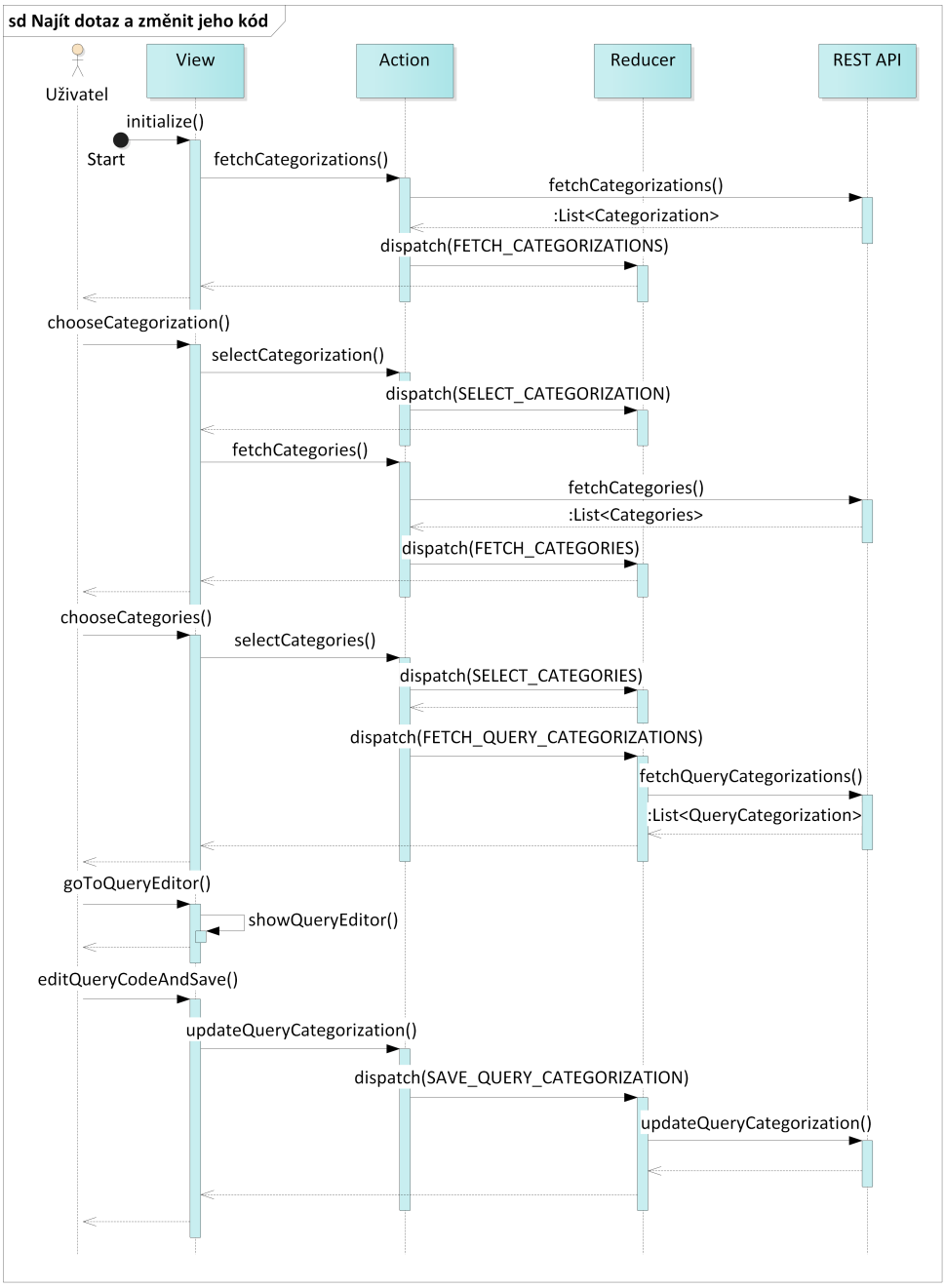
Teď když mám představu o tom, z čeho se výsledný webový klient skládá, proberu její nejdůležitější procesy a to pomocí sekvenčních diagramů. Budu se snažit pokryt co nejvíce případů užití a zároveň popisovat jenom klíčové procesy.

4.5.1 Vyhledávání a změna dotazu

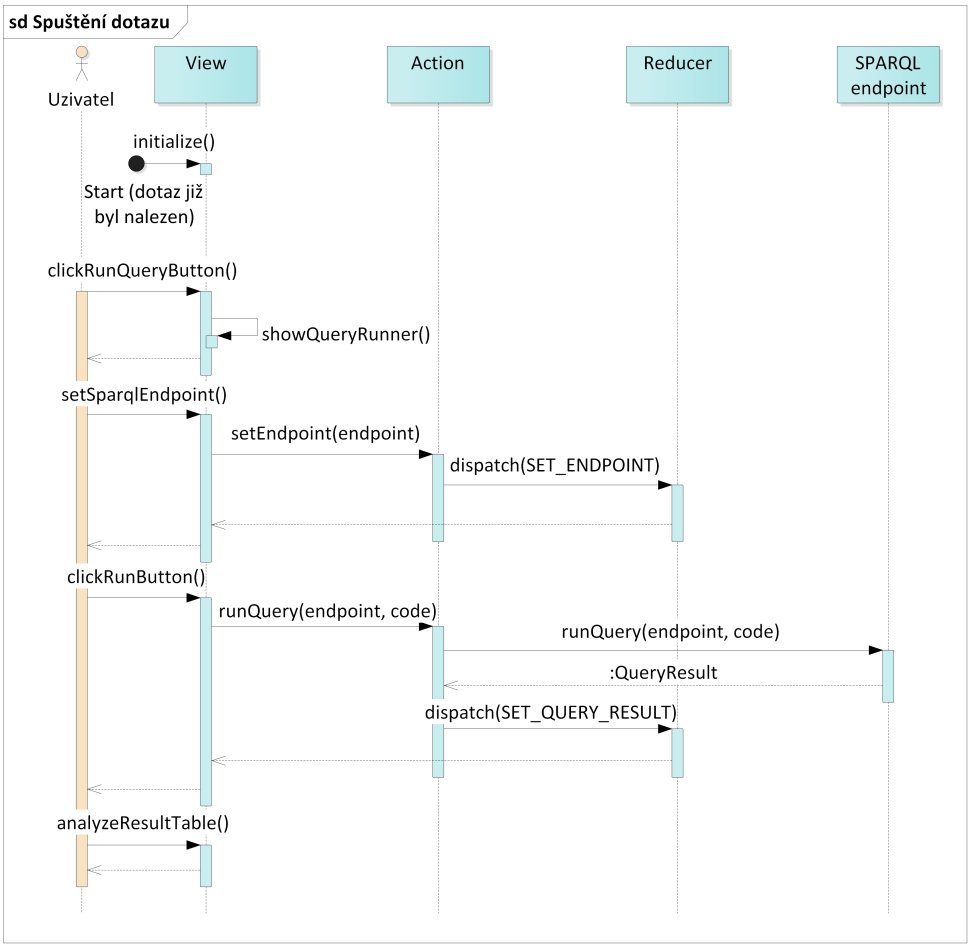
Hned po inicializaci webového klientu se zavolá metoda pro načtení dostupných kategorizací (**Categorization**) z REST API. Uživatel je poté schopen zvolit libovolnou kategorizaci, čímž se spustí metoda pro načtení stromu kategorií (**Category**) z REST API pro zvolenou kategorizaci (**Categorization**). Nyní je uživatel schopen filtrovat dotazy pomocí množiny zvolených kategorií. Jakmile uživatel najde potřebný dotaz, pomocí příslušného tlačítka přejde na obrazovku pro editaci dotazu, kde po provedení požadované změny uloží dotaz (**QueryCategorization**) a tím zašle modifikovaný dotaz na příslušný REST API endpoint. Tento proces je podrobně popsán na diagramu aktivit 4.5.

4.5.2 Spuštění dotazu a analýza výsledků

Po nalezení SPARQL dotazu (**QueryCategorization**) uživatel může přejít na obrazovku pro spuštění tohoto dotazu, kde nejprve zadá SPARQL-endpoint, proti kterému dotaz poběží, a následně jej spustí. Jakmile dotaz bude úspěšně dokončen bude uživateli zobrazena tabulka s výsledky. Podrobný popis tohoto procesu od okamžiku nalezení dotazu je popsán na diagramu 4.6.



Obrázek 4.5: Diagram vyhledávání a změna dotazu



Obrázek 4.6: Diagram spuštění dotazu

Kapitola 5
Implementace

V této kapitole popíšu nástroje, které budu používat během implementace a netriviální řešení, které přímo neplynou z návrhu.

5.1 Prostředí

Pro implementaci webového klientu potřebuji pouze správce balíčků NPM pro import knihoven a IDE podporující syntaxi React JSX, což v mém případě bude IntelliJ IDEA.

5.2 State management

Protože manipulace se stavem webového klientu provádím pomocí frameworku Redux, dále uvádím stručný jeho komponent.

5.2.1 Store

Store je centrální uložiště dat v aplikaci, proto do něj budu ukládat pouze data, která jsou zapotřebí v různých částech aplikace a nejsou spojeny přímo vazbou rodič-potomek. Počateční globálním stav (**Store**) bude mít následující strukturu:

```
const initialState = {
    runnerReducer: {
        endpoint: "https://query.wikidata.org/sparql"
    }
};
```

5.2.2 Actions

Veškeré manipulace s globálním stavem aplikace jsou realizovány pomocí volání **Action** metod. Tyto metody provádí potřebnou logiku a vytváří událost určitého typu, která dále mění stav v příslušném objektu **Reducer** na základě aktuálního stavu a dat předaných z **Action**.

5.2.3 Reducers

Po úspěšném dokončení metoda z **Action** vytvoří událost určitého typu, do které také přidá data potřebná pro změnu stavu v **Store**. Tato událost je zpracována pomocí objektu **Reducer**. Je důležité uvést, že jediné místo, kde se může změnit **Store** je příslušný **Reducer**.

5.3 Test Driven Development

Test Driven Development (TDD) je proces vývoje softwarů, kde vývojář se zabývá psaním automatických testů ještě před implementací logiky systému. TDD je rozděleno do 5 fázi:

1. Vytvoření testu.
2. Spuštění testů a kontrola, že nově vytvořené testy neprochází.
3. Implementace testované logiky.
4. Spuštění testů a kontrola, že testy, které neprošli v bodě 2, nyní úspěšně doběhly.
5. Refaktorování logiky implementované v bodě 3.

Sice TDD vede ke zvětšení množství zdrojového kódu, má výhodu v tom, že celá logika projektu je zabezpečená proti chybám jak při podpoře, tak při provedení změnových řízení. Toto nakonec vede ke snížení celkové pracnosti. TDD má největší přínos u projektu, kde jsou požadavky pevně specifikovány a nebudou se v budoucnosti příliš často měnit. U projektu, kde dochází k častým změnám požadavků, TDD může teoreticky mít i negativní vliv na výslednou pracnost.
Protože proces vývoje webového klientu v této práci probíhá sekvenčně, metoda TDD bude velmi užitečná.

5.4 Unit testy

Veškeré unit testy budou psány pomocí frameworku Jest[6]. Test bude mít následující strukturu:

```
describe('název souboru s testovacími scénáři', () => {
    it('popis jednotlivého scénáře', () => {
        expect(/*hodnota z testované metody*/)
        .toEqual(/*očekávaná hodnota*/);
    });
    //další scénáře...
}
```

Protože celá logika aplikace bude probíhat v **Actions** a **Reducers**, soustředím se na jejich pokrytí testy.

5.4.1 Actions testování

Veškeré Actions se dají rozdělit do dvou základních skupin:

1. Action, který tvoří událost, do níž přidá vstupní data.
2. Action, který zavolá REST API a vytvoří událost, do níž přidá stažená data.

Test pro Action z první skupiny bude mít následující tvar (test pro událost CLEAN_STORE, viz 5.5.1):

```
it('creates CLEAN_STORE when store has been cleaned', () => {
    const expectedActions = [
        {
            type: 'CLEAN_STORE'
        }
    ];
    const store = mockStore({});
    store.dispatch(cleanStore());
    expect(store.getActions()).toEqual(expectedActions)
});
```

Test pro Action z druhé skupiny bude mít navíc mockovací metodu (test pro událost FETCH_CATEGORIZATIONS, viz 5.5.1):

```
it('creates FETCH_CATEGORIZATIONS when fetching categorizations has been done',
    () => {
    fetchMock.getOnce(`${Constants.REST_API}/categorizations`, {
        body: JSON.stringify([]),
        headers: {'content-type': 'application/json'}
    });
    const expectedActions = [
        {
            type: 'FETCH_CATEGORIZATIONS',
            data: []
        }
    ];
    const store = mockStore({categorizations: []});
    return store.dispatch(fetchCategorizations()).then(() => {
        expect(store.getActions()).toEqual(expectedActions)
    })
})
```

5.4.2 Reducers testování

Testování Reducerů je triviální a má stejnou strukturu:

```
it('popis jednotlivého scénáře', () => {
    expect(
```

```
        reducer(/* původní stav */, {
            type: /* typ testované události */,
            data: /* data */
        })
    ).toEqual(/* nový stav */);
})
```

5.5 Implementace logiky

5.5.1 Actions implementace

explorerAction.js

explorerAction.js - implementuje metody pro načtení kategorizací a kategorií z REST API. Implementuje také metody pro zpracování zvolené kategorizace a zvolených kategorií a speciální metodu **cleanStore**, která je určena pro návrat globálního stavu do původní formy.

```
export const fetchCategories = (categorizationSchemaId) => dispatch => {
    return fetch(`${Constants.REST_API}/categories
        ?categorizationSchemaId=${categorizationSchemaId}`)
        .then(response => response.json())
        .then(json => dispatch(
            {type: "FETCH_CATEGORIES", data: json}))
        .catch(err => dispatch(
            {type: "ERROR", msg: "Unable to fetch data", e: err}))
};

export const fetchCategorizations = () => dispatch => {
    return fetch(`${Constants.REST_API}/categorizations`)
        .then(response => response.json())
        .then(json => dispatch(
            {type: "FETCH_CATEGORIZATIONS", data: json}))
        .catch(err => dispatch(
            {type: "ERROR", msg: "Unable to fetch data", e: err}))
};

export const selectCategorization = (selectedCategorization) => dispatch => {
    dispatch({
        type: 'SELECT_CATEGORIZATION',
        selectedCategorization: selectedCategorization
    })
};

export const selectCategories = (selectedCategories) => dispatch => {
    dispatch({
        type: 'SELECT_CATEGORIES',
```

```
        selectedCategories: selectedCategories
    })
};

export const cleanStore = () => dispatch => {
    dispatch({
        type: 'CLEAN_STORE'
    })
};
```

queryEditorAction.js

explorerAction.js - implementuje metody pro vytvoření nových kategorizovaných dotazů (**QueryCategorization**) a změnu existujících.

```
export const updateQueryCategorization = (queryCategorization) => dispatch => {
    return fetch(`${Constants.REST_API}/queryCategorization`, {
        method: "PUT",
        body: JSON.stringify(queryCategorization),
        headers: {"Content-Type": "application/json", "Accept": "application/json"}
    })
        .then(() => dispatch({
            type: "SAVE_QUERY_CATEGORIZATION",
            data: {
                status: "ok",
                updatedQueryCategorization: queryCategorization
            }
        }))
        .catch(err => dispatch({type: "ERROR", msg: "Unable to fetch data", e: err}))
};

export const createQueryCategorization = (queryCategorization) => dispatch => {
    return fetch(`${Constants.REST_API}/queryCategorization`, {
        method: "POST",
        body: JSON.stringify(queryCategorization),
        headers: {"Content-Type": "application/json", "Accept": "application/json"}
    })
        .then(response => response.json())
        .then(json => dispatch({
            type: "CREATE_QUERY_CATEGORIZATION",
            data: {
                status: "ok",
                createdQueryCategorization: json
            }
        }))
        .catch(err => dispatch(
            {type: "ERROR", msg: "Unable to fetch data", e: err}))
```

```
};
```

queryListAction.js

queryListAction.js - implementuje metodu pro načtení kategorizovaných dotazů podle zvolené uživatelem kategorizace a seznamu zvolených kategorií.

```
export const fetchQueryDocuments = (categorizationId, categories) => dispatch =>
    if (categorizationId && categories) {
        return fetch(`${Constants.REST_API}/queryCategorizations
            ?categorizationId=${categorizationId}
            &categoriesIds=${categories.map(category => category.id)}`)
            .then(response => response.json())
            .then(json => dispatch(
                {type: "FETCH_QUERY_CATEGORIZATIONS", data: json}))
            .catch(err => dispatch(
                {type: "ERROR", msg: "Unable to fetch data", e: err}))
    }
};
```

runnerAction.js

runnerAction.js - implementuje metodu pro uložení zadaného uživatelem SPARQL-endpointu, proti kterému budou spuštěny dotazy, metodu pro smazaní výsledku SPARQL dotazu, které již nejsou relevantní a metodu pro spuštění dotazu.

```
export const setEndpoint = (endpoint) => dispatch => {
    dispatch({
        type: 'SET_ENDPOINT',
        endpoint: endpoint
    })
};

export const deleteQueryResult = () => dispatch => {
    dispatch({
        type: 'DELETE_QUERY_RESULT'
    })
};

export const runQuery = (endpoint, query) => dispatch => {
    console.log(`Sending ${query} to ${endpoint}`);
    return fetch(endpoint, {
        method: "POST",
        headers: {"Content-Type": "application/x-www-form-urlencoded", "Accept":
        body: `${encodeURIComponent("query")}=${encodeURIComponent(query)}`
    })
        .then(response => response.json())
```

```
        .then(json => dispatch({
            type: 'SET_QUERY_RESULT',
            data: json
        }))
        .catch(err => dispatch(
            {type: "ERROR", msg: "Unable to fetch data", e: err}))
};
```

queryAction.js

queryAction.js - implementuje metodu pro uložení kódu SPARQL dotazu pro spuštění do globálního stavu a metodu pro smazaní kategorizace dotazu (**QueryCategorization**) z katalogu.

```
export const setQueryToRun = (queryToRun) => dispatch => {
    dispatch({
        type: 'SET_QUERY_TO_RUN',
        queryToRun: queryToRun
    })
};

export const deleteQueryCategorization = (queryCategorizationId) => dispatch => {
    return fetch(`${Constants.REST_API}/queryCategorization?queryCategorizationId=${quer
        method: "DELETE",
        headers: {"Content-Type": "application/json", "Accept": "application/json"}
    })
        .then(() => dispatch({
            type: "DELETE_QUERY_CATEGORIZATION",
            data: queryCategorizationId
        }))
        .catch(err => dispatch(
            {type: "ERROR", msg: "Unable to fetch data", e: err}))
};
```

5.5.2 Reducers implementace

rootReducer

rootReducer je hlavní **Reducer** aplikace a je rodičem dalších čtyř Reducerů. Samotný rootReducer zpracovává pouze jednu událost - vyčištění **Store**.

```
const appReducer = combineReducers({
    explorerReducer,
    queryListReducer,
    queryReducer,
    runnerReducer
});
```

```
export default (state, action) => {
    if (action.type === 'CLEAN_STORE') {
        state = undefined
    }
    return appReducer(state, action)
};
```

explorerReducer

explorerReducer zpracovává události: načtení kategorizací, načtení kategorií, volba kategorizace uživatelem, volba kategorie uživatelem. Každá událost je zpracována triviální způsobem - data se uloží do globálního stavu beze změn.

```
export default (state = {}, action) => {
    switch (action.type) {
        case 'FETCH_CATEGORIES':
            return {
                ...state,
                categories: action.data
            };
        case 'FETCH_CATEGORIZATIONS':
            return {
                ...state,
                categorizations: action.data
            };
        case 'SELECT_CATEGORIZATION':
            return {
                ...state,
                selectedCategorization: action.selectedCategorization
            };
        case 'SELECT_CATEGORIES':
            return {
                ...state,
                selectedCategories: action.selectedCategories
            };
        default:
            return state
    }
}
```

queryListReducer

queryListReducer zpracovává události: načtení kategorizací dotazů, smazání kategorizace dotazu, uložení kategorizace dotazu, vytvoření nové kategorizace dotazu.
V případě načtení kategorizací dotazů jde jenom o uložení seznamu kategorizací (**QueryCategorization**) do stavu. Událost "smazání"najde příslušnou

kategorizaci v seznamu kategorizací dotazů a smaže ji.
Událost "uložení"najde příslušnou kategorizaci a nahradí ji.
Událost "vytvoření"přidá novou kategorizaci do seznamu kategorizací dotazů.

```
export default (state = {}, action) => {
    switch (action.type) {
        case 'FETCH_QUERY_CATEGORIZATIONS':
            return {
                ...state,
                queryCategorizations: action.data
            };
        case 'DELETE_QUERY_CATEGORIZATION':
            let temp = [];
            for (let i = 0; i < state.queryCategorizations.length; ++i) {
                if (state.queryCategorizations[i].id !== action.data) {
                    temp.push(state.queryCategorizations[i]);
                }
            }
            return {
                ...state,
                queryCategorizations: temp
            };
        case 'SAVE_QUERY_CATEGORIZATION':
            let queryCategorizations = [];
            for (let i = 0; i < state.queryCategorizations.length; ++i) {
                if (state.queryCategorizations[i].id === action.data.updatedQueryCategor
                    queryCategorizations.push(action.data.updatedQueryCategorization);
                } else {
                    queryCategorizations.push(state.queryCategorizations[i]);
                }
            }
            return {
                ...state,
                queryCategorizations: queryCategorizations
            };
        case 'CREATE_QUERY_CATEGORIZATION':
            let queryCategorizations2 = [];
            for (let i = 0; i < state.queryCategorizations.length; ++i) {
                queryCategorizations2.push(state.queryCategorizations[i]);
            }
            queryCategorizations2.push(action.data.createdQueryCategorization);
            return {
                ...state,
                queryCategorizations: queryCategorizations2
            };
        default:
```

```
            return state
    }
}
```

runnerReducer

runnerReducer zpracovává 4 událost:

- Uložení dotazu pro spuštění.
- Uložení SPARQL-endpointu.
- Uložení výsledků SPARQL dotazu.
- Smazaní výsledků SPARQL dotazu.

```
export default (state = {}, action) => {
    switch (action.type) {
        case 'SET_QUERY_TO_RUN':
            if (state.queryToRun === action.queryToRun) {
                return state;
            } else {
                return {
                    ...state,
                    queryToRun: action.queryToRun,
                    queryResult: undefined
                }
            }
        case 'SET_ENDPOINT':
            return {
                ...state,
                endpoint: action.endpoint
            };
        case 'SET_QUERY_RESULT':
            let headers = action.data.head.vars;
            let rows = action.data.results.bindings;
            let formattedResult = {columns: [], rows: []};
            for (let i = 0; i < headers.length; ++i) {
                formattedResult.columns[i] =
                    {
                        label: headers[i],
                        field: headers[i],
                        sort: 'asc',
                        width: 150
                    }
            }
            for (let i = 0; i < rows.length; ++i) {
                let formattedRow = {};
```

```
                for (let head of headers) {
                formattedRow[head] = rows[i][head] ? rows[i][head].value : " "
                }
                formattedResult.rows[i] = formattedRow;
            }
            return {
                ...state,
                queryResult: formattedResult
            };
        case 'DELETE_QUERY_RESULT':
            return {
                ...state,
                queryResult: undefined
            };
        default:
            return state
    }
}
```

5.6 REST API

Sice implementace REST API pro správu katalogu SPARQL dotazů není přímo součástí této práce, pro účely testování (kapitola 6) bylo takové API implementováno. API se řidí modelem z kapitoly 4.3 a splňuje rozhraní z kapitoly 4.4.

Kapitola 6

Testování

Množinu softwarových testů můžeme rozdělit do dvou základních skupin: funkční a nefunkční. Do první skupiny obvykle patří jednotkové, integrační, systémové, akceptační, regresní, smoke testy a testy uživatelského rozhraní. Do druhé potom spadají například testy odezvy a použitelnosti, zátěžové a penetrační testy. Jelikož funkční testy byli základem implementace webového klientu a již byly vytvořeny v kapitole 5, budu se nyní zabývat jenom nefunkčními testy. Protože webový klient neřeší otázky bezpečností, budu se věnovat pouze testům výkonu a použitelnosti.

6.1 Testování výkonu

Protože zvětšení počtu kategorizovaných dotazů v katalogu může vést k zpomalení webového klientu, provedl jsem testování jeho výkonu. Testovací scénář byl následující:

1. Zvolit kategorizaci
2. Zvolit jakoukoliv množinu kategorií
3. Porovnat čas renderování grafického rozhraní klientu pro katalogy s 3/10/100/1000 dotazů.

Výsledky porovnání jsou uvedeny v tabulce 6.1, ze které je vidět, že doba renderování grafického rozhraní není závislá na počtu dotazů v katalogu a je dostatečně malá.

Počet dotazů v katalogu	Doba renderování
3	4ms
10	3.4ms
100	6.4ms
1000	2.7ms

Tabulka 6.1: Doba renderování v závislosti na počtu dotazů

6.2 Testování použitelnosti

Podle návrhu Jakoba Nielsena[9] všechny testy použitelnosti rozdělím na dvě větší skupiny: s uživatelem a bez uživatele. Sice existuje velké množství testů z obou skupin, zvolíl jsem si pouze jeden typ testu pro každou ze skupin.

6.2.1 Testovací data

Pro testování byl vytvořen testovací katalog dotazů. Základem tohoto katalogu byly poučovací dotazy z Wikidata [13].

6.2.2 Testování bez uživatelů

Testování bez uživatelů (používám jako překlad pro "usability inspection"[10]) je skupinou metod testování, kde inspektor (osoba provádějící test) zkoumá uživatelské rozhraní. Tyto metody jsou:

- Kognitivní průchod - metoda zjišťuje míru náročností rozhraní pro splnění uživatelem předem definovaného úkolu. V každém kroku úkolu inspektor položí 3 otázky a odpoví na ně:
  - Q1: Je uživateli zřejmé, co má udělat?
  - Q2: Spojí si uživatel popis akce s danou akcí?
  - Q3: Dostane uživatel dostatečnou zpětnou vazbu?
- Heuristická evaluace - metoda zjišťuje problémy použitelnosti tak, že inspektor zkontroluje, jestli prvek rozhraní není v rozporu se seznamem heuristik.

Průběh testování

Webový klient jsem testoval pomocí kognitivního průchodu na základě seznamu úkolů, kde každý úkol obsahoval scénář jeho správného splnění.

- Úkoly.
  - Spustit dotaz, který vrací všechna muzea ve Velké Británii.
    1. Zvolit kategorizaci "Wikidata tutorial"
    2. Zvolit kategoii "Britain"

3. Kliknout na tlačítko "Run query"
4. Kliknout na tlačítko "Run"
5. Zkontrolovat výsledky

- Změnit kód dotazu pro načtení katedrál v Paříži a přidat jej novou kategorii “Coordinates”.
  1. Zvolit kategorizaci "Wikidata tutorial"
  2. Zvolit kategorii "Paris"
  3. Kliknout na tlačítko "Edit query"
  4. Libovolně změnit kód dotazu.
  5. Přidat dotazu kategorii “Coordinates”.
  6. Uložit změnu.
- Najít změněný dotaz pomocí nově přidané kategorie.
  1. Zvolit kategorizaci "Wikidata tutorial"
  2. Zvolit kategorii "Coordinates"
  3. Najít v seznamu kategorizovaných dotazů dotaz pro načtení katedrál v Paříži. Dotaz má obsahovat právě 4 kategorie: "Culture", "Churches", "Paris", "Coordinates"

Úkol	Q1	Q2	Q3
1.1	Ano.	Ano.	Ano.
1.2	Ano.	Ano.	Ano.
1.3	Ano.	Ano.	Ano.
1.4	Ano.	Ano.	Ano.
1.5	Ano.	Ano.	Ano.
2.1	Ano.	Ano.	Ano.
2.2	Ano.	Ano.	Ano.
2.3	Ano.	Ano.	Ano.
2.4	Ano.	Ano.	Ano.
2.5	Ano.	Ano.	Ano.
2.6	Ano.	Ano.	Ano.
3.1	Ano.	Ano.	Ano.
3.2	Ano.	Ano.	Ano.
3.3	Ano.	Ano.	Ano.

Tabulka 6.2: Výsledky kognitivního průchodu

Z tabulky 6.2 je vidět, že metoda kognitivního průchodu neodhalila žádnou nejasnost v grafickém rozhraní.

6.2.3 Testování s uživatelem

Testování s uživatelem (používám jako překlad pro usability testing[9]), jak je vidět z názvu, je skupina metod založených na testování pomocí reálných lidí z cílové skupiny uživatelů. Tyto metody většinou zahrnují natáčení videa

a audia, ideálně tak, aby se dalo vidět směry pohledů uživatele, případné pohyby jeho těla a jakékoliv manipulace uživatele s testovacím zařízením (např. pohyby myše a stisknutí kláves) pro následnou detailnější analýzu. Testování se také zúčastňuje moderátor, který je po celou dobu testu se nachází vedle uživatele a sděluje mu úkoly, případně poskytuje pomoc s jejich splněním.

Průběh testování

Testování se skládalo z dvou části: splnění účastníkem povinného seznamu úkolů z kapitoly 6.2.2 a diskuze po testu, kde účastník měl prostor podílet se o vlastní nápady či připomínky. Každý účastník před začátkem testu pověděl svoje dosavadní zaměstnání a ohodnotil svoje zkušenosti v oblasti Semantic Web pomocí následující škály:

Známka	A	B	C	D	F
Popis	Odborník	Vysoká úroveň znalosti.	Střední úroveň znalosti.	Minimální zkuše-nost.	Žádná zkuše-nost.

Tabulka 6.3: Škála hodnocení zkušenosti účastníků

Testování se zúčastnilo 5 lidí. Dále popíšu každého z účastníku a problémy, na které každý z nich narážel.

- Účastník 1
  Zaměstnání - Software engineer.
  Zkušenost v oblasti Semantic Web - C.
  Účastníkovi nebylo jasné, jakým způsobem se má editovat množina kategorií dotazů. Stejná stromečková komponenta jako na hlavní obrazovce bez textové pomůcky mu přišla matoucí.

- Účastník 2
  Zaměstnání - Software engineer.
  Zkušenost v oblasti Semantic Web - B.
  Účastník neměl žádné připomínky.

- Účastník 3
  Zaměstnání - Externí školitel na vysoké škole.
  Zkušenost v oblasti Semantic Web - A.
  Účastníkovi se podařilo nasimulovat situaci, kdy po spuštění dotazu proti nekorektnímu SPARQL-endpointu se nezobrazila žádná chybová hláška. Navíc když účastník záměrně zanesl chyby do kódu dotazu a tento dotaz spustil, nebylo z chovaní webového klientu jasné, jestli došlo k chybě při spuštění nebo se stále zpracovávají výsledky.

- Účastník 4
  Zaměstnání - Software consultant.
  Zkušenost v oblasti Semantic Web - D.
  Účastník neměl žádné připomínky.

- Účastník 5
  Zaměstnání - Software tester.
  Zkušenost v oblasti Semantic Web - C.
  Účastník neměl připomínky ke kvalitě grafického rozhraní, ale během testování odhalil několik chyb implementace.

Závěr

Všechny chyby, které byly během testování odhaleny byly také úspěšně opraveny:

- Přidaná textová pomůcka na stránce pro editace kategorizovaného dotazu.
- Přidaná chybová hláška na stránce pro spuštění dotazu.
- Opraveny logické chyby, které odhalil 5. účastník.

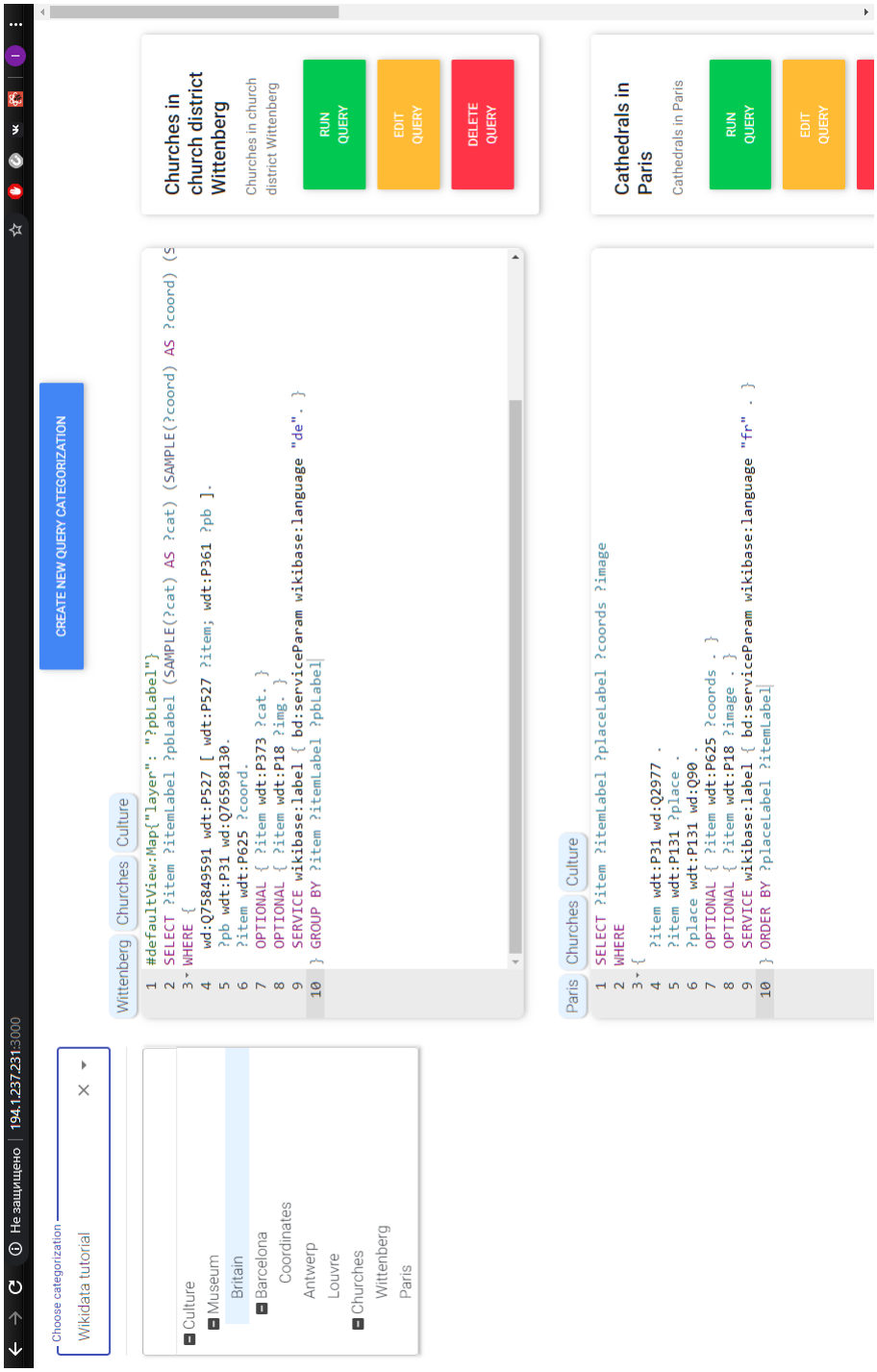
Kapitola 7

Závěr

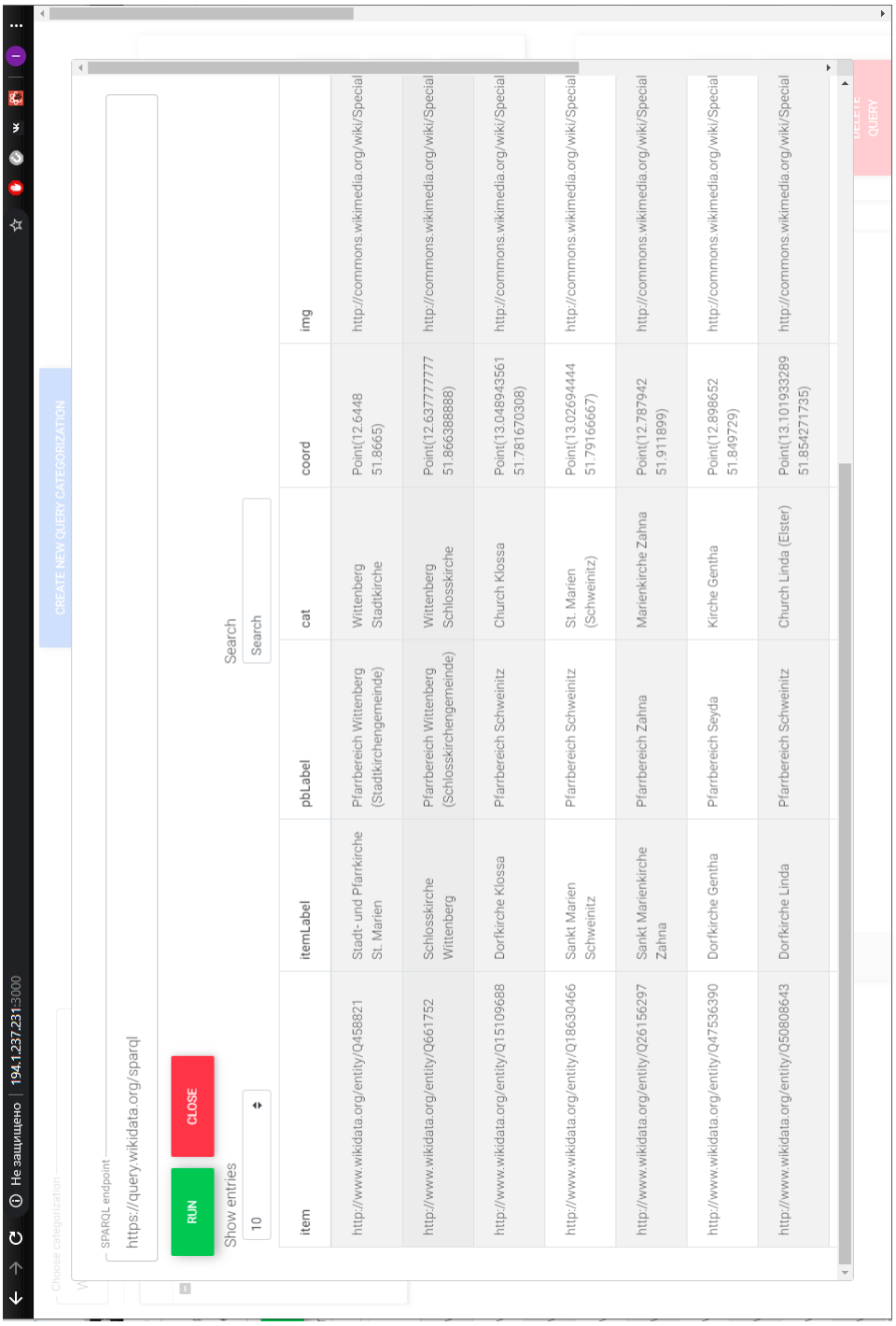
Nakonec se mi podařilo dosáhnout všech cíli této práce a implementovat webový klient pro správu katalogů SPARQL dotazů. Webový klient splňuje požadavky dané nejen cíli této práce, ale navíc splňuje požadavky, jejichž zdrojem byla rešerše existujících řešení. Validace nástroje s uživatelem odhalila drobné chyby a nejasnosti, které byly úspěšně opravny.

Příloha A

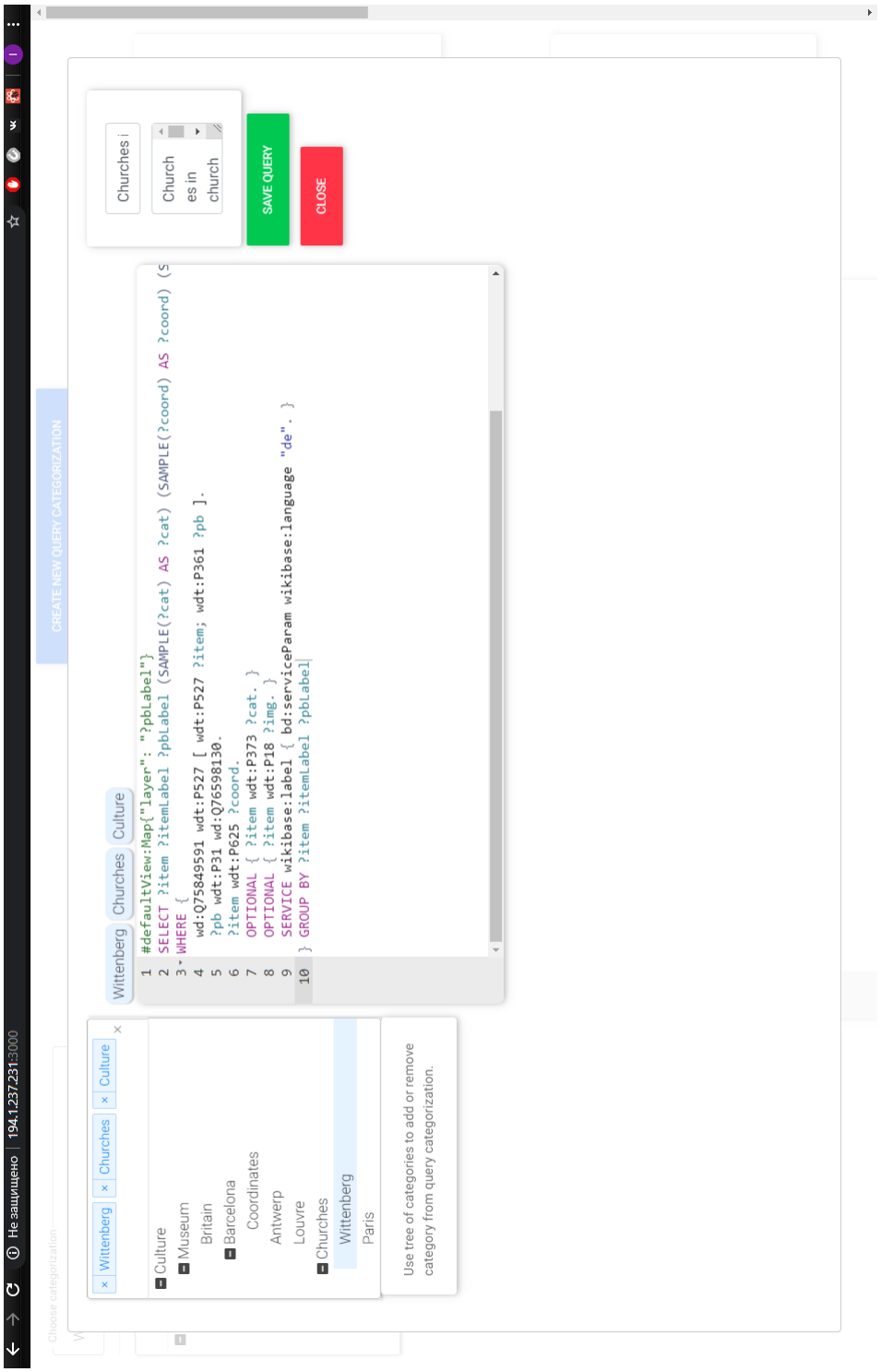
Pohledy webového klientu



Obrázek A.1: Hlavní obrazovka webového klientu



Obrázek A.2: Obrazovka spuštění webového klientu



Obrázek A.3: Obrazovka editace webového klientu

Příloha B

Příručka

B.1 Veřejně dostupná instance

Pro nahlédnutí webového klientu lze použit veřejně dostupnou instanci na adrese http://194.1.237.231:3000.

B.2 Prerekvizita

Pro použití webového klientu je potřeba mít REST API, které implementuje rozhraní z diagramu 4.4. Pro seznámení (nikoliv pro produkční použití) s webovým klientem lze použit REST API vytvořené v rámci této práce. Postup pro jeho rozchození je následující:

1. Nainstalovat databázi PostgreSQL 10.
2. Udělat checkout projektu https://github.com/Oyster-zx/sparqling-backend.
3. V souboru application.properties nastavit název databáze, jméno a heslo uživatele.
4. Spustit příkaz "mvn install".
5. Vygenerovaný .jar soubor spustit příkazem "java -jar %NAZEV%"

B.3 Návod na spuštění

Pro použití webového klientu je potřeba:

1. Udělat checkout webového klientu https://github.com/Oyster-zx/sparqling.
2. V souboru Constants.js nastavit adresu REST API.
3. Příkazem "npm start"se webový klient spustí na adrese localhost:3000.

B.4 Návod na použití

Po prvním spouštění webový klient obsahuje pouze dropdown menu se seznamem kategorizací (katalogů) SPARQL dotazů. Po zvolení kategorizace se uživatel dostane na hlavní obrazovku aplikace (viz. obrázek A.1). Zde uživatel může filtrovat kategorizované dotazy pomocí stromu kategorií. Přidáním nebo odstraněním kategorie uživatel přenačte seznam kategorizovaných dotazů.
Dále na hlavní obrazovce uživatel může:

- Vytvořit nový kategorizovaný dotaz pomocí tlačítka "Create new query categorization". Tímto tlačítkem se uživatel dostane na obrazovku pro editace dotazu (viz obrázek A.3).
- Přejít na obrazovku pro spuštění (viz obrázek A.2) dotazu pomocí tlačítka "Run query".
- Editovat kategorizovaný dotaz pomocí tlačítka "Edit query"(viz obrázek A.3).
- Smazat dotaz pomocí tlačítka "Delete query".

Na stránce editace/vytvoření kategorizovaného dotazu (viz obrázek A.3) uživatel může přidávat/mazat kategorie dotazu a měnit jeho kód, název a popis.
Při změně kódu dotazu uživatel může použit našeptáváč pomocí kombinace kláves "Ctrl+Space".
Na stránce spuštění dotazu (viz obrázek A.2) je uživatel schopen zadat SPARQL-endpoint a spustit dotaz pomocí tlačítka "Run". Po úspěšném doběhnutí dotazu uživatel si může prohlédnout tabulku s výsledky.

Příloha C

Literatura

[1] {SPARQL} 1.1 Overview. {W3C} recommendation, W3C, mar 2013.

[2] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, MarMar 2009.

[3] E. Daga, L. Panziera, and C. Pedrinaci. Basil: A cloud platform for sharing and reusing sparql queries as web apis. In *CEUR Workshop Proceedings*, volume 1486, 2015.

[4] Facebook. React framework. `https://reactjs.org/`. [Online; accessed 18-April-2019].

[5] GraphDB: an enterprise ready Semantic Graph Database, compliant with W3C Standards. Graphdb. `http://graphdb.ontotext.com/`. [Online; accessed 03-November-2019].

[6] Jest: JavaScript Testing Framework with a focus on simplicity. Jest. `https://jestjs.io//`. [Online; accessed 20-December-2019].

[7] Leipzig University, University of Mannheim. Dbpedia. `https://wiki.dbpedia.org`. [Online; accessed 15-May-2019].

[8] J. Lečbych. Intelligent tree data management component. May 2018.

[9] J. Nielsen. *Usability Engineering*. 1993.

[10] J. Nielsen. Usability inspection methods. In *Conference companion on Human factors in computing systems - CHI '94*, 1994.

[11] L. Rietveld. Yet another sparql query editor. `https://yasqe.yasgui.org/`. [Online; accessed 08-May-2019].

[12] M. Saleem. The linked sparql queries dataset. `https://aksw.github.io/LSQ/`. [Online; accessed 15-May-2019].

[13] Wikidata. Sparql query service/queries/examples. `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`. [Online; accessed 15-May-2019].

[14] D. Wood, M. Lanthaler, and R. Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.