Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Bakalářská práce

Webový server pro poskytování dynamicky generovaných objektů ve formátech RDF

Jan Řasa

Vedoucí práce: RNDr. Jakub Klímek, Ph.D.

8. května 2016

Poděkování

Díky všem

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2016 .....

Odkaz na tuto práci

Řasa, Jan. *Webový server pro poskytování dynamicky generovaných objektů ve formátech RDF*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Řada objektů na webu dat tvoří natolik velkou skupinu (až nekonečnou), že je není možné všechny perzistentně uložit a publikovat. Je nutné nabídnout server, který na základě požadavku klienta na daný objekt příslušná data dynamicky vygeneruje a odešle klientovi.

Cílem této práce je analýza již existujících řešení, návrh a implementace daného serveru včetně otestování základní funkcionality. Server bude umožňovat uživateli definovat RDF (Resource Description Framework) objekty pro dynamické generování v několika formátech. Návrh bude zohledňovat již zaběhlé technologie z oboru Linked data, jako je například dotazovací jazyk SPARQL (Protocol and RDF Query Language). Server bude implementován v jazyce Java.

Klíčová slova sémantický web, linked data, RDF, dynamické generování, SPARQL, java, webový server

Abstract

Sem doplňte ekvivalent abstraktu Vaší práce v angličtině.

Keywords Nahraďte seznamem klíčových slov v angličtině oddělených čárkou.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Při publikování dat na webu dat je vždy důležité zamyslet se nad tím, jak a kde budou tyto data uložena. Způsobů je mnoho. Mezi ty nejzákladnější patří ukládání dat do různých databázových systémů, nebo pouze přímo na určené místo na disku. Oba zmíněné způsoby a jim podobné mají ovšem jeden zásadní problém, a tím je kapacita uložiště.

Pro uložení většiny informací, se kterými se lze setkat na webu, jako jsou obrázky, zprávy, informace o počasí a mnoho dalších, se tato skutečnot nemusí příliš řešit. Kapacita uložiští nám pro tyto informace stačí. Nicméně existuje řada informací (dále také objektů), které tvoří natolik velkou skupinu (až nekonečnou), že je není možné všechny perzistentně uložit a publikovat.

Dobrým příkladem jsou například časová data - konkrétní čas či časový interval. V kontextu Linked Data [1] se často odkazuje na časový objekt. Ať už se jedná například o časy příjezdů autobusů, statistické údaje, nebo o datum nějaké události, vždy je potřeba mít daný časový objekt nějakým způsobem uložen.

Existují ale kapacity na uložení každého takového objektu? Časových objektů je přeci nekonečně mnoho. Mohou odkazovat jak do minulosti, tak do budoucnosti. A není potřeba se omezovat pouze na časové objekty. Jako další příklad může být informace o vztahu mezi lidmi, respektive mezi kterýmikoliv subjekty. Tyto informace taktéž vyžadují ohromné kapacity uložiští.

Mnoho typů objektů ale spojuje fakt, že mají vždy stejnou strukturu a jen část informací se mění (například jen sekunda, hodina, ...). A to je ideální příležitost pro to, aby se ukládání těchto objektů zaměnilo za dynamické generování. Pro vygenerování objektu stačí vždy použít stejnou strukturu a jen dosadit potřebné informace tak, aby vznikl požadovaný objekt.

Kapitola 1

Cíl práce

Cílem této práce je navrhnout, implementovat a otestovat webový server pro poskytování dynamicky generovaných objektů v RDF formátech. Server bude splňovat následující požadavky:

- Server bude umožňovat administrátorovi založit nový typ objektů včetně jejich atributů a umožní nakonfigurovat URL, pod kterými budou objekty dostupné.
- Typy objektů bude možné založit přes konfigurační soubor.
- Klient bude moci přístupem na dané URL získat data o příslušném objektu.
- Data o objektech budou klientům dostupná v RDF serializacích (RDF/XML, Turtle, N-Triples, JSON-LD) a jako webová stránka.
- Server bude využívat mechanismu Content Negotiation pro určení formátu výstupu požadavku klienta.
- Server bude implementován v jazyce Java.

Kapitola 2

Analýza

2.1 Účel aplikace

Webový server bude poskytovat uživatelům funkci pro dynamické generování objektů ve formátech RDF. Uživatel si bude moci nadefinovat vlastní typy objektů pomocí šablony včetně URL, na které budou dané objekty přístupné. Objekty budou generovány za použití informací, které bude obsahovat URL adresa při požadavku na daný objekt.

Tento způsob generování objektů ve velké míře šetří především finanční zdroje za pořizování nových uložišť. Pro velké množství stejných objektů (až nekonečně mnoho) stačí vytvořit pouze jednu definici (šablonu) objektu, která se dle specifikovaných pravidel vyplní informacemi z URL adresy a uživateli se zobrazí jako požadovaný objekt.

Další výhodou je z hlediska jednotné definice objektů i možnost lehce upravit strukturu konkrétních definic na jednom místě. Na konkrétní definici objektu může být odkazováno s různými parametry z mnoha jiných objektů a jedinou změnou v definici lze ovlivnit informace i v těchto objektech. Měnit již dříve vytvořené a uložené konkrétní objekty v takovém počtu by bylo téměř nereálné.

Účelem serveru je také chování, které co nejméně omezuje možné klienty [1]. Server využívá principu Content Negotiation [2] a podporuje nejpoužívanější typy RDF serializací, včetně možnosti zobrazit informace o objektu v HTML podobě při zobrazení prohlížečem.

2.2 Existující řešení

Pro funkce, které má tento server splňovat, neexistuje v současnosti žádné jiné řešení. Minimálně není možné dohledat žádné veřejné řešení, ani informace o nějakém soukromém řešení, které by sloužilo například pro soukromá data

[1] Zde se klientem rozumí především aplikace které by k objektům přistupovaly.

v rámci nějaké společnosti. Nicméně existuje velmi populární řešení jednoho typu objektu - časových intervalů, tzv. British Time Intervals, které je hojně používané, převážně pak také v rámci dat, které poskytuje britská vláda.

2.2.1 British Time Intervals

Pro popis těchto objektů je asi nejlepší odkázat se na konkrétní definici na webové stránce, ze které jsou tyto intervaly dostupné.[3]

> Linked data for every time interval and instant into the past and future, from years down to seconds. This is an infinite set of linked data. It includes government years and properly handles the transition to the Gregorian calendar within the UK.

Zde je vhodné všimnout si hlavně slov *infinite set*, což přesně charakterizuje typy objektů, kterými se tato práce zabývá.

2.2.1.1 Struktura URI

Pro přístup k jednotlivým generovaným objektům slouží URL adresa, která má vždy předepsanou strukturu. Informace, které jsou dostupné k tomuto datasetu, obsahují popis těchto struktur a jsou veřejně dostupné na stránkách společnosti Epimorphics [4]. Takovou URL adresou může být například *http://reference.data.gov.uk/doc/government-year/{year1}-{year2}*, kde po dosazení let míto *year1* a *year2* a přístupem na danou adresu získáme požadovaný objekt časového intervalu.

2.2.1.2 Generování objektu

Takový objekt je tedy dynamicky vygenerován s použitím parametrů z URL adresy. Konkrétní popis toho, jak jsou tyto objekty interně generovány není veřejný, ale je velmi pravděpodobné, že je použit minimálně jeden z následujících způsobů:

- Předem připravená RDF šablona (v libovolném formátu - Turtle, RDF/XML, N-TRIPLES ...) s placeholdery, do kterých se dosadí parametry z URL adresy.
- SPARQL [5][6] dotaz s placeholdery.

2.3 Požadavky

2.3.1 Funkční požadavky

- přístup k administraci objektů přes více rozhraní
- podpora více formátů pro definice a zobrazení objektů

- zobrazení existujících definic objektů
- administrace definic objektů
- konfigurace objektů bude možná několika způsoby
- vygenerování a zobrazení konkrétních objektů

2.3.1.1 Přístup k administraci objektů přes více rozhraní

Administrátori bude umožněn přístup k definicím objektů dvěma způsoby:

- přes webové rozhraní
- přes API

Oba tyto způsoby budou poskytovat stejnou funkcionalitu. Webovým rozhraním se myslí jednoduchá aplikace pro administraci objektů z webového prohlížeče. API bude sloužit k administraci z jiných potenciálních aplikací.

2.3.1.2 Podpora více formátů pro definice a zobrazení objektů

Server bude podporovat následující formáty pro definice objektů přes API:

- JSON formát
- RDF serializace TURTLE [7]

Konkrétní vygenerované objekty budou klientům dostupné v těchto RDF serializacích:

- RDF/XML [8]
- JSON-LD [9]
- N-TRIPLES [10]
- TURTLE

Každá definice objektů bude také umožňovat definovat HTML formát objektu pro zobrazení v prohlížeči.

2.3.1.3 Zobrazení existujících definic objektů

Ve webové aplikaci budou zobrazeny všechny aktuální definice objektů v tabulce s možností konkrétní definici upravit (tedy také zobrazit definici konkrétního objektu) nebo smazat přes tlačítka vedle každé definice.

Přes API bude možné získat definice objektů ve dvou formátech:

- RDF serializace

- JSON formát

Konkrétní formát bude určen principem Content Negotiation. Získat půjde seznam všech definic a také konkrétní definice.

2.3.1.4 Administrace definic objektů

Administrátorovi bude umožněno přidávat nové definice, měnit a mazat již vytvořené definice. Tyto akce budou umožněny jak z webové aplikace, tak i přes API. Dále bude moci administrátor definovat objekt konfiguračním souborem v RDF serializaci TURTLE.

2.3.1.5 Konfigurace objektů bude možná několika způsoby

Koknfigurací objektů se zde myslí možné způsoby jak a z čeho se bude generovat výsledný RDF objekt. Server bude podporovat následující způsoby:

- generování ze SPARQL šablony - CONSTRUCT dotaz [6]

  Vstupem bude SPARQL šablona CONSTRUCT dotazu s placeholdery [2], za které se dosadí při požadavku na objekt hodnoty z URL adresy a provede se příkaz který vygeneruje objekt.

- generování ze SPARQL šablony - vzdálený CONSTRUCT nebo DESCRIBE dotaz [6]

  Vstupem bude SPARQL šablona jako v prvním případě. Navíc bude možné provést DESCRIBE dotaz. Rozdíl oproti prvnímu případu je v tom, že se dotaz přepošle na zvolenou adresu SPARQL Endpointu [6], zde se provede a klientovy je pak vrácen daný objekt.

- generování z RDF šablony

  Vstupem bude RDF šablona podporovaných serializací s placeholdery, za které se dosadí při požadavku na objekt hodnoty z URL adresy.

- Proxy objekt

  Server bude umožňovat roli prostředníka při generování objektů. Požadavek na objekt se přepošle na jiný server a výsledek se přeloží klientovi dle požadované serializace. Tento způsob umožňuje generování objektů přes jiné aplikace.

2.3.1.6 Vygenerování a zobrazení konkrétních objektů

Klient bude moci přístupem na konkrétní URL adresu získat data o příslušném objektu. Typ RDF serializace nebo zobrazení HTML stránky se určí přes Content Negotiation.

[2] Placeholder: část šablony, která jasně identifikuje místo, kam se dosadí parametry z URL adresy.

2.3.2 Nefunkční požadavky

- server bude implementován v jazyce Java
- celá aplikace bude uložena ve WAR souboru pro zjednodušené nasazení

Kapitola 3

Návrh

3.1 Použité technologie

3.1.1 Resource Description Framework (RDF)

Resource Description Framework je rodina specifikací, která se používá jako metoda pro modelování informací - objektů. Jedná se o model metadat, které popisují nějaké zdroje.

Příkladem může být obyčejná webová stránka obsahující nějaké informace. Webové stránky se zaměřují především na uživatele. Důležité je, aby se uživateli stránky líbily a dbá se tedy hodně na design. Pokud jsou stránky přehledné, pak člověk nemívá problémy pochopit dané informace. Nicméně stroj (počítač, program) tyto informace sice zobrazí uživateli, ale samotné informace si interpretovat nedokáže.

RDF model popisuje tedy způsoby, jakými docílit toho, že poskytované informace budou čitelné i pro stroje.

Základní kostrou RDF modelu dat jsou takzvané *trojice*. Tyto trojice se skládají ze subjektu, predikátu a objektu. Trojice se může volně přeložit i do podoby, kde subjekt má nějakou vlastnost (predikát) s konkrétní hodnotou (objekt). Tedy všechny trojice, které mají stejný subjekt tento subjekt definují.

Každý subjekt je identifikován nejčastěji přes URI. Bavíme-li se o datech na webu, tak zde může být URI klasická URL adresa, jak ji známe z běžného používání. Co se týče objektu (hodnoty), tak se může jednat o literál (řetězec, číslo apod.), ale hodnotou může být zase URI nějakého dalšího objektu. Dokonce i predikát může být objektem s vlastním URI. Tím, že se objekt skládá z dalších objektů, které jsou jednoznačně identifikovány pomocí URI, získáváme velkou výhodu tohoto modelu. Ve výsledku vzniká graf popisující tyto trojice, což je i pro stoje čitelná struktura.

Samotné RDF popisuje pouze model. Pro uložení tohoto modelu je zapotřebí informace nějakým způsobem serializovat. Pro uložení RDF objektů se používají nejčastěji tyto RDF serializace:

- RDF/XML
- Turtle
- N-Triples
- JSON-LD

Pro ukázku je zde příklad ve formátu Turtle, který je převzat z W3C specifikace [7].

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ; # in the context of the Marvel universe
    foaf:name "Green Goblin" .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman".
```

<#green−goblin> je zde subjekt identifikovaný URI (http://example.org/#green-goblin), *rel:enemyOf* je predikát - v tomto případě zase objekt identifikovaný URI (http://www.perceive.net/schemas/relationship/enemyOf) a *<#spiderman>* je objekt - zase identifikován URI. Tento model dohromady poskytuje informaci, že objekt na dané URI je nepřítelem Spidermana. *afoaf:Person* znamená, že se jedná o objekt Person a *foaf : name"GreenGoblin"* zase to, že jeho jméno je Green Goblin. Tímto je tedy definován objekt *< #green − goblin >* a podobně tomu je u objektu *<#spiderman>*.

3.1.2 Linked Data

Linked data, jak už název napovdá, popisuje metodu propojování informací na webu dat mezi sebou. Tim Berners-Lee, zakaldatel WWW, popisuje Linked Data výstižně takto: „*Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data.*“ [11]

Aby se publikovaná data mohly využívat v co největší míře, nestačí je pouze zpřístupnit na webu dat. Největší užitek přináší linkování těchto dat dohromady. Stejně, jak je tomu u HTML dokumentů, slouží pro linkování v rámci RDF URI, které jasně identifikují objekty. Nad takto prolinkovanými

daty se poté dají najít různé vztahy, které by bez prolinkování nebylo monžé nalézt.

Linked Data staví na těchto čtyřech základních principech:

- Pro názvy a identifikaci objektů se používá URI
- Aby byly data přístupné, používají se HTTP URI
- Při přístupu na konkrétní URI lze získat informace dle standardů RDF, SPARQL, ...
- Na ostatní data se odkazuje také přes HTTP URI

Při zohlednění všech těchto principů lze získat maximálně propojené informace a jak už bylo u RDF zmíněno, ve výsledku vznikne ohromný graf vzájemně propojených informací, ve kterém je možné následně najít cenné informace i o objektech, které na první pohled nemusí mít mezi sebou nic společného.

3.1.3 Java

Při vývoji je použit programovací jazyk Java. Použití javy vychází už z požadavků ze zadání. Hlavní důvody, proč je zde java vhodná a proč byla i jedním z požadavků jsou následující:

- Rozšířenost javy ve světě Linked Data
- Snadná integrace kvalitních knihoven pro práci s RDF
- Jednoduše nasaditelná aplikace přes webový archiv (war soubor)
- Výkon

Rozšířenost javy ve světě Linked Data je opravdu velká. Důvodem, proč je tento fakt zmíněn ve výhodách, je možnost případné snadné integrace dalších systémů pro vývojáře z tohoto oboru. Důležitým důvodem je zmíněný výkon aplikace. Na ten mají vliv jak knihovny pro práci s RDF, tak ale i samotný typ aplikace. Server si může uchovávat předzpracované šablony, patterny regulárních výrazů a další informace přímo v paměti, tudíž klient ve výsledku dostane výsledek rychleji, než by tomu bylo při použití například PHP nebo podobných jazyků.

3.1.4 Apache Jena

Pro práci s RDF daty byla zvolena knihovna Apache Jena. Tato knihovna podporuje práci přímo s RDF serializacemi a díky procesoru ARQ také práci se SPARQL dotazy. Další možností pro použití byla knihovna Sesame. Knihovny si jsou velice podobné. Obě mají dobrou dokumentaci i velmi dobře zpracované

ukázky, jak lze knihovny využívat. Rozhodujícím faktorem při výběru byl výkon.

Při výkonostním srovnání byl zohledněn publikovaný test *The Berlin SPARQL Benchmark* [12]. Jena a Sesame se dle něj výkonostně velmi liší. Sesame je několikanásobně rychlejší na dotazování, ale Jena naopak s velkým náskokem vede při načítání souborů v Turtle formátu.

Výkonostně má vliv načítání definic objektů, které se uchovávají v turtle formátu. Jedná se o proces, který bude spuštěn při zapnutí aplikace a při případném novém načtení (reloadu) definic (v případě nahrání definice přímo do filesystému dle požadavku). Rychlost SPARQL dotazů tedy není v tomto případě tolik důležitá. Při požadavku klienta na vygenerování objektu už se s RDF definicí nijak nepracuje - není potřeba se nad definicemi dále dotazovat. U typů SPARQL Endpoint/Construct také nedochází k dotazovaní se nad lokálně uloženými daty.

3.1.5 Další Java knihovny

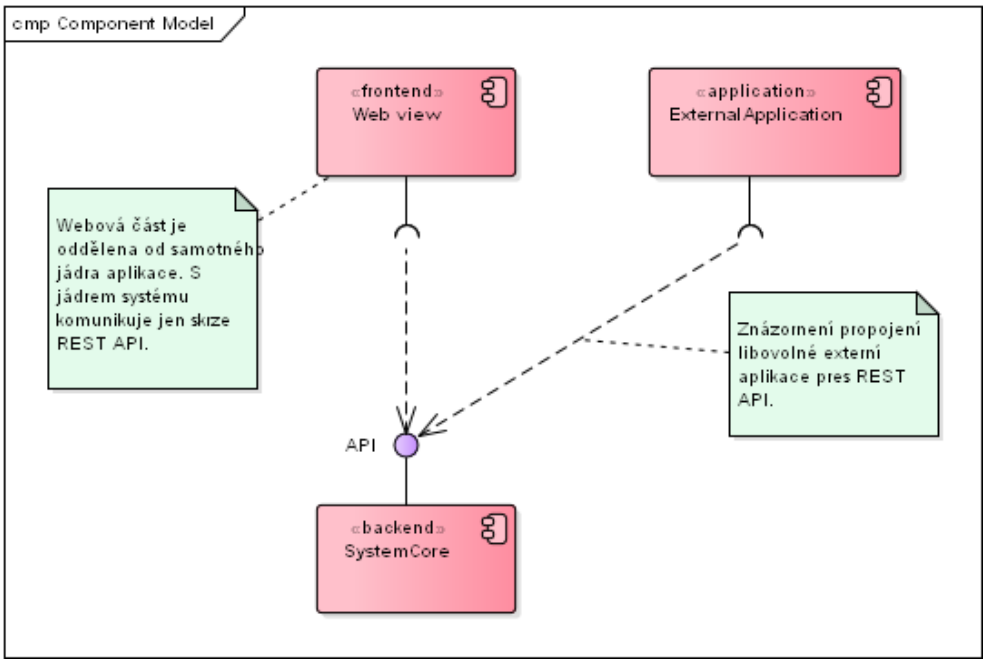
- Jersey, GSON - pro práci s definicemi objektů v rámci API
- Log4j - logovací systém
- Typesafe - konfigurace
- JUnit - testování

3.1.6 Maven

Závislosti, sestavení a testování se provádí přes build manažer Maven. Maven byl vybrán jako standard pro většinu Java aplikací. Vývoj taktéž probíhal v IDE IntelliJ IDEA, které obsahuje velmi dobrou integraci tohoto systému.

3.1.7 AngularJS

Součástí celého systému je i webová aplikace pro správu definic objektů. Zde je použit framework AngularJS, který je snadno napojitelný na REST API serveru.



Obrázek 3.1: Dvě základní části systému (jádro systému a webová, nebo jakákoliv jiná aplikace napojená na API)

3.2 Návrh architektury

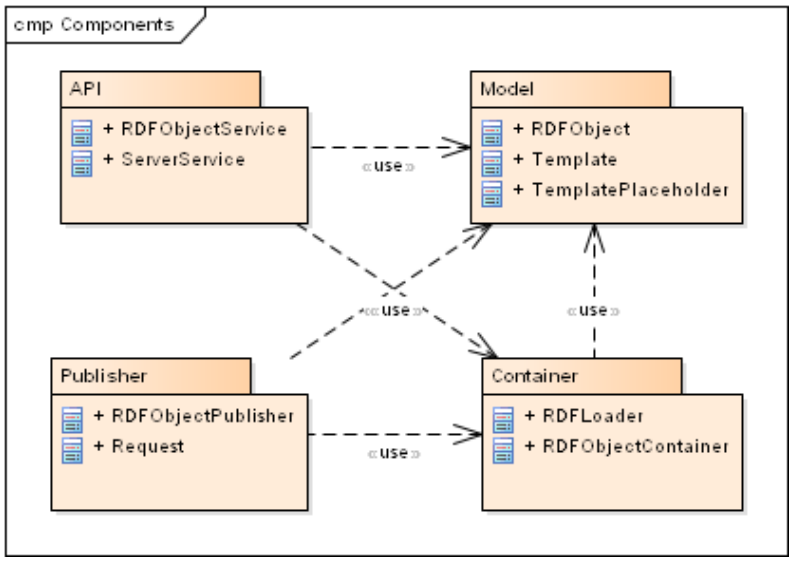
Na diagramu 3.1 lze vidět, že je systém rozdělen do 2 částí. První částí je webová aplikace, která slouží pro ulehčení administrace definic objektů. Hlavní funkcionalita tohoto serveru ale není touto částí ovlivněna, na API se lze napojit i z jiných aplikací. Tato část je napsaná ve frameworku AngularJS která využívá serverové API. U této aplikace se dá mluvit o známé MVC[3] architektuře, kde je ovšem Model (zde definice objektu) primárně součástí druhé části systému.

Druhou částí je serverová část, dále také pod názvem *Jádro systému*. Architektura této části je velice podobná MVC architektuře. O čistém MVC nelze mluvit z toho důvodu, že je zde velká provázanost komponent a není až tak striktně oddělena zodpovědnost jednotlivých částí.

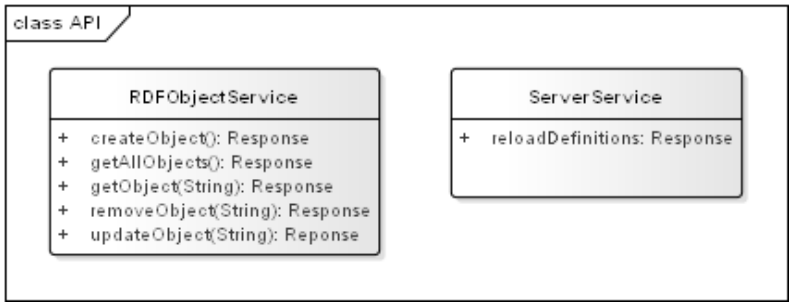
3.2.1 Jádro systému

Jádro systému se skládá ze čtyř hlavních komponent, které lze také vidět na obrázku 3.2. Pro administraci definic objektů slouží komponenta API. Další komponentou je Model, který obsahuje třídy využívané všemi komponentami. O načítání, ukládání a přístup k definicím objektů se stará komponenta Con-

[3] Model-View-Controller



Obrázek 3.2: Základní čtyři komponenty jádra systému



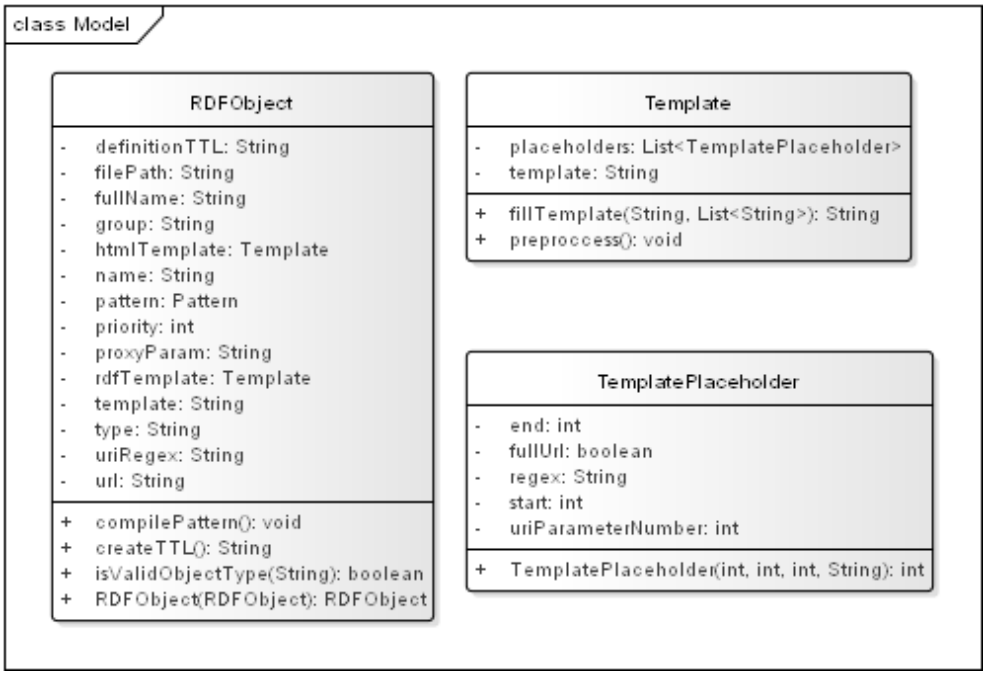
Obrázek 3.3: Třídy reprezentují služby, které mají metody vystavené pro API

tainer. Poslední základní komponentou je Publisher, který zpřístupňuje celý systém klientovi v podobě vygenerovaných objektů.

3.2.1.1 Komponenta API

Tato komponenta slouží pro administraci definic objektů. Jedná se o komponentu, ke které má přístup pouze administrátor systému. Obsahuje dvě třídy, které mají své metody vystavené pro napojení přes API.

Tyto třídy jsou zobrazeny na obrázku 3.3. Třída RDFObjectService slouží k samotnému vytváření, úpravě a mazání definic. ServerService pak obsahuje metodu, která se dá taktéž volat přes API a slouží ke znovunačtení všech definic z nastaveného uložiště.



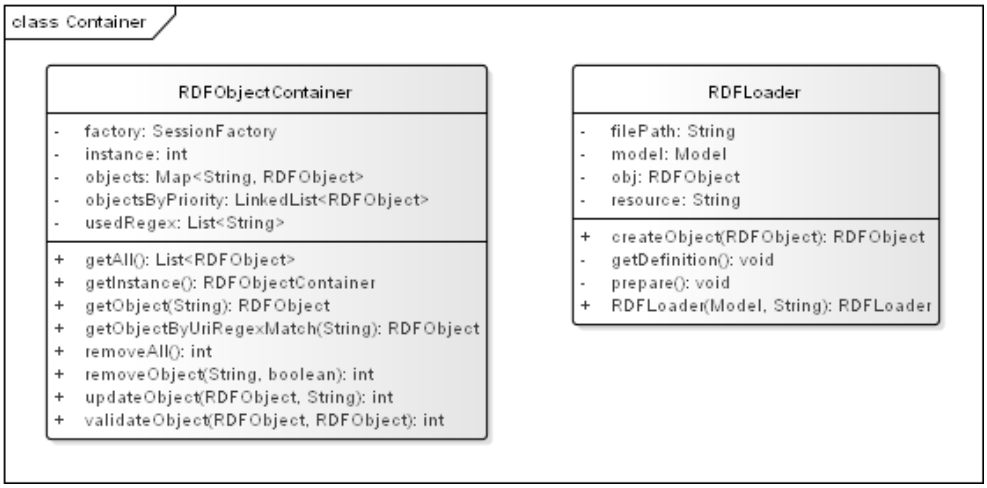
Obrázek 3.4: Třídy v komponentě Model. Pro přehled jsou uvedeny pouze nejdůležitější metody. Třídy jinak obsahují i další metody, převážně settery a gettery.

3.2.1.2 Komponenta Model

Tato komponenta obsahuje tři základní třídy, které jsou využívány zbytkem aplikace. Jedná se o třídy RDFObject, Template a TemplatePlaceholder, které jsou zobrazeny na obrázku 3.4.

RDFObject reprezentuje samotnou definici objektu. Obsahuje dva atributy typu Template. Prvním z nich je šablona definice (RDF serializace, nebo SPARQL dotaz) a druhou je HTML šablona sloužící pro zobrazení objektu v HTML formátu. Dále obsahuje metody jako jsou například validace a kompilace patternu regulárního výrazu pro pozdější identifikace definice dle požadavku klienta. Obsahuje také gettery a settery pro atributy, které ale nejsou na zmíněném diagramu vidět pro jejich primitivnost.

Třídy Template a TemplatePlaceholder zastřešují celý šablonovací systém. Třída Template obsahuje dvě důležité metody pro přdzpracování šablony a následné vyplnění šablony při požadavku. Předzpracování šablony probíhá tak, že se v šabloně naleznou všechny placeholdery a uloží se do seznamu pro pozdější vyplnění. Předzpracování probíhá pouze jendou při nahrátí definic (při startu serveru nebo znovunačtení definic). Cílem takto předzpracované šablony je urychlení generování objktů tak, aby se pouze dosazovaly hodnoty a případně aplikovaly podporované regulární výrazy.



Obrázek 3.5: Komponenta container obsahuje třídy pro administraci objektů

3.2.1.3 Komponenta Container

O správu nahraných definic se stará komponenta Container. Z obrázku 3.5 je patrné, že obsahuje 2 třídy.

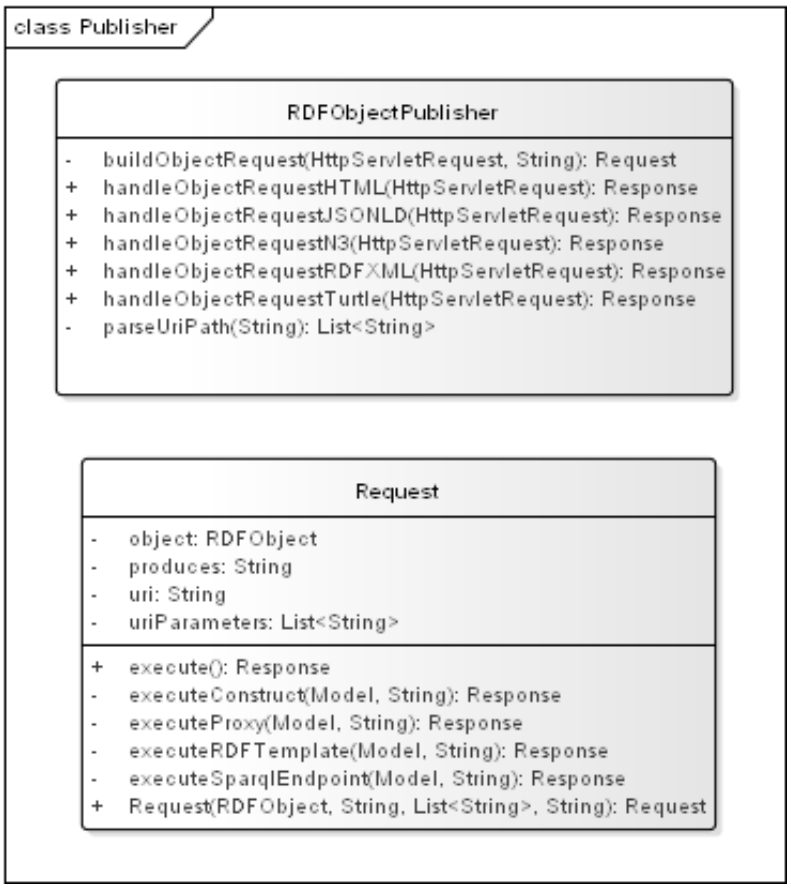
Třída RDFLoader se stará o čtení jednotlivých definic z file systému a následnou transformaci definice v Turtle formátu do objektu RDFObject, který reprezentuje konkrétní definice.

Třída RDFContainer slouží jako kontejner (malá databáze) všech nahraných objektů. Objekty si drží jak podle celého jména definice pro rychlý přístup k definicím pro API, tak i podle priorit, podle kterých dochází k vyhledávání definic při požadavku klienta. Kontejner se v aplikaci vyskytuje díky použití singleton patternu pouze jeden a je nejvytíženějším objektem v celé aplikaci, protože je využíván všemi komponentami.

3.2.1.4 Komponenta Publisher

Tato komponenta slouží k odbavování požadavků klienta na vygenerování objektu. Na obrázku 3.6 lze vidět 2 třídy, které mají za úkol interakci s klientem.

Třída RDFObjectPublisherService zpracovává prvotní požadavek klienta metodou *handleObjectRequest*(). V rámci systému je tato metoda definována pro všechny podporované serializace výstupu (HTML, RDF/XML, Turtle, JSON-LD a N-Triples). Při zavolání těchto metod je dále vytvořen objekt třídy Request, kterému je předána zodpovědnost za vygenerování výsledného objektu.



Obrázek 3.6: Komponenta container obsahuje třídy pro administraci objektů

3.3 Identifikace objektů, URL

V kontextu RDF se dají objekty identifikovat pouze jedním způsobem, a to URL adresou. V tomto případě ale URL zastává ještě jednu důležitou úlohu. Vzhledem k tomu, že je jediným spojením mezi objektem (a jeho definicí) a vnějším prostředím, tak musí nést i informace, ze kterých bude později vygenerován konkrétní objekt. Návrhu struktury URL byla proto věnována velká pozornost.

3.3.1 Struktura URL v.1

Prvotní návrh byl takový, že se URL rozdělí na následující 3 části:

- Hostname

  Touto částí se rozumí identifikace serveru a protokolu, například *https://dynrdf.com*. Z pohledu objektu slouží jen jako část identifikátoru.

- Identifikace objektu - první část cesty za hostname

  Tato část slouží k identifikaci objektu. Jedná se o unikátní název pro každý objekt. Příkladem může být například objekt časového intervalu s URL začínající *https://dynrdf.com/time-interval*, kde *time-interval* identifikuje tento objekt.

- Parametry objektu

  Zbývající část URL nese informace, které se dosadí do šablon jednotlivých definic objektů. Jednotlivé parametry jsou vždy odděleny lomítkem. Například pro objekt ročního intervalu by mohla URL vypadat následovně *https://dynrdf.com/time-interval/2015/2016*, kde by zvolené roky znamenaly parametry *od* a *do*.

Tento návrh by byl naprosto dostačujícím pro generování objektů všech podporovaných typů. Nicméně s sebou nese dva velké problémy.

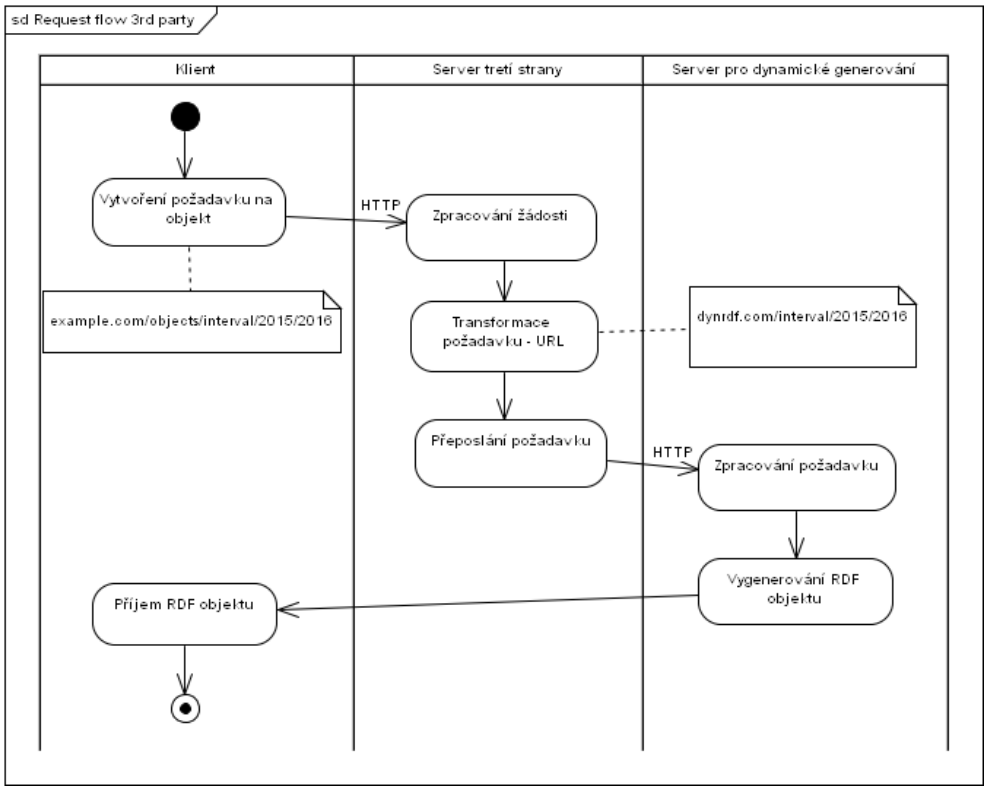
Prvním problémem je ten, že by musel být každý objekt identifikován IP adresou nebo doménou, kde tento server bude spuštěn. To by tedy znamenalo, že by tu nebyla možnost určit pro každý objekt extra URL v identifikátoru. Pokud by například funkce tohoto serveru chtěly využívat dvě různé společnosti, které by svoje objekty dále publikovali, tak by mohlo docházet k nekonzistenci odkazů na objekty. Jinými slovy se dá také zeptat na to, proč by nějaká společnost chtěla publikovat data, kde jejich identifikace není spojena například s názvem společnosti. Vždy by v identifikaci figuroval tento server, který ale s původem dat nemá nic společného. Cílem tohoto systému je pouze generovat objekty a identifikaci a vlastnictví ponechat na autorovi daných definic objektů.

Druhý problém, nebo spíše nepříjemnost nastává v případě, že by se servery společností využívající tohoto dynamického generování chovali jako prostředníci a požadavky na objekty přeposílaly na tento server. Pokud by každá definice měla vždy konkrétní unikátní identifikátor v URL adrese, pak by se pro každý takový objekt muselo přidávat pravidlo ve webových serverech na přeposílání požadavku a případně i dál složitě parsovat URL z požadavku na primárním serveru na URL, která by byla dle zmíněných specifikací.

Pro lepší představu, jak může vypadat požadavek na konkrétní objekt přes prostředníka, třetí stranu, slouží obrázek 3.7. Při požadavku přímo na tento server z klienta se proces na jiném serveru přeskočí. Ani jeden z těchto způsobů právě není bez nějakého výše zmíněného problému.

3.3.2 Struktura URL v.2

Výše zmíněný návrh by znamenal téměř nepoužitelnost tohoto systému. Bylo proto nutné navrhnout jiné řešení. To se od původního návrhu liší na dvou místech, v předání informací o objektu a identifikací objektů.



Obrázek 3.7: Požadavek na objekt přes server třetí strany. Prvotní návrh struktury URL adresy.

3.3.2.1 Informace o objektu v GET parametru

Předání dat o objektu v GET parametru naprosto jednoduše řeší problém s parsováním URL adresy na serverech třetích stran. V případě, kdy na jiný server přijde požadavek na nějaký objekt, tak stačí v konfiguraci webových serverů nastavit pouze jedno pravidlo pro přeposlání požadavku pro všechny objekty.

3.3.2.2 Identifikace objektů, regulární výraz

Předání informací o objektu v GET parametru má ale za následek, že URL adresa už neobsahuje identifikátor objektu jako první část cesty. Objekt se dá identifikovat pouze z informací v GET parametru, pro který účelově není stanovena žádná struktura.

Strukturu URL adresy v GET parametru, která má identifikovat konkrétní objekt, zná pouze autor definice. Ideálním nástrojem, jak docílit mapování URL na konkrétní objekt je zde regulární výraz, který popisuje konkrétní strukturu požadavku.

3.3.3 Shrnutí

Druhým návrhem struktury URL adresy se docílilo toho, že tento server bude schopný generovat objekty, jejichž identifikátor (URL) nebude závislý na adrese, kde tento server bude spuštěn.

Příkladem může být například adresa *http://dynrdf.com/?url=http://dataowner.com/objects/year/2015*. Původní požadavek mohl přijít na server, který je uveden v parametru *url*. Tento požadavek byl následně přesměrován na tento server pouze pomocí jednoho pravidla, kde se za parametr dosadila původní URL adresa požadavku. Následně se vygeneruje objekt roku 2015, jehož identifikátorem je obsah url parametru.

3.4 Definice objektů

Každý objekt, který má být dynamicky generován tímto systémem, musí být nadefinován administrátorem a následně uložen. Základními atributy, které daná definice musí obsahovat jsou:

- název

  Název slouží k identifikaci objektu v rámci seznamu.

- skupina

  Skupina zde označuje například název společnosti, jméno autora, nebo jiný identifikátor autora dané definice objektu. Společně s názvem tvoří plně kvalifikované jméno definice.

- URL regex

  Tento regulární výraz slouží k identifikaci konkrétní definice.

- priorita objektu

  Priorita ovlivňuje pořadí definic, v jakém se pokouší systém najít shodu regulárního výrazu s příchozí URL. Některé regulární výrazy moho popisovat více objektů. Pokud je nějaký výraz více specifičtější než jiný, tak se zvýšením priority dané definice dosáhne k požadované shodě. Priorita se určuje celým číslem. Čím menší číslo, tím větší priorita a tím dříve se bude snažit systém najít shodu s příchozí adresou právě u dané definice.

- typ definice

  Typ definice označuje způsob zadání a vygenerování výsledného objektu. Konkrétním typům dle požadavku 2.3.1.5 se tento text věnuje dále.

- šablona objektu

  Šablonou se zde rozumí text vyplněný placeholdery, do kterých se při generování doplní hodnoty. Šablonou může být SPARQL dotaz nebo kterákoliv podporovaná RDF serializace.

- HTML šablona

  Jedná se o podobnou šablonu jako pro konkrétní objekt, která je určena pro zobrazení ve webových prohlížečích.

U některých typů jsou definovány výjimky, nebo další povinné atributy. Tyto specifikace jsou zmíněny dále u popisu těchto typů.

Definice objektů přes Turtle formát musí splňovat RDF schéma, které je dostupné v přiloženém souboru *dynrdf.rdfs.ttl*.

3.4.1 RDF serializace

Definice objektu, který je definovám šablonou v RDF formátu, musí obsahovat všechny zmíněné povinné atributy. Do RDF šablony jsou doplněny při generování parametry z URL a vyplněná šablona je poté už výsledným požadovaným objektem.

3.4.2 SPARQL Construct

Šablonou pro tento typ je SPARQL construct dotaz, kde se jednotlivé parametry bindují pomocí funkce *BIND()*. Tento typ definice díky jazyku doplňuje RDF serializace o další funkcionality tohoto dotazovacího jazyka.

Pro vygenerování objektu jsou zde zapotřebí 2 kroky. Dosazení parametrů do šablony jako v případě RDF serializace a následně spuštění SPARQL dotazu lokálně, který vytvoří požadovaný objekt.

3.4.3 SPARQL Endpoint

SPARQL je navržen pro dotazování se nad datasety. Tato data jsou dostupná přes služby běžící na SPARQL protokolu. Tento typ tedy slouží ke konstrukci komplikovanějších objektů, jejichž atributy mohou být výsledkem dalších SPARQL dotazů nad konkrétním datasetem. Výsledkem tohoto typu je RDF dokument construct nebo describe dotazu.

Oproti lokálnímu construct dotazu se tento dotaz vykonává na jiném serveru. Proto je pro tento typ definice dalším povinným atributem URL adresa endpointu.

3.4.4 Proxy

Server má fungovat také jako prostředník mezi klientem a jinou aplikací poskytující RDF objekty. Jinou aplikací se rozumí webová služba, na kterou se bude požadavek přeposílávat. Tyto aplikace nemusí implementovat překlad objektů do jiných RDF serializací. Překlad do požadovaných serializací funguje zde na serveru stejně jako pro ostatní objekty.

Pro tento typ definice není potřeba definovat šablonu objektu, ale dalšími povinnými atributy jsou:

- proxy URL

  Jedná se o URL webové služby pro předaní zodpovědnosti za vygenerován objektu.

- název GET parametru

  Příchozí URL s informacemi o objektu je také přeposlána na danou službu. Názvem GET paramteru se rozumí parametr, do kterého se dosadí tato URL.

3.5 Šablonovací systém

Šablonovací systém je jedním ze stavebních kamenů této práce. Vyplněním šablony vzniká buď už konkrétní objekt, nebo SPARQL dotaz pro vygenerování objektu. Cílem návrhu tohoto systému je jednoduchost, ale zároveň dostatečná funkcionalita pro generování objektů různými způsoby.

3.5.1 Placeholder

Data o objektech jsou do šablon dosazeny přes placeholdery. Placeholderem se rozumí textový objekt, který definuje místo v dokumentu pro dosazení parametrů a má takovou strukturu, aby ho nebylo možné zaměnit s částí textu která nemá být nahrazena. V kontextu této práce se bude placeholderem rozumět textový objekt v tomto formátu:

$$[@ < d > [, < regex >]] \tag{3.1}$$

Placeholder se skládá ze dvou částí. První povinnou částí je parametr *@<d>*, který určuje konkrétní parametr URL adresy objektu, který se místo placeholderu dosadí. Jedná se tedy o *vstup* do placeholderu. Druhým nepovinným parametrem je regulrní výraz, který se může aplikovat před dosazením textu na vstup placeholderu.

3.5.1.1 Vstup placeholderu

Jak bylo zmíněno v kapitole o identifikaci objektu 3.3, každý objekt je definován URL adresou, která je serveru předána GET parametrem. Pomocí informací, které obsahuje tato adresa, je potřeba vygenerovat konkrétní objekt.

URL adresa je pro vstup do placeholderů rozdělena na části mezi lomítky. Na tyto části se dá odkazovat v placeholderu následujícím způsobem. Jako příklad budou uvedeny reference k adrese *http://intervals.com/year-interval/2013/2016*.

- @0 - reference na celou adresu (*http://intervals.com/year-interval/2013/2016*)
- @1, @2, ... (kladné čísla za znakem @) jsou reference na pozice mezi lomítky URL adresy.

  @1 = *http:*

  @2 = prázdný řetězec (mezi //)

  @3 = *intervals.com*

  @4, @5, @6 postupně *year-interval*, *2013* a *2016*

Tento způsob dává autorům šablon celkem snadný způsob, jak přímo z URL adresy dosadit do šablony požadované informace. Nicméně né všechny URL identifikátory objektů nemusí mít takovouto strukturu, aby se dalo snadno referencovat na konkrétní atributy objektu. Pokud by například v uvedeném příkladu nebyly roky rozděleny lomítkem, ale byly by v jednom parametru jako text „*2013-2016*“, nedal by se tento atribut rozdělit v RDF šabloně. Musel by se využít SPARQL construct dotaz, protože SPARQL obsahuje i funkce pro práci z řetězci (regulární výrazy). Proto je v placeholderu jako druhým, nepovinným parametrem regulární výraz.

3.5.1.2 Regulární výraz v placeholderu

Pro možnost extrahování pouze části hodnoty parametru v šabloně slouží nepovinný atribut regulárního výrazu. Tento regulární výraz podporuje Capturing groups [13]. A to způsobem, kdy je ze vstupu placeholderu extrahována hodnota, která se nachází ve skupině číslo 1.

Pokud by se autor šablony chtěl referencovat například na atribut *@5* který by obsahoval „*2013-2016*“, použil by jako placeholdery [@5, “(\d+)-”] pro *2013*, resp. [@5, “-(\d+)”] pro *2016*. Autoři šablon nebudou tedy nuceni používat SPARQL pro případy, kdy by potřebovali získat pouze část atributu, případně část z celé URL.

3.6 API

API je další důležitou komponentou v aplikaci. Slouží pro ulehčení administrace definic a zároveň otevírá možnosti pro napojení externích aplikací do systému.

Prvotním návrhem API bylo pouze několik funkcí, které by zpracovávaly požadavky v JSON formátu pro základní CRUD operace. Vzhledem ale k tomu, že celá aplikace pracuje primárně v RDF formátech, byly tyto funkce rozšířeny i o další, které umožňují administraci objektů za pomoci definic v Turtle formátu. Jedná se o totožné soubory, které administrátor může nahrát do složky s definicemi objektů, odkud se při startu nebo reloadu aplikace nahrají do systému. Výhodou tohoto kroku je i validace při požadavku na vytvoření nebo aktualizaci definice, která se u definic přímo vložených do složky provádí až při dalším spuštění nebo reloadu. Administrátor tedy může vidět chyby nebo možné konflikty ihned při pokusu o nahrátí definice přes API a nemusí hledat v logu důvody, přoč byla jeho definice, kterou nahrál přímo do složky, odmítnuta.

Celé API je rozdělené do dvou částí. Tou první je část pro zmíněnou administraci objektů, druhou pak API pro operace nad celým systémem. Nyní se počítá u systémového API pouze s funkcí pro znovunačtení definic ze složky pro ně určené. Pro tuto jedinou funkci by se API rozdělovat nemuselo, nicméně

se počítá s dalším vývojem, kdy může být zapotřebí spravovat již běžící instanci serveru. Pro tyto operace už je vhodné rozdělit API na dvě části.

3.6.1 Struktura API

Pro administraci definic se v API využívá několik základních typů HTTP požadavků. Všechny operace také podporují vstup, respektive výstup v několika formátech. Stejně jako při požadavku klienta na konrétní objekt je zde využito principu Content negotiation.

3.6.1.1 Vytvoření objektu

Vytvoření objektu je možné jak ve formátu JSON, tak i v Turtle. Při vytváření objektu dochází také k validaci definice. Výsledek operace se klient dozví přes kód odpovědi (200 - úspěch, 400 - chyba). V případě neúspěchu je v odpovědi k dispozici i popis chyby. Parametry HTTP požadavku jsou následující:

- Cesta: /api/objects
- Typ požadavku: POST
- Content type: application/json, text/turtle

3.6.1.2 Aktualizace objektu

Stejně jako při vytváření objektu je i změna objektu možná přes JSON i Turtle a provádí se také validace. Zde jsou parametry HTTP požadavku následující:

- Cesta: /api/objects/<fullname>

  <fullname> zde označuje řetězec ve formátu <skupina>/<název> (parametry definice)
- Typ požadavku: PUT
- Content type: application/json, text/turtle

3.6.1.3 Smazání objektu

Parametry požadavku pro smazání jsou:

- Cesta: /api/objects/<fullname>
- Typ požadavku: DELETE

3.6.1.4 Přístup k definicím

Klient si přes API může nechat zobrazit definici konkrétního objektu nebo celou sadu definic. Pro zobrazení si může také vybrat jakýkoliv z podporovaných formátů. Struktura požadavku je následující:

- Cesta: /api/objects/[<fullname>]

  Pokud chce uživatel získat konkrétní definici, tak za nepovinný parametr <fullname> dosadí stejně jako například u vytváření objektu celé jméno ve tvaru <skupina>/<název>. Pro zobrazení všech definic se tento parametr vynechá.

- Typ požadavku: GET

- Content type: libovolný z podporovaných (například text/turtle, application/n-triples ...)

3.6.2 RDF schéma

Každá definice objektu má povinné atributy, které musí obsahovat. Pro popis těchto atributů, respektive celé definice, slouží RDF schéma (RDFS). RDF schéma je rozšířením RDF umožňující popsat konkrétní zdroje. V kontextu této aplikace se jedná o definice objektů.

RDFS této aplikace popisuje definice jako objekty tříd konkrétních typů definic (RDF serializace, Sparql Construct ...). Dále definuje atributy, jako jsou například *dynrdf:priority* nebo *dynrdf:regex*. Pokud autor neví, k čemu například tyto atributy slouží, může se podívat do schématu a přečíst si například komentář ke konkrétnímu atributu. Schéma k definicím je možné zobrazit si v příloze pod názvem *dynrdf.rdfs.ttl*.

Kapitola 4

Implementace

Implementace probíhala už také zčásti při analýze. Bylo potřeba připravit si celý projekt, převážně pak vyzkoušet a osvojit si práci s knihovnami pro RDF (Apache Jena) a tvorbu webových služeb v Javě. Jako vývojové prostředí bylo po předchozích zkušenostech zvoneno IntelliJ IDEA od společnosti JetBrains. Při implementaci byl zohledněn následující vývojový plán:

- Vytvoření prototypu webové služby a deploy na Glassfish [14]
- Tvorba modelu definic a persistence
- API a webové rozhraní
- Vývoj šablonovacího systému
- Zpracování požadavku klienta a vygenerování výsledného objektu

4.1 Prototyp webové služby

Cílem vývoje prototypu webové služby bylo získat přístup a napojení do celé aplikace zvenčí. Ještě se nejednalo o napojení na konkrétní funkce serveru. Tento bod byl také důležitým pro otestování základu aplikace přímo na aplikačním serveru (Glassfish).

Prvnímu úspěšnému zobrazení „Hello world!“ předcházelo mnoho práce s laděním závislostí na jednotlivých balíčcích a nastavení samotného Glassfish serveru. I přesto, že se jednalo v základu pouze o napojení na jednu metodu, byl tento prototyp důležitý pro další vývoj. Při implementaci dalších funkcí stačilo už jen využít strukturu prototypu a jen trochu ji pozměnit dle toho, co daná funkcionalita měla dělat (nastavení vstupů, content negotiation, ...).

4.2 Model a persistence

Model definic a jejich persistence šly při implementaci vždy vedle sebe. Konkrétně se jedná o třídy *RDFObject*, *RDFObjectContainer* a *RDFLoader*. Podobně jako například při návrhu struktury URL adresy také zde došlo k jedné velké změně. I vzhledem k tomu, že se jednalo o změnu návrhu až při implementaci, nebyl dopad příliš velký. Týkalo se to pouze tříd pro model definice a kontejneru, kde byla implementační závislost mezi sebou minimální.

První návrh spočíval v tom, že se definice objektů budou ukládat do databáze. Pro tyto účely měla sloužit pro jednoduchou nasaditelnost SQLite databáze a pro komunikaci s ní pak framework Hibernate. Tento systém ukládání definic nicméně přinesl do celé aplikace zbytečnou složitost. Vzhledem k tomu, že jedním z požadavků byla i konfigurace objektů přes konfigurační soubor, tak by zde musely být 2 systémy pro správu definic, u kterých by při dalším vývoji mohlo docházet k nekonzistenci.

Nejlepším řešením bylo přepracovat celkovou správu definic tak, aby byla přehledná a co nejjednodušší pro další vývoj. Vzhledem k požadavku na konfigurační soubor se ukládání definic sjednotilo do definic v turtle formátu, které jsou ukládány jako jednotlivé soubory a napojení na SQLite databázi bylo ze systému úplně odstraněno.

4.2.1 RDFObject

RDFObject je primitivní třída, která představuje definice objektů. Kromě atributů a jejich setterů a getterů obsahuje metodu pro vygenerování definice v turtle formátu. Tato metoda je využívána v případech, kdy klient posílá na server požadavek obsahující informace o definici v JSON formátu. Definice je poté reprezentována v turtle formátu a následně se na ní aplikují operace jako jsou například validace stejně jako na definice, které jsou už přímo v turtle formátu.

Třídy RDFObject se změna ukládání definic dotkla nejméně. Z projektu byl odstraněn framework Hibernate a s ním také anotace v této třídě, které mapovaly definice a jejich atributy do databázové tabulky.

4.2.2 RDFObjectContainer

Tato třída byla implementována od začátku jako jedináček (Singleton). Vzhledem k tomu, že se jedná o databázi zpracovaných definic, tak zde dává smysl mít pouze jednu instanci této třídy v celém systému. Bylo by zbytečně paměťově náročné, aby systém obsahoval více instancí.

Při implementaci této třídy byly ve velkém množství používány java kolekce, jako jsou mapy, listy a jejich implementace HashMap, ArrayList a LinkedList. Byla také brána v potaz efektivita algoritmů pracující nad těmito kolekcemi. Jedná se především o vyhledávání správné definice objektu při po-

žadavku klienta dle regulárních výrazů. Například ty se v systému kompilují pouze při nahrátí do kontejneru.

Větší optimalizací vyhledávání pak bylo zavedení atributu „group“ pro definice. Aby se při vyhledávání v seznamu definic nemuselo v nejhorším případě procházet celým seznamem a u každé definice zjišťovat, zda regulární výraz neodpovídá vstupní URL, může URL adresa obsahovat i název skupiny definic (například se dají definice dělit dle názvu organizací) a ve výsledku může docházet k procházení pouze několika pár definicí.

Při změně celkového systému pro ukládání definic byly v této třídě odstraněny metody, které měly na startost vkládání, mazání a aktualizaci definic v databázi. Tato zodpovědnost byla předána nové třídě RDFLoader.

4.2.3 RDFLoader

Tato třída byla zavedena do systému po změně systému ukládání definic. Pro nalezení všech uložených definic je zde implementována rekurzivní funkce, díky které je možné ukládat definice objektů do podsložek a tím i případně získat přehled uložení definic, pokud by administrátor preferoval nahrávání definic ručně, což je také možné.

Tato třída také nejvíce ze všech pracuje s knihovnou Apache Jena. Používá ji pro načítání modelů z definic v turtle formátu a následně se nad těmito modely dotazuje jazykem SPARQL na konkrétní atributy pro namapování definic na objekty třídy RDFObject. K mapování zde dochází proto, že by bylo velmi neefektivní dotazovat se neustále na atributy definic přes SPARQL.

4.3 Implementace API

Při implementaci API byly ve velkém množství využity především dostupné anotace z balíčku *javax.ws.rs*. Díky těmto anotacím je výsledný kód velice čistý, protože nebylo zapotřebí vlastní logiky pro zpracování požadavků dle typu HTTP metod a podobně. Třída RDFObjectService sice obsahuje kvůli tomu více metod, které se liší například pouze výstupní serializací (použití anotace „@Produces“), ale pro další vývoj je tímto umožněno snadno zakázat nějakou funkci, případně přidat možnosti pro výstup v jiných formátech.

Výstupní serializace objektů je zde implementována převážně funkcemi knihovny Jena, kterými se turtle definice konvertuje na jiné serializace. Nedochází zde k načítání definic ze souborů, ale každý RDFObject si drží atribut $definitionTTL$, který byl zaveden právě pro rychlejší výstupní serializaci do RDF formátů.

Podporovaným výstupem je také JSON formát, pro který je využita Gson knihovna, která automaticky vygeneruje z atributů objektu řetězec v JSON formátu. Pro toto generování se také využívá ve třídě RDFObject u některých atributů parametr *transient*, aby nedocházelo k serializaci zbytečných para-

metrů, které slouží pouze pro interní účely aplikace a neměly by být přístupny klientům.

4.4 Požadavek klienta na vygenerování objektu

Požadavky klientů už na konkrétní vygenerované objekty zastřešují dvě třídy. Třída RDFObjectPublisherService se chová velmi podobně jako API třída RDFObjectService. Zpracování požadavků je směrováno na metody pomocí anotací, ve kterých se volá metoda *buildObjectRequest*, která se kontejneru dotazuje na konkrétní definici a následně vytvoří objekt třídy Request, na který je předána zodpovědnost za vygenerování.

Zpracování požadavku probíhá v několika krocích:

- Nalezení definice dle URL adresy a vytvoření objektu Request
- Zavolání metody *execute∗* (executeProxy, executeRDFTemplate, ...)
- Vyplnění šablony parametry z URL adresy
- Spuštění procesů dle typu definice (SPARQL dotaz, proxy dotaz, ...)
- Zobrazení výsledného vygenerovaného objektu

4.4.1 Nalezení odpovídající definice

Požadavek na každou definici je identifikovatelný přes regulární výraz. Problémem regulárních výrazů zde je to, že několik regulárních výrazů může popisovat stejné řetězce. Příkladem může být třeba URL „http://dynrdf.com/object/42“. Pokud by jedna definice používala regulární výraz pouze „obj“ a druhá „object“, mohlo by dojít ke konfliktu, protože by danému řetězci odpovídaly obě definice. Komplikovaněji, pokud by „obj“ mělo odpovídat jiné očekávané URL „http://company.com/rdf/obj/99“ a první z adres by se porovnávala k tomuto zkrácenému řetězci, došlo by k vygenerování špatného objektu.

Proto bylo už v návrhu počítáno se systémem priorit. Při hledání odpovídající definice se porovnávají nejdříve definice s největší prioritou, která odpovídá číslu jedna. Nastavením větší priority pro definici s výrazem „object“ dojde ve zmíněném příkladu u URL „http://dynrdf.com/object/42“ k dřívějšímu porovnání a tedy i k vygenerování správného objektu.

Prioritní systém je implementován přes spojový seznam. Ke každé skupině existuje daný seznam definic dle priorit. Navíc je v systému i seznam celkově všech definic dle priorit, ve kterém se vyhledává, pokud URL adresa požadavku neobsahuje parametr group. Po nalezení definice se volá jedna z metod *execute∗* a proces pokračuje vyplněním šablony.

4.4.2 Vyplnění šablony a šablonovací systém

Šablonovací systém je implementován s použitím regulárních výrazů. Už při registraci definice do kontejneru dochází k předzpracování šablony tak, že se naleznou všechny placeholdery a uloží se do listu. Každý nalezený placeholder je reprezentován objektem třídy TemplatePlaceholder. Tento objekt obsahuje informace o pozici placeholderu v šabloně, číslo parametru, který se má z URL dosazovat a případně i předkompilovaný regulární výraz pro aplikaci na parametr.

Všechny placeholdery jsou nalezeny přes regulární výraz „\[\s*@(\d+)\s*(,\s*[\"|\"](.*)[\"|\"])?\s*\] “. Tento výraz popisuje strukturu placeholderu jak s regulárním výrazem, tak i bez něho. Dále je také umožněno používání lomítek pro escapování uvozovek, ve kterých se regulární výraz vyskytuje. To bylo zavedeno hlavně kvůli problémům při parsování definic typu JSON-LD. Přes capturing groups jsou následně extrahovány číslo parametru a regulární výraz.

Samotné vyplnění šablony je poté velice jednoduché. Při iteraci nad listem placeholerů se do výsledného výstupu vyplňují nejdříve data, které se vyskytují před pozicí aktuálního placeholderu. Poté se vyplní placeholder a v případě poslední položky v listu i data, která se nacházejí za posledním placeholderem.

4.4.3 Generování objektu

U definic typu RDF serializace je vyplnění šablony téměř vše, co je potřeba udělat. Pro tento typ je poslední akcí transformace na požadovanou RDF serializaci, o což se stará zase knihovna Apache Jena.

Pro definice typu Sparql construct se po vyplnění šablony spustí navíc tento dotaz, jehož výsledkem je RDF objekt, který je poté taktéž serializován na formát dle požadavku klienta.

U typů Sparql endpoint a proxy dochází ke komunikaci se servery třetích stran. Komunikaci se servery nebylo potřeba složitě implementovat. Apache Jena podporuje jak čtení z URL adresy, tak i spuštění SPARQL dotazu na endpointu. Pro spuštění SPARQL dotazu na endpointu se narozdíl od vnitřku serveru používá jako výstupní formát endpointu JSON-LD. Při testování totiž došlo k problémům kódování v jiných formátech, převážně v turtle formátu. Například endpoint serveru Dbpedia (http://dbpedia.org/sparql) při většině dotazů vracel v turtle formátu znaky, pro které knihovna Jena vyhazovala výjimky. Kódovaní ve formátu JSON-LD bylo ale v pořádku ve všech nalezených případech a proto byl zvolen tento formát pro komunikaci se serverem třetí strany. Výsledný formát je nicméně stále určen požadavkem klienta.

4.4.4 Webová aplikace pro administraci definic

Součástí aplikace je také jednoduchá webová aplikace pro administraci definic. Jedná se o aplikaci psanou ve frameworku AngularJS. Nastavení komunikace

Angularu a REST API na serveru bylo triviální. Potřeba bylo pouze nastavit URL adresu API a HTTP metodu pro aktualizaci definic (HTTP PUT).

Aplikace obsahuje pouze dva kontrolery. RDFObjectEntityController se stará o tvorbu a aktualizaci definic a OverviewController o zobrazení již vytvořených definic klientovi.

Kapitola 5

Testování

Pro téměř každou netriviální metodu, které jsou klíčové pro běh systému, byly vytvořeny testy. Jedná se jak o klasické unit testy, které testují pouze chování konkrétních metod, tak i o integrační testy, které testují chování aplikace jako celku. Pro tento typ testování byl použit framework JUnit a Jersey Test.

Aplikace byla také testována na výkon pomocí aplikace JMeter. Více o testování výkonu je uvedeno dále.

5.1 Základní testy

Několik tříd v této aplikaci obsahuje snadno testovatelné metody. Jedná se hlavně o třídu Template, kde je testováno správné dosazování parametrů do šablony. Také třída RDFContainer obsahuje několik podobných testů na validaci objektů, které se do kontejneru ukládají.

RDFContainer obsahuje kromě lehčích testů na validaci objektů také složitější testy metod, které komunikují převážně s knihovou Apache Jena. Jedná se o testy vytváření, aktualizace a mazání objektů. Ačkoliv je implementace těchto metod komplikovanější, tak testování probíhá lehce přes dotazy na objekt, který je testován. Pokud kontejner na dotaz odpoví neočekávaně (například vytvořený objekt neexistuje, aktualizovaný objekt obsahuje nesprávné atributy, ...), je tato chyba odchycena a test je neúspěšný.

Další vyšší úrovní testování jsou testy třídy Request. Tato třída obsahuje metody, které ze vstupních parametrů generují výsledný objekt. Jedná se o metody pro všechny podporované typy definic - *executeRDFTemplate*(), *executeProxy*() a další. Pro tyto požadavky jsou předpřipraveny definice určené pro tyto testy. Před samotným testováním se inicializuje kontejner s těmito definicemi. Nad těmito definovanými objekty jsou poté spouštěny metody pro generování a výsledné objekty jsou poté porovnány vůči vstupním parametrům.

5.2 Testování požadavků klienta a API

Nejvyšší úrovní testování jsou testy samotných HTTP požadavků. Pro tento typ testování je použit framework Jersey Test a celá aplikace je spuštěna v kontejneru pomocí knihovny Jersey Container Grizzly2.

Testování API probíhá pro všechny typy požadavků, včetně všech podporovaných serializací. Stejně je tomu i u požadavků klienta na vygenerování objektu. Ty jsou testovány na všechny testovací objekty přes všechny podporované serializace.

V těchto testech bylo potřeba se vypořádat s nekompatibilitou některých knihoven. Jednalo se hlavně o injektování objektu *HttpServletRequest* do parametrů metod přes anotaci @Context. Provider Jersey frameworku Grizzly musel být zaměněn za InMemory pro úspěšné spuštění testů.

5.3 Testování výkonu

Součástí testů je i základní test výkonu aplikací JMeter. Aplikace byla pro tyto účely nasazena do webového kontejneru Glassfish 4.1.1 na VPS s těmito parametry:

- Kontejnerová virtualizace OpenVZ
- CPU Intel(R) Xeon(R) E5-2630 @2.30GHz (8 virt. jader)
- 4GB RAM
- Ubuntu server 14.04
- Konektivita 300 Mbps

Testování probíhalo ze zařízení s těmito parametry:

- CPU Intel(R) Core(TM) i5-2410M @2.30GHz
- 8GB RAM
- OS Ubuntu 12.04

Sledovaným parametrem byla převážně propustnost (throughput). U ostatních parametrů, jako jsou například průměrná doba odpovědi, mohlo docházet ke zkreslení testu. Testovací případy byly spouštěny na osobním PC s cílem zjistit pouze přibližný počet dotazů, které je aplikace shopna odbavit. Tento počet je nicméně omezen výkonem tohoto PC a také připojením k internetu. Předpokládá se, že aplikace je na dané konfiguraci výkonější, než ukazují tabulky 5.1 a 5.2.

Tyto tabulky ukazují výsledky testů, u kterých je znázorněna také výhoda používání parametru *group* v URL adrese požadavku. V prvním případě

Tabulka 5.1: Test výkonu s parametrem group. Celková propustnost při tomto testu byla 147 požadavků za sekundu. Sloupce average, median, min a max označují trvání požadavku v ms po připojení.

Label	Samples	Average	Median	Min	Max	Throughput
Turtle	300	372	22	10	5138	29.8
RDF/XML	300	62	19	9	991	56.3
NTriples	300	553	24	9	5132	31.6
JSON-LD	300	1005	87	9	5141	33.8
SPARQL Construct	99	106	36	13	950	32.3
SPARQL Endpoint	99	1132	315	89	5190	13.5
Proxy	99	2122	765	325	5212	13.1
TOTAL	1497	621	29	9	5212	147

Tabulka 5.2: Test výkonu bez parametru group. Celková propustnost při tomto testu byla přibližně 91 požadavků za sekundu. Sloupce average, median, min a max označují trvání požadavku v ms po připojení.

Label	Samples	Average	Median	Min	Max	Throughput
Turtle	300	2348	1549	98	5483	19
RDF/XML	300	1855	922	134	5419	21.2
NTriples	300	3070	3539	232	5527	19.4
JSON-LD	300	3878	4063	970	5503	20.4
SPARQL Construct	99	3807	3979	1829	5467	8.4
SPARQL Endpoint	99	3975	3936	3490	4518	8
Proxy	999	4170	4306	3244	5159	7.5
TOTAL	1497	3025	3596	98	5527	91.4

parametr group je uveden a tudíž není potřeba v nejhorším případě prohledávat celý seznam objektů. Pro testovací účely bylo do kontejneru nahráno 300 definic s rozdílnými prioritami a implementován testovací parametr do URL adresy, který definoval prioritu testované definice. Tímto byly nasimulovány požadavky na objekty s rozdílnou prioritou. V aplikaci JMeter toho bylo dosaženo implementací jednoduchého scriptu pro náhodné generování díky komponentě BeanShell.

Závěr

Literatura

[1] Linked Data community, Tom Heath: *Linked Data - Connect Distributed Data across the Web [online]*. [cit. 2016-05-08]. Dostupné z: `http://linkeddata.org/`

[2] W3C: *HTTP/1.1: Content Negotiation [online]*. [cit. 2016-04-16]. Dostupné z: `https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html`

[3] Datahub: *data.gov.uk Time Intervals [online]*. [cit. 2016-04-16]. Dostupné z: `https://datahub.io/tr/dataset/data-gov-uk-time-intervals`

[4] Epimorphics: *Using Interval Set URIs in Statistical Data [online]*. [cit. 2016-04-16]. Dostupné z: `http://www.epimorphics.com/web/wiki/using-interval-set-uris-statistical-data`

[5] W3C: *SPARQL Query Language for RDF [online]*. [cit. 2016-04-16]. Dostupné z: `https://www.w3.org/TR/rdf-sparql-query/`

[6] DuCharme, B.: *Learning SPARQL-Querying and Updating with SPARQL 1.1*. O'Reilly Media, 2011.

[7] W3C: *RDF 1.1 Turtle [online]*. [cit. 2016-04-19]. Dostupné z: `http://www.w3.org/TR/2014/REC-turtle-20140225/`

[8] W3C: *RDF 1.1 XML Syntax [online]*. [cit. 2016-04-25]. Dostupné z: `https://www.w3.org/TR/rdf-syntax-grammar/`

[9] W3C: *JSON-LD 1.0 [online]*. [cit. 2016-04-25]. Dostupné z: `https://www.w3.org/TR/json-ld/`

[10] W3C: *RDF 1.1 N-Triples [online]*. [cit. 2016-04-25]. Dostupné z: `https://www.w3.org/TR/n-triples/`

[11] Tim Berners-Lee: *Linked Data - Design Issues [online]*. [cit. 2016-04-21]. Dostupné z: `https://www.w3.org/DesignIssues/LinkedData`

[12] Christian Bizer, Andreas Schultz: *The Berlin SPARQL Benchmark [online]*. [cit. 2016-04-22]. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.8030&rep=rep1&type=pdf`

[13] Jan Goyvaerts: *Regular Expression Reference: Capturing Groups and Backreferences [online]*. [cit. 2016-04-17]. Dostupné z: `http://www.regular-expressions.info/refcapture.html`

[14] Oracle Corporation: *GlassFish Server [online]*. [cit. 2016-04-25]. Dostupné z: `https://glassfish.java.net/`

Příloha **A**

Seznam použitých zkratek

RDF Resource description framework

URI Uniform Resource Identifier

URL Uniform Resource Locator

HTTP Hypertext Transfer Protocol

WWW World Wide Web

HTML HyperText Markup Language

SPARQL SPARQL Protocol and RDF Query Language

API Application program interface

CRUD Create, read, update and delete

IP Internet Protocol

MVC Model–view–controller

JSON JavaScript Object Notation

Příloha **B**

Obsah přiloženého CD

s

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS