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.

15. května 2016

# Poděkování

Chtěl bych poděkovat vedoucímu mé práce RNDr. Jakubu Klímkovi za odborné vedení, pomoc při zpracování této práce a cenné rady. Dále bych chtěl také poděkovat svým rodičům a přátelům za podporu nejen při tvorbě této práce, ale po celou dobu studia.

# 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 15. 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.

Tato práce se zabývá analýzou již existujících řešení, návrhem a implementací daného serveru včetně otestování základní funkcionality. Server bude umožňovat uživateli definovat RDF (Resource Description Framework) entities 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, web dat, Linked Data, RDF, dynamické generování, SPARQL, Java, webový server

# Abstract

There are many objects on the web of data that make such a big group (almost infinite) that it's not possible to persist them all and publish them. It's necessary to offer a web server that to the client's requests to the given objects dynamically generates the data and sends them back to the client.

The goal of this bachelor's thesis is an analysis of existing solutions, to design and implement this web server, including tests of the basic functionality. Users of the web server will be able to define RDF (Resource Description Framework) entities for dynamically generating in the several formats. The design will include popular technologies from Linked Data such as SPARQL language (Protocol and RDF Query Language). The server will be implemented in Java.

**Keywords** semantic web, web of data, Linked Data, RDF, dynamically generating, SPARQL, Java, web server

# Obsah

**Úvod** **1**

**1 Cíl práce** **3**

**2 Analýza** **5**
2.1 Účel aplikace . . . . . 5
2.2 Existující řešení . . . . . 6
2.3 Požadavky . . . . . 7

**3 Návrh** **11**
3.1 Použité technologie . . . . . 11
3.2 Návrh architektury . . . . . 16
3.3 Identifikace objektů, URL . . . . . 20
3.4 Definice objektů . . . . . 24
3.5 Šablonovací systém . . . . . 26
3.6 API . . . . . 27

**4 Implementace** **31**
4.1 Komponenta Model . . . . . 31
4.2 Komponenta Container . . . . . 34
4.3 Komponenta API . . . . . 38
4.4 Komponenta Publisher . . . . . 40
4.5 Webová aplikace pro administraci definic . . . . . 41

**5 Testování** **43**
5.1 Unit testy . . . . . 43
5.2 Integrační testy . . . . . 44
5.3 Testování výkonu . . . . . 47

**6 Uživatelská dokumentace** **51**

6.1 Webová aplikace . . . . . 52
6.2 API . . . . . 55
6.3 Typy definic . . . . . 55
6.4 Struktura šablony . . . . . 56
6.5 Vygenerování RDF entity na žádost klienta . . . . . 57
6.6 Instalace serveru . . . . . 60
6.7 Nastavení serveru pro původ dat . . . . . 63

**Závěr** **65**
Využití aplikace a další vývoj . . . . . 65

**Literatura** **67**

**A Seznam použitých zkratek** **71**

**B Obsah přiloženého CD** **73**

# Seznam obrázků

3.1 Základní části systému . . . . . 16
3.2 Model komponent . . . . . 17
3.3 Model tříd API . . . . . 17
3.4 Model tříd komponenty model . . . . . 18
3.5 Model tříd komponenty container . . . . . 19
3.6 Model tříd komponenty publisher . . . . . 20
3.7 Požadavek na objekt přes server třetí strany . . . . . 22

6.1 Přehled definic ve webové aplikaci - hlavní stránka . . . . . 52
6.2 Formulář pro založení a editaci definice . . . . . 54
6.3 Vygenerovaná HTML stránka pro příklad loggerTurtle . . . . . 60
6.4 Nastavení parametrů v konfiguračním souboru . . . . . 62
6.5 Nahrání webového archivu na server Tomcat . . . . . 62

# Seznam tabulek

5.1 Test výkonu - s parametrem group . . . . . 49
5.2 Test výkonu - bez parametru group . . . . . 49

# 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 většinou 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, časy uvedené ve statistických údajích, 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

Tato kapitola se zabývá analýzou vyvíjené aplikace. Zmíněn je účel aplikace, analýza již existujících řešení a analýza požadavků kladených na tuto aplikaci. Cílem analýzy je stanovit základní funkce aplikace a zjistit možné konkurující aplikace, analyzovat je a navrhnout vylepšení, které bude později implementováno.

## 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.

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

## 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 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 URL

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

Tato kapitola obsahuje výčet funkčních a nefunkčních požadavků, které budou na výslednou aplikaci kladeny.

### 2.3.1 Funkční požadavky

Funkční požadavky jsou:

- 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

Konfigurací objektů se zde myslí možné způsoby jak a z čeho se bude generovat výsledný 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.

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

### 2.3.2 Nefunkční požadavky

Nefunkční požadavky jsou:

- server bude implementován v jazyce Java
- celá aplikace bude uložena ve WAR souboru pro zjednodušené nasazení
- aplikace bude spustitelná v Javě verze 8
- k aplikaci budou přiloženy ukázkové definice objektů

Kapitola 3

# Návrh

Tato kapitola se zabývá návrhem výsledné aplikace. První část je věnována použitým technologiím. Poté následuje návrh celkové architektury. Převážná část této kapitoly je věnována návrhu konkrétních komponent, jako je například API a administrace definic objektů.

## 3.1 Použité technologie

Cílem této kapitoly je vymezit technologie, které se v této práci budou vyskytovat. Vzhledem k méně známým technologiím, jako jsou Linked Data, jsou tyto technologie i krátce popsány.

### 3.1.1 Resource Description Framework (RDF)

Resource Description Framework je rodina specifikací, která se používá jako metoda pro modelování informací - RDF entit. 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 IRI. Bavíme-li se o datech na webu, tak zde může být IRI 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 IRI nějaké další entity. Dokonce i predikát může být RDF entitou s vlastním IRI. Tím, že se objekt skládá z dalších entit, které jsou jednoznačně identifikovány pomocí IRI, získáváme velkou výhodu tohoto modelu. Ve výsledku vzniká grafová struktura popisující tyto trojice, což je i pro stroje čitelná struktura.

Samotné RDF popisuje pouze model. Pro uložení RDF entit je zapotřebí informace nějakým způsobem serializovat. V rámci RDF se nejčastěji používají tyto serializace:

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

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

Listing 3.1: RDF Turtle: W3C Enemies

```
@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 IRI `http://example.org/#green-goblin`, `rel:enemyOf` je predikát - v tomto případě zase objekt identifikovaný IRI `http://www.perceive.net/schemas/relationship/enemyOf` a `<#spiderman>` je objekt - zase identifikován IRI. Tento model dohromady poskytuje informaci, že objekt na dané IRI je nepřítelem Spidermana. `a foaf:Person` znamená, že se jedná o objekt Person a `foaf:name "Green Goblin"` 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 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 mohla 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 IRI, 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 [12]. Tato knihovna podporuje práci přímo s RDF serializacemi a díky procesoru ARQ také práci se SPARQL dotazy. Další možností pro použití byla knihovna Sesame [13]. 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* [14]. 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 [15], GSON [16]

  Tyto knihovny slouží pro práci s definicemi objektů v rámci API. Konkrétní objekty tříd jsou těmito knihovnami serializovány do JSON formátu.

- Log4j [17]

  Log4j je logovací systém, který je hojně využívaný v mnoha aplikacích. Jedná se o spolehlivý a jednoduše nastavitelný logovací systém.

- Typesafe [18]

  Typesafe je knihovna pro konfiguraci aplikací. Umožňuje snadnou konfiguraci přes konfigurační soubor a nahrání konfigurace do aplikace.

- JUnit [19]

  JUnit je testovací framework pro aplikce v Javě. Jedná se o nejpoužívanější testovací framework v Java komunitě.

- Jersey Test [20]

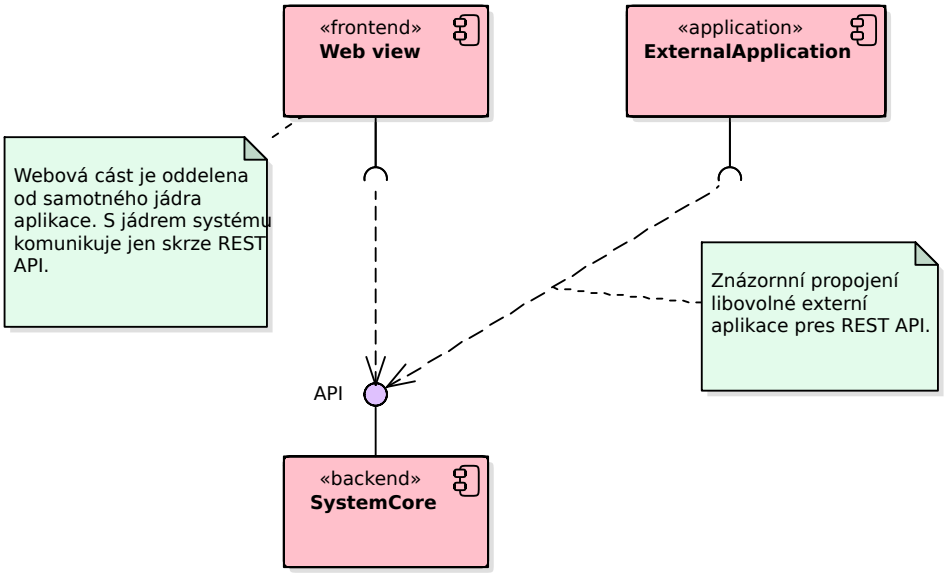
  Tato knihovna podporuje testování webových služeb. Umožňuje spustit webovou aplikaci v lokálně vytvořeném prostředí, ve kterém se poté dají testovat HTTP požadavky.

### 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 [21], který je snadno napojitelný na REST API serveru.




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

## 3.2 Návrh architektury

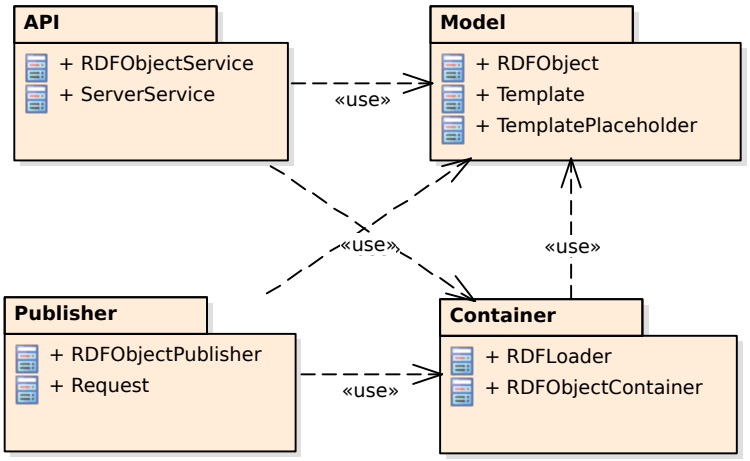
Na diagramu 3.1 lze vidět, že je systém rozdělen do 2 částí. První částí je webová aplikace, která slouží pro ulehčení administrace definic objektů. Hlavní funkcionalita tohoto serveru ale není touto částí ovlivněna, na API se lze napojit i z jiných aplikací. Tato část je napsaná ve frameworku AngularJS 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.

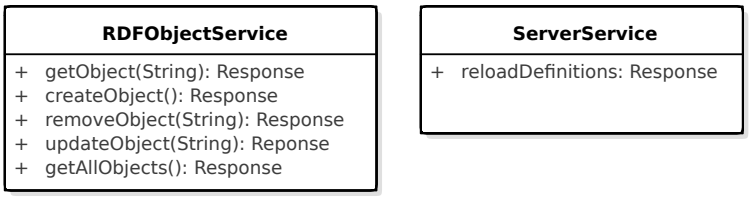
### 3.2.1 Jádro systému

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.

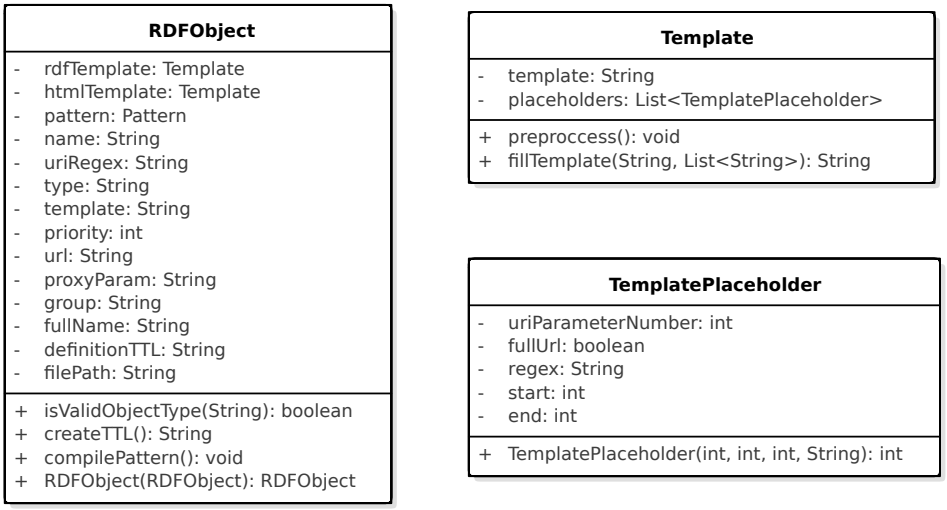
#### 3.2.1.1 Komponenta API

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. Třída `ServerService` pak obsahuje metodu, která se dá taktéž volat přes API a slouží ke znovunačtení všech definic z nastaveného uložiště.

#### 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.




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

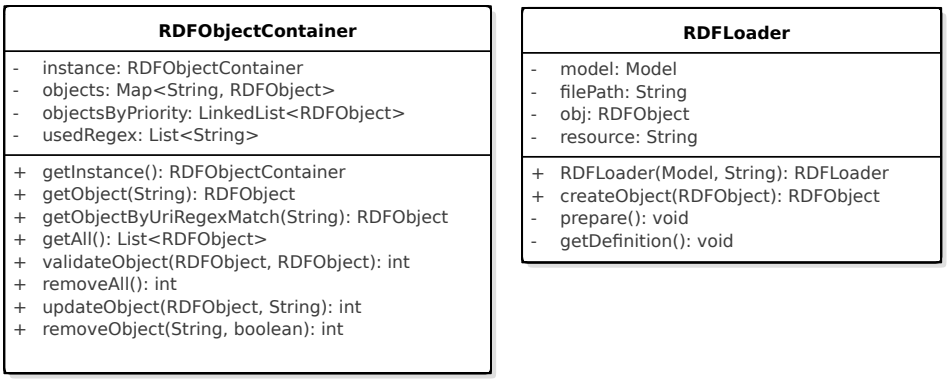
Třída `RDFObject` reprezentuje samotnou definici objektu. Obsahuje dva atributy typu `Template`. Prvním z nich je šablona definice (RDF serializace, nebo SPARQL dotaz) a druhou je HTML šablona sloužící pro zobrazení objektu v HTML formátu. Dále obsahuje metody jako jsou například validace a kompilace 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 předzpracování 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 nahrání definic (při startu serveru nebo znovunačtení definic). Cílem takto předzpracované šablony je urychlení generování objektů tak, aby se pouze dosazovaly hodnoty a případně aplikovaly podporované regulární výrazy.

#### 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




Obrázek 3.5: Komponenta container obsahuje třídy pro administraci objektů

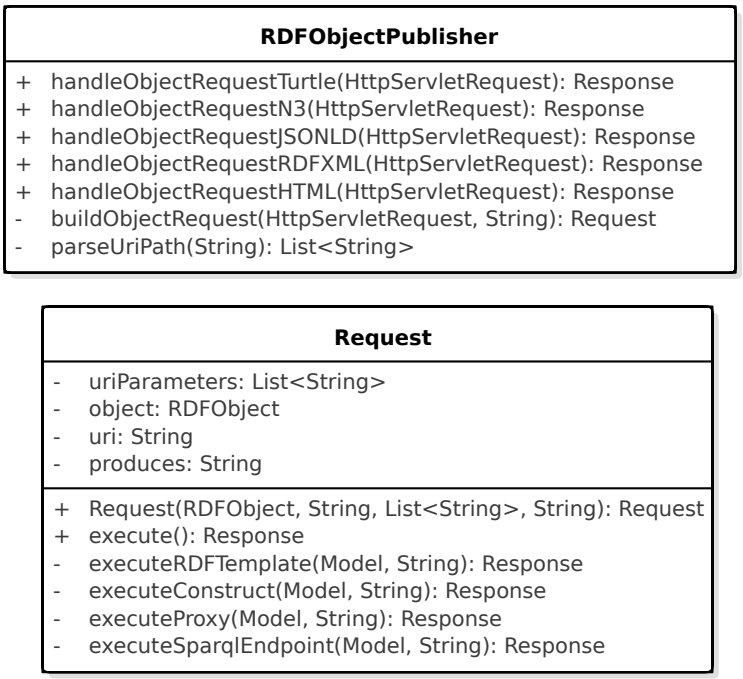
reprezentuje konkrétní definice.

Třída `RDFObjectContainer` 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 publisher obsahuje třídy pro zpracování požadavků klienta

## 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.

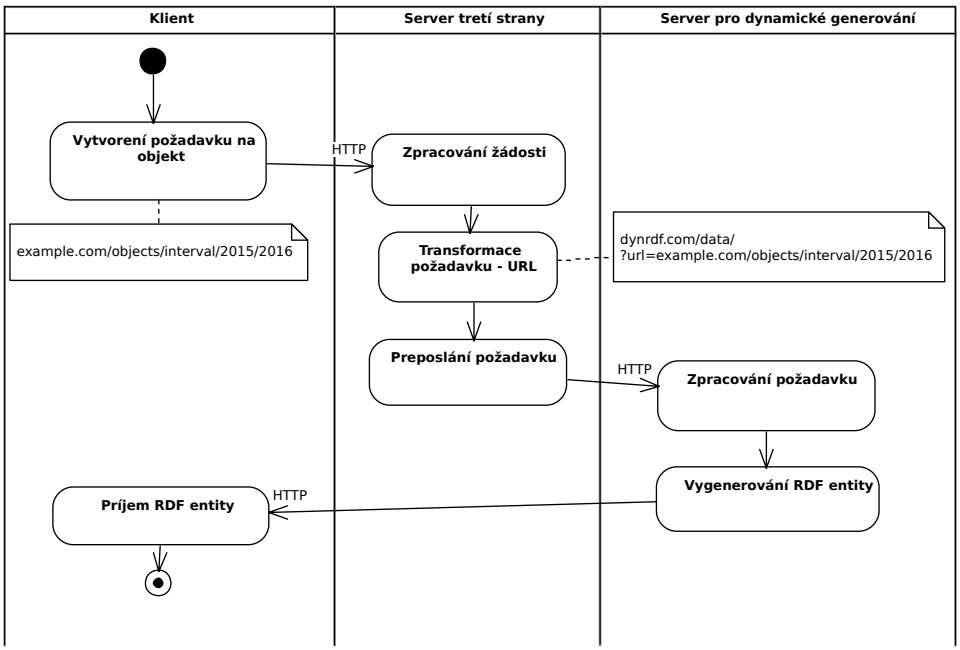
Prvním problémem je ten, že by musel být každý objekt identifikován IP adresou nebo doménou, kde tento server bude spuštěn. To by tedy znamenalo, že by tu nebyla možnost určit pro každý objekt extra URL v identifikátoru. Pokud by například funkce tohoto serveru chtěly využívat dvě různé společnosti, 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, kterému odpovídá konkrétní struktura 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, kdy 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. Regulární výraz by měl popisovat URL adresy, pod kterými bude výsledná RDF entita dostupná.

- 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 entity. 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>]]
```
(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 [22]. 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 nahrání 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

Tato kapitola obsahuje popis implementační částí této práce. Implementace je rozdělena na části dle komponent jádra systému uvedených v návrhu 3.2 a implementaci webové aplikace pro administraci definic. Z velké většiny vychází implementace z návrhu, od kterého se více liší pouze při implementaci optimalizovaného vyhledávání definic, jak je dále popsáno v kapitole Komponenta Container 4.2.

## 4.1 Komponenta Model

V této kapitole je popsána implementace komponenty Model dle návrhu 3.2.1.2. Tato komponenta obsahuje následující tři třídy:

- `RDFObject`

  Objekty této třídy reprezentují definice konkrétních objektů.

- `Template`

  Tato třída zastřešuje šablonovací systém pro šablony v RDF serializacích, SPARQL dotazech a HTML šablony.

- `TemplatePlaceholder`

  Objekty této třídy reprezentují jednotlivé placeholdery v šablonách.

Implementace těchto tříd je popsána v dalších kapitolách. Třídy `Template` a `TemplatePlaceholder` jsou popsány společně v kapitole Šablonovací systém.

### 4.1.1 Třída RDFObject

Tato třída popisuje konkrétní definice objektů. Jedná se o třídu, která neobsahuje téměř žádnou logiku aplikace, protože se jedná opravdu pouze o model. Kromě běžných getterů a setterů obsahuje pouze jednu složitější metodu, která

z atributů konkrétních objektů vytvoří ekvivalentní definici v Turtle formátu. Touto metodou je `createTTL()` a je využívána při vytváření nových definic při požadavcích v JSON formátu, jak je dále popsáno v kapitole o implementaci API 4.3.

### 4.1.2 Šablonovací systém

Tato kapitola popisuje implementaci šablonovacího systému. Z komponenty Model se jedná o popis implementace tříd `Template` a `TemplatePlaceholder`.

Třída `Template` dle návrhu 3.2.1.2 obsahuje atributy `template`, ve kterém je při načtení definice do kontejneru uložen řetězec definované šablony a atribut `placeholders`, ve kterém jsou drženy objekty třídy `TemplatePlaceholder` po předzpracování šablony.

Předzpracování šablony slouží ke zrychlení běhu systému pro generování RDF entit. Jedná se o metodu `preprocess()` 4.1.

Listing 4.1: Metoda preprocess(): předzpracování šablony

```
public void preprocess(){
      if(template == null) return; // proxy
      Matcher matcher = templatePattern.matcher(template);
      while (matcher.find()) {
          int start = matcher.start();
          int end = matcher.end();
          String regex = null;
          int groups = matcher.groupCount();

          // parameter number + regex
          if( groups > 1 ){
              regex = matcher.group(3);
          }
          int uriParameter = Integer.parseInt(matcher.group(1));

          TemplatePlaceholder record = new TemplatePlaceholder(start,
              end, uriParameter, regex);
          placeholders.add(record);
      }
  }
```

Tato metoda v definované šabloně hledá všechny placeholdery, ke kterým vytváří ekvivalentní objekty třídy `TemplatePlaceholder`. Placeholdery jsou nalezeny v šabloně s využitím knihovny pro regulární výrazy `java.util.regex`. Regulární výraz, který popisuje strukturu placeholedru dle návrhu v kapitole Šablonovací systém 6.4, má následující strukturu:

```
\[\s*@(\d+)\s*(,\s*[\"|\\"](.*)[\"|\\"])?\s*\]
```
(4.1)

Tento regulární výraz popisuje následující placeholdery:

- `[@0]`, `[@1]`, ...

  Placeholder, do kterého je později dosazena část URL nazývaná *vstupem placeholderu*, která je popsána v samostané kapitole 3.5.1.1. Tuto strukturu popisuje první část regulárního výrazu: `@(\d+)`.

- `[@0, "regex"]`, `[@0, \"regex\"]`, ...

  Dle návrhu lze ze vstupu placeholderu extrahovat část řetězce pro dosazení. Část za čárkou umožňuje definovat regulární výraz, který může obsahovat jednu skupinu, která se bude při generování dosazovat místo placeholderu. Regulární výraz palceholderu se nachází mezi uvozovkami, před kterými se může vyskytovat zpětné lomítko pro escapování uvozovek v RDF serializaci JSON-LD.

  Část placeholderu s regulárním výrazem popisuje část regulárního výrazu 4.1 `(,\s*[\"|\\\"](.*)[\"|\\\"])?`. Otazník na konci znamená, že se jedná o nepovinnou část.

V placeholderu se také může vyskytovat libovolný počet mezer, odřádkování či tabulátorů. Toho je v regulárním výrazu, který popisuje strukturu placeholderu, docíleno speciálním znakem `\s*`. Tento regulární výraz je i s příkladem pro ukázku dostupný na adrese `http://regexr.com/3ddcr`.

V metodě `preprocess()` se využívá třída `Matcher` z knihovny `java.util.regex`, která dle zadaného regulárního výrazu vyhledává v textu shody. Tyto shody, neboli nalezené placeholdery, jsou poté zpracovány. Jsou vytvořeny jejich ekvivalentní objekty třídy `TemplatePlaceholder` a jsou vloženy do seznamu placeholderů v k dané šabloně. Při požadavku klienta se poté pracuje hlavně nad seznamem těchto placeholderů a není už potřeba znovu vyhledávat každý placeholder v šabloně.

Šablonovací systém se stará hlavně také o vyplnění šablony při požadavku klienta. Třída `Template` obsahuje metodu `fillTemplate()`, která má jako vstupy URl požadavku a seznam parametrů URL adresy (parametry jsou zde myšleny řetězce mezi lomítky v URL adrese). Vyplnění šablony se provádí dle algoritmu 1.

V tomto algorimtu se pro každý předzpracovaný placeholder dosazuje do výsledného řetězce text před daným placeholderem a poté samotná hodnota placeholderu po dosazení části URL adresy (případně se aplikuje regulární výraz na tuto hodnotu). Na konci je výsledný text doplněn o zbývající část textu šablony a vznikne zde už výsledná vyplněná šablona. Může se jednat jak o RDF serializaci, SPARQL dotaz, tak i o HTML kód. Tato vyplněná šablona je poté navrácena k dalšímu zpracování.

V algoritmu 1 jsou uvedeny parametry třídy `TemplatePalceholder` jako `start`, `end`, `regex` a `paramNum`. Ty dle návrhu 3.2.1.2 popisují pozici placeholderu v šabloně, číslo parametru a případně regulární výraz placeholderu.

**Algorithm 1** Algoritmus vyplnění šablony při požadavku klienta

```
1: procedure Fill template
2: templateText := copy template.templateText
3: filledTemplate := ""
4:     for each placeholder in template.placeholders  do
5:
6:         filledTemplate .= templateText.substr(0, placeholder.start)
7:         paramValue := URI.get(placeholder.paramNum)
8:         if placeholder.regex is not null then
9:             filledTemplate .= paramValue.apply(placeholder.regex)
10:        else
11:            filledTemplate .= paramValue
12:        end if
13:
14:        templateText.remove(0, placeholder.end)
15:    end for
16:    filledTemplate .= templateText
17:
18:    return filledTemplate
19: end procedure
```

## 4.2 Komponenta Container

Komponenta Container slouží k manipulaci s definicemi objektů. Zahrnuje dle návrhu 3.2.1.3 třídy `RDFLoader` a `RDFObjectContainer`, které se starají o nahrání definic do aplikace, vyhledávání konkrétních definic apod. Tyto třídy jsou popsány v dalších kapitolách.

### 4.2.1 Třída RDFLoader

Tato třída se stará o načítání definic objektů ze souborového systému. Načtené definice poté zpracovává za pomoci knihovny Apache Jena [12] a vytváří již konkrétní instance třídy `RDFObject`, které reprezentují dané definice.

O načítání definic v Turtle formátu se stará rekurzivní metoda 4.3, která prohledává každou podsložku ve složce pro uložení definic objektů.

Tato třída z nalezených definic vytváří ekvivalentní objekty třídy `RDFObject` reprezentující definice objektů v metodě `createObject()`. V této metodě nejdřív dochází k volání metody `prepare()`, která mimo jiné validuje pomocí SPARQL dotazů danou definici v Turtle formátu, a poté k zavolání metody `getDefinition()` 4.2, která také za pomoci SPARQL dotazu a knihovny Jena vytváří objekt dané definice. Z knihovny Jena jsou zde použity třídy `Query`, `QueryExecution`, `QuerySolution` a další, které slouží k provedení SPARQL dotazu nad danou definicí v Turtle formátu.

Listing 4.2: Metoda pro vytváření ekvivalentních objektů definic z Turtle souborů. Nastavení hodnot definice se provádí voláním metody setParam()

```
private void getDefinition() throws Exception{
    String queryString = "PREFIX dynrdf: <" + Config.ObjectRDFS + ">\n"
      + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n"
      + "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n"
      + "PREFIX owl: <http://www.w3.org/2002/07/owl#>"
      + "PREFIX dcterms: <http://purl.org/dc/terms/>"
      + "SELECT ?type ?value \n"
      + "WHERE \n"
      + " { <" + resource + "> ?type ?value }" ;
    Query query = QueryFactory.create(queryString) ;
    try (QueryExecution qexec =
      QueryExecutionFactory.create(query, model)) {
        ResultSet results = qexec.execSelect() ;
        for ( ; results.hasNext() ; ) {
            QuerySolution soln = results.nextSolution() ;

            RDFNode n = soln.get("value") ;
            String val;
            if(n.isResource()){
                val = n.asResource().toString();
            }
            else{
                val = ((Literal)n).getLexicalForm();
            }

            Resource typeRes = soln.getResource("type") ;
            String type = typeRes.toString();

            Log.debug("Found: type=" + type + ", val=" + val);
            try{
                setParam(type, val);
            }
            catch(Exception ex){
                throw new Exception("Exception during reading" +
                  "and setting parameters of the RDFObject: "
                  + ex.getMessage());
            }
        }
    }
  }
```

Listing 4.3: Metoda pro načítání definic ze souborového systému

```
public static List<File> findObjectsDefinitionsRec(File dir)
  throws IOException {
      List<File> definitions = new ArrayList<>();
      File[] files = dir.listFiles();
      if(files != null){
          for (File file : files) {
              if (file.isDirectory()) {
                  definitions.addAll(findObjectsDefinitionsRec(file));
              } else {
                  String name = file.getName();
                  if(name.matches("^.*\\.ttl$")){
                      definitions.add(file);
                  }
              }
          }
      }
      else{
          throw new IOException("Cannot read directory: "
              + dir.getAbsolutePath());
      }

      return definitions;
  }
```

### 4.2.2 Třída RDFObjectContainer

Tato třída je implementována jako jedináček (Singleton). V celé aplikaci je tedy k dispozici pouze jedna isntance této třídy, ke které lze přistupovat voláním metody `RDFObjectContainer.getInstance()`. Tento typ třídy byl zvolen proto, že tato třída slouží jako databáze všech nahraných definic objektů a cílem je minimalizovat paměťovou náročnost tím, že by se objekty definic (objekty třídy `RDFObject`) neukládají na více místech a neduplikují se.

Třída je z většiny implementována dle návrhu 3.2.1.3. Liší se ale v systému vyhledávání definic dle priorit, jak je popsáno samostatně v kapitole 4.2.2.1.

`RDFObjectContainer` obsahuje několik atributů, které se vždy vážou na již vytvořené definice objektů. Každý z těchto atributů má jinou datovou strukturu vhodnou pro konkrétní účel. Mezi tyto atributy patří:

- `Map<String, RDFObject> objects`

  Tento kontejner slouží pro uložení objektů pod celým názvem. Je využíván při dotazován se na definice z API a při validaci nových definic, kdy jsou kontorovány na duplicitní názvy.

- `List<String> usedRegex`

Tento list obsahuje seznam regulárních výrazů již nahraných definic do aplikace. Slouží pro validaci u vytváření nových definic, kdy je zakázáno, aby dvě rozdílné definice měly stejný regulární výraz.

- `Map<String, LinkedList<RDFObject» groupPriority`

  V této mapě jsou uloženy definice objektů ve spojovém seznamu (pro účely priorit), který je mapován na názvy skupin jednotlivých definic. Jedná se oproti návrhu o optimalizovanou verzi určenou k rychlejšímu vyhledávání definic při požadavku klienta. Touto optimalizací se zabývá kapitola 4.2.2.1.

Třída `RDFObjectContainer` obsahuje především metody pro nahrání definic do kontejneru, aktualizaci, mazání a vyhledávání konkrétních definic. Tyto metody pracují se zmíněnými atributy. Pro příklad, jak tyto metody vypadají, je zde uvedena 4.4 metoda `getObjectByUriRegexMatch()`, která je volána při požadavku klienta a vyhledává definici, která by odpovídala požadované URL adrese.

Tato metoda nejdříve najde požadovanou skupinu, ve které by se definice měla nacházet. Pokud tato skupina není součástí požadavku, je definice vyhledávána v základní skupině kterou reprezentuje prázdný řetězec. Konkrétní definice je vyhledávána průchodem seznamu, který je udržován dle priorit definic při vkládání a aktualizaci definic. U každé definice je příchozí URL porovnána vúči regulárnímu výrazu dané definice a pokud je nalezena shoda, je výsledná definice navrácena. V opačném případě je navrácena hodnota `null`.

#### 4.2.2.1 Zefektivnění vyhledávání dle atributu group

Součástí definice objektu 3.4 je atribut skupina (group). Tento atribut měl původně sloužit pouze k přesnější identifikaci definic, nicméně byl využit i k efektivnějšímu vyhledávání definic při požadavku klienta.

V původním návrhu by se při každém požadavku klienta na vygenerování RDF entity musel v nejhorším případě procházet celý seznam vytvořených definic. Při průchodu tímto seznamem je porovnávána URL adresa s regulárním výrazem definic a pokud by k tomu mělo docházet u všech požadavků, jednalo by se o celkem pomalý proces.

Oproti návrhu tedy byla přidána možnost zadat při požadavku v GET parametru i parametr `group` a tím vytyčit pouze malou část všech definic v systému. Při požadavku klienta je pak vyhledávána definice, která by odpovídala požadované URL adrese, pouze v seznamu definic dané skupiny. Vyhledávání konkrétní definice je tímto způsobem urychleno a aplikace tedy umožňuje zpracovat více požadavků.

Implementace třídy `RDFObjectContainer` se oproti návrhu 3.2.1.3 liší v atributu `groupPriority`, který už není pouze spojovým seznamem dle priorit (původní atribut `objectsByPriority`), ale objekt třídy `Map`, díky kterému je

Listing 4.4: Metoda pro vyhledání definice dle požadované URL adresy

```
/**
  * Get first RDF object from container by regex match
  * @param uri String
  * @param group String
  * @return RDFObject | null if not found
  */
public RDFObject getObjectByUriRegexMatch(String uri, String group){
    if(groupPriority.get(group) == null){
        // unknown group
        return null;
    }
    for( RDFObject o : groupPriority.get(group) ){

        Pattern pattern = o.getPattern();
        Matcher matcher = pattern.matcher(uri);
        if (matcher.find()) {
            return new RDFObject(o); // return copy
        }
    }

    return null;
}
```

na každou skupinu vyskytující se v apliakci namapován vlastní seznam definic dle priorit.

Pro ověření výsledků byl také proveden test, který popisuje kapitola 5.3.

## 4.3 Komponenta API

Komponenta API obsahuje dle návrhu 3.2.1.1 třídy `RDFObjectService` a `ServerService`. Zodpovědností těchto tříd je zpracovávat požadavky pro administraci objektů.

Při implementaci API byly ve velkém množství využity 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. Příklad 4.5 ukazuje metodu `createObject()` včetně třídy, na kterou je díky použitým anotacím přesměrován požadavek na vytvoření nové definice z JSON formátu.

Listing 4.5: Příklad použití anotací při požadavku na vytvoření nové definice

```
@Path("/objects")
public class RDFObjectService {

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public Response createObject(String o){
        try{
            RDFObjectContainer container = RDFObjectContainer.getInstance();

            container.createObject(new Gson().fromJson(o, RDFObject.class));
        }
        catch( Exception ex ){
            return returnError(ex.getMessage());
        }

        return returnOK("Created.");
    }
}
```

Použité anotace mají následující význam:

- `@Path("/objects")` určuje cestu, na které jsou metody této třídy dostupné. Jedná se například o URL adresu `http://dyrdf.com/objects`. Pokud klient přistoupí na tuto adresu, tak je v této třídě vyhledána metoda dle následujících anotací, která má zpracovávat konkrétní požadavek klienta.
- `@POST` určuje HTTP metodu, kterou daná metoda zpracovává. Podobně existují i anotace `@GET`, `@PUT` a `@DELETE`.
- `@Consumes("application/json")` určuje, jaký `Content Type` [23] metoda zpracovává. V tomto případě se jedná o `application/json`.
- `@Produces("application/json")` určuje, v jakém formátu metoda poskytuje odpověď. Tuto hodnotu určuje parametr `Accept` v hlavičce HTTP požadavku.

Na metodu `createObject()` jsou ve výsledku mapovány požadavky, které obsahují a zároveň i požadují data v JSON formátu. Tyto požadavky musí být zároveň typu POST. Uvnitř metody je nad kontejnerem pro uchování definic zavolána metoda se stejným jménem, které je předán už přímo objekt třídy `RDFObject`, který vytvořila knihovna Gson [16] z JSON řetězce.

Podobných metod třída `RDFObjectService` obsahuje více. Oproti návrhu 3.2.1.1 obsahuje pro každou metodu jako je `createObject()` ekvivalentní metody, které ale odpovídají v jiných formátech, případně jiný formát akceptují.

Třída `ServerService` obsahuje aktuálně pouze jednu metodu, na kterou je mapován požadavek na znovunačtení definic objektů. Tato funkcionalita byla implementována pro načtení definic, které uživatel při běhu aplikace vloží do složky s definicemi.

## 4.4 Komponenta Publisher

Komponenta Publisher obsahuje dle návrhu 3.2.1.4 dvě třídy, které jsou zodpovědné za vyřízení požadavků klienta na vygenerování RDF entit.

Třída `RDFObjectPublisherService` se chová velmi podobně jako API třída `RDFObjectService`. Zpracování požadavků je směrováno na metody pomocí anotací, které byly pospsány v kapitole Implementace API 4.3. V těchto metodách, jak lze vidět na ukázce kódu 4.6, se dále uvnitř volá metoda `buildObjectRequest()`, která se kontejneru dotazuje na konkrétní definici a následně vytvoří objekt třídy `Request`, na který je předána zodpovědnost za vygenerování. Uvedený příklad oproti metodám v API obsahuje navíc anotaci `@Context`, díky které je do parametrů metody injektován objekt třídy `HttpServletRequest`, ze kterého je možné přistupovat k obsahu HTTP požadavku.

Listing 4.6: Metoda pro zpracování požadavku klienta na vygenerování RDF entity v Turtle formátu

```
@GET
@Produces("text/turtle")
public Response
    handleObjectRequestTurtle(@Context HttpServletRequest request) {

    Request objectRequest;
    try{
        objectRequest = buildObjectRequest(request, "TURTLE");
    }
    catch(RequestException ex){
        return return4xx(ex.getMessage());
    }

    return objectRequest.execute();
}
```

Funkcí uvedeného příkladu je zpracovávat požadavky na vygenerování RDF entit, které jako výstup očekávají Turtle formát. Zodpovědnost za vygenerování je v těle metody předána třídě `Request`.

Třída `Request` reprezentuje požadavky na už konkrétní RDF entity. Hlavní zodpovědností této třídy je na základě URL adresy a nalezené definice vygenerovat výslednou RDF entitu.

Tato třída obsahuje metody `executeConstruct()`, `executeProxy()` a další, jejichž úkolem je vygenerovat výslednou entitu dle typu definice. Příklad 4.7 ukazuje metodu pro generování RDF entity z definice typu SPARQL Construct 3.4.2. Vyplněná šablona obsahuje SPARQL Construct dotaz, který je pomocí třídy `QueryExecutionFactory` vykonán. Výsledek dotazu je zapsán do modelu RDF Entity a na řádku `model.write(out, produces);` je serializován do zvolené RDF serializace, která je uložena v proměnné `produces`.

Listing 4.7: Metoda pro generování entit ze SPARQL Construct definicí

```
private Response executeConstruct(Model model, String filledTemplate){
     try (QueryExecution qexec = QueryExecutionFactory
                    .create(filledTemplate, model)) {
         model = qexec.execConstruct();
     }
     catch(Exception e){
         return Response.status(404).build();
     }

     StringWriter out = new StringWriter();
     model.write(out, produces);

     return Response.status(200).entity(out.toString()).build();
  }
```

## 4.5 Webová aplikace pro administraci definic

Implementace webové aplikace je postavena na frameworku AngularJS [21]. Pro komunikaci s API byla implementována REST služba, která definuje chování frameworku při administraci definic. Tato služba je uvedena v příkladu 4.8. V rámci této služby bylo potřeba pouze nastavit URL adresu API a HTTP metodu pro aktualizaci definic.

Aplikace obsahuje pouze dva kontrolery. `RDFObjectEntityController` se stará o tvorbu a aktualizaci definic a `OverviewController` o zobrazení již vytvořených definic klientovi. Implementace těchto kontrolerů je uložena v příloze ve zdrojových kódech v souboru `src/main/webapp/js/controllers.js`.

Pro vizuální část aplikace byl použit front-endový framework Bootstrap [24].

Listing 4.8: Služba pro komunikaci s API ve webové aplikaci. Proměná REST_SERVICE je definovaná globálně a jejím obsahem je adresa serveru pro dynamické generování

```
angular.module('dynrdfApp.services',[])
  .factory('RDFObject',function($resource){
    return $resource(REST_SERVICE + 'objects/:id',{id:'@id'},{
        update: {
            method: 'PUT'
        }
    }
    );
})
```

Kapitola 5

# Testování

Cílem testování této aplikace je ověřit celkovou funkčnost systému. Při úpravách zdrojových kódů může dojít k ovlivnění chování již implementovaných metod a tudíž k neočekávanému chování celé aplikace. Nestandartní chování může také ovlivnit prostředí (např. jiná verze Javy), ve kterém je aplikace spuštěna. Cílem testů je toto nestandartní chování odhalit a upozornit na něj vývojáře s cílem aby toto chování analyzoval a opravil.

Jedním typem testování jsou testy implementovaných funkcí přes **unit testy** a **integrační testy**. Druhým typem testů je **testování výkonu**. Testování aplikace dle těchto typů je popsáno v následujících kapitolách.

## 5.1 Unit testy

Unit testy slouží k otestování správné funkčnosti jednotlivých částí systému. V této aplikaci jsou tímto způsobem testovány nejvíce metody tříd `Template` 3.2.1.2 a `RDFObjectContainer` 3.2.1.3. Ostatní třídy jsou testovány na úrovni integračních testů.

### 5.1.1 Testy třídy Template

Třída Template pracuje s šablonami jednotlivých definic. Nerozlišuje přitom, zda se jedná o HTML šablonu nebo například o šablonu v Turtle serializaci. Tato třída obsahuje 2 metody - `preprocess()`, která vyhledává a ukládá si pro další použití informace o placeholderech v šabloně a metoda `fillTemplate()`, která vyplňuje šablonu při požadavku klienta.

Testy těchto metod jsou obsaženy ve třídě `TemplateTest`. Příkladem jednoho z testů je test na vyplnění šablony včetně extrahování části URL adresy díky regulárnímu výrazu. V tomto testu 5.1 je vytvořena šablona obsahující pouze jeden placeholder s regulárním výrazem, který z URL adresy extrahuje část za řetězcem `http://`. Testovanou URL adresou je zde

`http://test.com/a/b/c` a výsledný řetezec po vyplnění šablony je testován na obsah `test.com/a/b/c`.

Listing 5.1: Unit test na vyplnění placeholderu v šabloně s regulárním výrazem

```
@Test
public void testRegexTemplate(){
    String plainTemplate = "[@0, \"http://(.*)\"]";
    Template template = new Template(plainTemplate);
    template.preprocess();
    Assert.assertEquals(1, template.getPlaceholders().size());
    String testUrl = "http://test.com/a/b/c";
    String filledTemplate = template.fillTemplate(testUrl,
      Arrays.asList(testUrl.split("/")));

    String expected = "test.com/a/b/c";
    Assert.assertEquals(expected, filledTemplate);
}
```

### 5.1.2 Testy třídy RDFObjectContainer

Třída RDFObjectContainer 3.2.1.3 obsahuje metody pro manipulaci s definicemi objektů, jako je přidání, smazání, aktualizace a také validaci těchto definic. Ke každé z těchto metod jsou napsány testy ve třídě `RDFObjectContainerTest`, které ověřují jejich funkcionalitu. Validační testy pokrývají všechny typy nevalidních definic, jako jsou duplicitní celé názvy definic (stejná skupina a název), chybějící parametry, nevalidní URL například pro Proxy objekt aj. Pro testování těchto funkcionalit jsou předpřipravené definice v adresáři `textObjects`.

Příkladem těchto testů je například validační test na špatnou strukturu URL adresy 5.2, která v tomto případě neobsahuje protokol, nebo test pro přidání definice do kontejneru 5.3, který definici vytvoří a přidá do kontejneru a poté se ho snaží získat zpět.

## 5.2 Integrační testy

Integrační testy slouží k testování funkčnosti více komponent dohromady. V rámci této aplikace se jedná o komponenty dle návrhu 3.2 (komponenty API, Model, Container a Publisher). Pro úspěšné spuštění integračních testů je potřeba připojení k internetu, protože se testuje i generování RDF entit u definic typů Proxy a SPARQL Endpoint, které se připojují ke SAPRQL Endpointu portálu DBpedia na adrese `http://dbpedia.org/sparql`. Pro tyto testy se používají testovací definice dostupné v příloze v adresáři `testObjects`.

Listing 5.2: Unit test třídy RDFObjectContainer: Malformed URL

```
@Test(expected = ContainerException.class)
public void validateProxyMalformedUrl() throws ContainerException{
   RDFObjectContainer container = RDFObjectContainer.getInstance();
   // missing protocol - URL validator is set to check protocol
   RDFObject o = new RDFObject("name", "tests", "validations",
     "PROXY", "empty template", 1, "test.com", "param", "html", null);

   container.validateObject(o, false, null);
}
```

Listing 5.3: Unit test třídy RDFObjectContainer: Vytvoření a vložení definice

```
@Test
public void testCreate(){
    RDFObjectContainer container = RDFObjectContainer.getInstance();
    RDFObject o = new RDFObject("createTestContainer", "tests",
      "createTestContainer", "TURTLE", "empty template", 1,
      "", "", "html", null);
    boolean ok = true;
    try{
        container.createObject(o);
    }
    catch(Exception ex){
        ok = false;
    }
    Assert.assertTrue(ok);
    Assert.assertNotNull(container.getObject(o.getFullName()));
}
```

### 5.2.1 Testy třídy Request

Třída `Request` 3.6 má na starost zpracování požadavku klienta na vygenerování už konkrétní RDF entity. Obsahuje metody pro vygenerování entit, jako jsou `executeProxy()`, `executeRDFTemplate()` a další dle typů definice. Cílem testů pro tuto třídu je ověřit integraci s ostatními komponentami pro generování RDF entit. Každý typ definice je testován na všechny podporované výstupní serializace.

Jako příklad 5.4 je zde uveden test metody `executeSPARQLEndpoint()`, který ověřuje funkčnost generování objektu typu SPARQL Endpoint. Při tomto testu dochází uvnitř aplikace ke komunikaci se serverem DBpedia, což je určeno testovanou definicí `loggerEndpoint`, která je k dispozici v příloze v adresáři `testObjects/logger_endpoint.ttl`. Před provedením tohoto testu je inicializován kontejner a jsou do něj nahrány testovací definice. V rámci to-

hoto testu se ověřuje, zda kontejner obsahuje danou definici a IRI vygenerované entity se shoduje s očekávaným identifikátorem nastavený v dané definici. Testování ostatních metod je velice podobné tomuto testu.

Listing 5.4: Test metody executeSPARQLEndpoint()

```
@Test
public void testRequestExecuteSPARQLEndpoint(){
    RDFObject o = RDFObjectContainer.getInstance()
                .getObject("dynrdf/loggerEndpoint");
    Assert.assertNotNull(o);

    String uri = "http://logservice.com/data/loggerEndpoint/" +
                "42/2016-05-01/10:00:23/org.dynrdf.Request/" +
                "some exception msg";
    List<String> params = new ArrayList<String>(
           Arrays.asList(uri.split("/")));
    Request request = new Request(o, uri, params, "TURTLE");
    Response r = request.execute();

    Assert.assertEquals(200, r.getStatus());
    String result = (String)r.getEntity();

    StringReader sr = new StringReader(result);
    Model model = ModelFactory.createDefaultModel();
    model.read(sr, null, "TURTLE");

    String res = selectresource(model);
    Assert.assertEquals("http://logservice.com/data/loggerEndpoint#42",
                      res);
}
```

### 5.2.2 Testy komponent API a Publisher

Cílem testů těchto komponent je ověřit funkčnost samotných HTTP požadavků pro administraci objektů a požadavky na RDF entity. Testování těchto komponent je prováděno we webovém kontejneru, který je spuštěn díky testovacímu frameworku Jersey Test [20] před každým testem.

Komponenta API 3.3 obsahuje třídy `RDFObjectService`, která se stará o zpracování HTTP požadavků administrace definic, a třídu `ServerService` která obsahuje zatím pouze metodu `reloadDefinitions()` pro znovu nahrání definic do systému. Všechny metody, respektive všechny požadavky, které jsou na tyto metody mapovány, jsou testovány vůči testovacím definicím dostupných v příloze v adresáři `testObjects`.

Příkladem testu API je test 5.5 na vytvoření definice, který posílá informace o definici v JSON formátu. Otestováním tohoto požadavku je ověřena funkčnost požadavku na vytvoření definice z webové aplikace, ta odesílá data v JSON formátech. V tomto testu je neprve vytvořen z objektu definice požadavek na vytvoření, který je poté odeslán. Následuje požadavek na získání definice tohoto objektu ze serveru, který je kontrolován na úspěch (status kód 200).

Listing 5.5: Test vytvoření definice přes API v JSON formátu

```
@Test
public void testCreateJson() {
    RDFObject o = new RDFObject("testobj1", "tests", "testobj1","
                               "TURTLE", "empty template", 1,
                               "", "", "html", null);

    final Response res = target("objects").request()
             .post(Entity.entity(o, MediaType.APPLICATION_JSON_TYPE));
    Assert.assertEquals(200, res.getStatusInfo().getStatusCode());
    // get object
    final Response resGet = target("objects/" + o.getFullName())
             .request(MediaType.APPLICATION_JSON_TYPE).get();
    Assert.assertEquals(200, resGet.getStatusInfo().getStatusCode());
}
```

V rámci komponenty Publisher jsou testovány metody třídy `RDFObjectPublisherService`. Ty zpracovávají požadavky na konkrétní RDF entity. Testování této třídy má za cíl ověřit funkčnost generování entit všech typů pro všechny podporované výstupní RDF serializace.

## 5.3 Testování výkonu

Cílem testování výkonu je zjistit, kolik požadavků za sekundu je tato aplikace schopna odbavit a s jakou dobou odpovědi. Na tyto hodnoty není předem kladen požadavek, který by aplikace musela splňovat. Jedná se o orientační test.

Testování výkonu bylo provedeno aplikací Apache JMeter [25]. Aplikace byla pro tyto účely nasazena do webového kontejneru Glassfish 4.1.1 [26] 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. Throughput je oproti těmto atributům testu více imunní na výkyvy, protože je vypočítáván algoritmicky a je brán v potaz například počet spuštěných testů na testovacím stroji.

Při testování výkonu byly testovány všechny typy definic. Poměr definic typů RDF serializací vůči ostatním typům byl trojnásobný, protože se předpokládá s větším využitím definic tohoto typu.

Těmito testy se ověřovalo také efektivnější vyhledávání při použití parametru `group` v URL adrese. Tato optimalizace je popsána v kapitole implementace - RDFObjectContainer 4.2.2.1. Testovány byly dva případy. V prvním případě, jehož výsledky jsou uvedeny v tabulce 5.1, byly testovány dotazy včetně parametru `group` v URL adrese. Při druhém testu, jehož výsledky ukazuje tabulka 5.2, byl tento parametr vynechán.

Testy výkonu ukázaly poměrně velký rozdíl mezi testovanými případy. Tím bylo ověřeno, že zmíněná optimalizace byla poměrně důležitým optimalizačním krokem. Při předpokladu, že některé požadavky na vygenerování RDF entit nebudou obsahovat zmiňovaný parametr `group`, je výsledný počet požadavků za sekundu větší než 100. Tento výsledek je na dané konfiguraci vyhodnocen jako postačující.

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** | **Median** | **Min** | **Max** | **Throughput** |
|---|---|---|---|---|---|
| Turtle | 300 | 22 | 10 | 5138 | 29.8 |
| RDF/XML | 300 | 19 | 9 | 991 | 56.3 |
| NTriples | 300 | 24 | 9 | 5132 | 31.6 |
| JSON-LD | 300 | 87 | 9 | 5141 | 33.8 |
| SPARQL Construct | 100 | 36 | 13 | 950 | 32.3 |
| SPARQL Endpoint | 100 | 315 | 89 | 5190 | 13.5 |
| Proxy | 100 | 765 | 325 | 5212 | 13.1 |
| TOTAL | 1500 | 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** | **Median** | **Min** | **Max** | **Throughput** |
|---|---|---|---|---|---|
| Turtle | 300 | 1549 | 98 | 5483 | 19 |
| RDF/XML | 300 | 922 | 134 | 5419 | 21.2 |
| NTriples | 300 | 3539 | 232 | 5527 | 19.4 |
| JSON-LD | 300 | 4063 | 970 | 5503 | 20.4 |
| SPARQL Construct | 100 | 3979 | 1829 | 5467 | 8.4 |
| SPARQL Endpoint | 100 | 3936 | 3490 | 4518 | 8 |
| Proxy | 100 | 4306 | 3244 | 5159 | 7.5 |
| TOTAL | 1500 | 3596 | 98 | 5527 | 91.4 |

Kapitola 6

# Uživatelská dokumentace

Tato kapitola obsahuje dokumentaci k vyvíjené aplikaci. Touto aplikací je webový server pro poskytování dynamicky generovanýcý objektů v RDF [27] formátech.

Cílem této aplikace je uživatelům usnadnit tvorbu objektů v RDF formátech pomocí jednotných šablon pro výsledné RDF entity. Generování RDF entit přes tuto aplikaci má tyto výhody:

- **Jedna šablona pro mnoho objektů**

  Uživatel může vytvořit jednu definici s šablonou pro mnoho (až nekonečně mnoho) podobných RDF entit, které se liší pouze v hodnotách atributů, ale strukturu mají stejnou (nebo velmi podobnou).

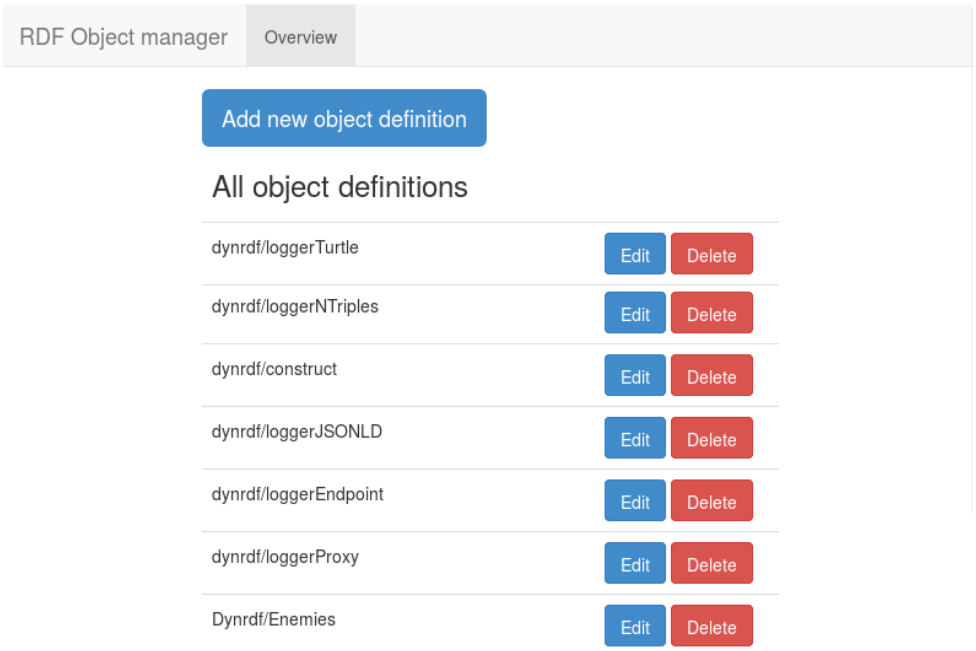
- **Malá kapacita uložišť**

  Díky jednotné šabloně pro mnoho RDF entit není potřeba investovat do velkých úložných prostorů. Zapotřebí je mít pouze kapacity pro uložení definic těchto objektů, které jsou v porovnání s ukládáním všech RDF entit minimální.

- **Čas pro údržbu**

  Tato aplikace také šetří čas uživatelů při údržbě již publikovaných RDF entit. Pokud chce uživatel něco změnit v publikovaných RDF entitách, nemusí tak dělat pro každou entitu zvlášť, ale stačí pouze změnit šablonu na jednom místě a tato změna se automaticky promítne do všech výsledných RDF entit při dalším požadavku klienta.

V této aplikaci je uživateli umožněno nadefinovat si vlastní RDF entity a vytvořit si k nim šablony, ze kterých jsou následně dynamicky generovány dle příchozí URL. K administraci definic slouží webová aplikace a API. Klientovi jsou výsledné entity dostupné v několika RDF serializacích (Turtle, N-Triples, JSON-LD a RDF/XML) a jako webová stránka (HTML dokument).




Obrázek 6.1: Přehled definic ve webové aplikaci - hlavní stránka

## 6.1 Webová aplikace

Pro administraci definic může uživatel využít webovou aplikaci. V této aplikaci je mu umožněno zobrazit si seznam definic, vytvářet, upravovat nebo smazat tyto definice.

### 6.1.1 Přehled definic

Při přístupu na hlavní stránku webové aplikace je uživateli k dispozici seznam již vytvořených definic. U každé definice se vyskytují dvě tlačítka - pro zobrazení definice (a následnou editaci) a pro smazání definice. Pro založení nové definice může uživatel kliknout na tlačítko „Add new object definition“ a aplikace ho přesměruje na stránku s formulářem pro založení nové definice. Hlavní stránku s přehledem lze vidět na obrázku 6.1

### 6.1.2 Vytvoření nové definice

Po kliknutí na tlačítko „Add new object definition“ je uživatel přesměrován na formulář pro založení nové definice. Tento formulář je zobrazen na obrázku 6.2. Uživatel zde nastavuje všechny atributy pro generování objektů. Ke každému atributu je na stránce také k dispozici nápověda, která se uživateli zobrazí po kliknutí na obrázek otazníku. Těmito atributy jsou:

- Název definice a skupina

  Název a skupina tvoří dohromady celé jméno definice ve formátu *<skupina>/<název>*. Toto označení definice musí být v celém systému unikátní. Uživatel může pojmenovat více definic stejným jménem, ale tyto definice musí mít rozdílné skupiny.

  Zavedení parametru *skupina* má také výkonostní důvody, které jsou popsány v kapitole Přístup k objektům 6.5.

- Regulární výraz

  Konkrétní definice jsou v systému identifikovány regulárním výrazem. Při požadavku klienta na konkrétní RDF entitu se server pokouší najít v definicích regulární výraz, který požadované URL adrese odpovídá. Pokud tuto definici nalezne, tak podle ní výslednou entitu vygeneruje.

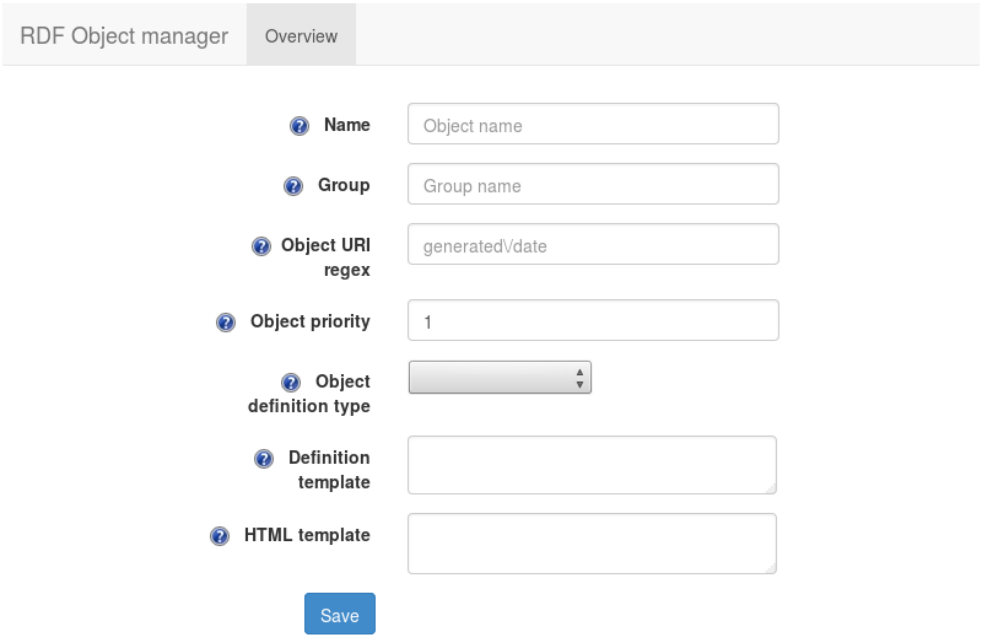
  Například URL adresa `http://company.com/objects/time/year/2016` odpovídá regulárnímu výrazu „`\/objects\/time`“, což je možné vyzkoušet si i online na adrese `http://regexr.com/3dddd`. Pokud požadavek klienta obsahuje tuto URL adresu a v systému existuje definice s daným regulárním výrazem, je výsledný objekt vygenerován dle této definice.

- Priorita

  Tento atribut určuje pořadí definic, ve kterém bude systém hledat odpovídající definici při požadavku klienta. Mezi jednotlivými definicemi může docházet ke konfliktům regulárních výrazů. Tyto výrazy nemusí být stejné, ale mohou popisovat velmi podobné struktury. Při nedefinovaném pořadí vyhledávání v definicích může docházet k nalezení shody u špatných definic a tudíž k vygenerování špatného objektu. Nastavením správných priorit lze tomuto problému alespoň zčástí předcházet. Definice jsou prohledávány v pořadí od nejvyšší (1) priority po nejmenší.

- Typ definice

  Typ definice určuje, jak bude generování objektu probíhat. Aplikace podporuje několik typů definic. Jedná se o RDF serializace (Turtle, JSON-LD, N-Triples a RDF/XML), SPARQL dotazy (lokální Construct, nebo Construct/Describe dotaz na SPARQL endpointu) a typ Proxy, který funguje na principu přeposílání požadavků. Všechny tyto typy jsou dále popsány v kapitole Typy definic 6.3




Obrázek 6.2: Formulář pro založení a editaci definice

- Šablona definice

  Typ šablony závisí na typu definice. U RDF serializací se jedná o šablonu v konkrétní serializaci, zatímco u SPARQL typů se jedná o SPARQL dotaz. V šablonách se mohou využívat placeholdery pro dosazení parametrů z URL adresy. Šablonovací systém včetně formátu je popsán v kapitole Šablonovací systém 6.4. U typu Proxy se tento atribut nevyplňuje.

- HTML šablona

  HTML šablona slouží pro zobrazení dat o objektu v HTML formátu při požadavku z webových prohlížečů. Stejně jako u šablony definice se mohou využívat placeholdery pro dosazení parametrů z URL adresy.

### 6.1.3 Editace definic

Po kliknutí na tlačítko „Edit“ v přehledu definic je uživatel přesměrován na informace o dané definici. Tyto informace jsou přístupné v editovatelném formuláři. Tento formulář je strukturou totožný s formulářem pro tvorbu nových definic.

### 6.1.4 Mazání definic

Uživatel může smazat konkrétní definici přes webovou aplikaci kliknutím na tlačítka „Delete“ a následním potvrzením. Smazanou definici již nejde žádným způsobem znovu obnovit.

## 6.2 API

Administrace definic je také možná přes API. API podporuje všechny operace, které jsou popsány v kapitole 6.1. Struktura API včetně dotazů je popsána v návrhu v kapitole API 3.6.

## 6.3 Typy definic

U každé následující definice jsou uvedeny příklady pro stejný typ objektu - logovací systém, který je také dostupný v příloze v ukázkových definicích. Jedná se o objekt logované informace nastavený pro URL adresu, která na konci obsahuje mezi lomítky hodnoty atributů pro ID logu, datum, zprávu a název třídy.

- **RDF serializace**

  Tento typ se dělí na definice dle konkrétních serializací - Turtle, N-Triples, RDF/XML a JSON-LD. Uživatel do šablony RDF entity dosadí již serializovaná data a na místech, kam se mají dosadit parametry z URL adresy, vytvoří placeholdery. Následující příklad šablony je typu Turtle:

Listing 6.1: Šablona RDF entity v Turtle formátu

```
@base <[@0, "^([^/]*\/[^/]*\/[^/]*\/[^/]*\/[^/]*)\/"]> .
@prefix rlog:
<http://persistence.unileipzig.org/nlp2rdf/ontologies/rlog#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#[@6]> # log id
       a rlog:Entry ;
       rlog:level rlog:WARN ;
       rlog:date "[@7]T[@8]Z"^^xsd:dateTime ;
       rlog:className "[@9]";
       rlog:message "[@10]".
```

- **SPARQL Construct**

  Šablonou pro tento typ je SPARQL Construct dotaz, kde se jednotlivé parametry mohou například vytvářet pomocí funkce *BIND()*. Tento typ

definice díky jazyku SPARQL doplňuje RDF serializace o další funkcionality tohoto dotazovacího jazyka. Příklad stejného objektu pro SPARQL dotaz:

Listing 6.2: Šablona SPARQL dotazu

```
PREFIX rlog:
<http://persistence.unileipzig.org/nlp2rdf/ontologies/rlog#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {
?res a rlog:Entry ;
       rlog:level rlog:WARN ;
       rlog:date "[@7]T[@8]Z"^^xsd:dateTime ;
       rlog:className "[@9]";
       rlog:message "[@10]".
  }
WHERE {
    BIND (URI('[@0,
    "^([^/]*\/[^/]*\/[^/]*\/[^/]*\/[^/]*)\/"]#[@6]')
    AS ?res) }
```

- **SPARQL Endpoint**

  Tento typ je rozšířením typu SPARQL Construct. Dotaz je spouštěn vzdáleně na SPARQL endpointu, přes který je možné přistupovat k datasetům které jsou na endpointu dostupné. Tento typ podporuje dva typy dotazů - Construct a Describe a narozdíl od předchozího typu může obsahovat komplikovanější dotazy právě nad zmíněnými datasety.

- **Proxy**

  Uživatel může nakonfigurovat přeposílání požadavků na jiný server, který poskytuje také data v RDF serializacích. Jedná se o případy, kdy uživateli nestačí výše zmíněné typy a potřebuje nějaký sofistikovanější generátor pro konkrétní objekt. Server tedy pouze přeposílá požadavek a výsledná data poté serializuje dle požadavku klienta.

## 6.4 Struktura šablony

Strukturou šablony se rozumí text, který uživatel vyplní jako atribut „šablona definice“ nebo „HTML šablona“. Do šablon jsou při přístupu klienta doplněny informace z URL adresy a následně je za použití této šablony vygenerována výsledná HTML stránka nebo RDF entita. Struktura šablon je popsána v kapitole Šablonovací systém 6.4.

## 6.5 Vygenerování RDF entity na žádost klienta

Pro vygenerování RDF entity klient přistupuje na URL adresu ve formátu `http://example.com/data/?group=<group>&url=<url>`, kde:

- `example.com` označuje adresu serveru
- `<group>` označuje skupinu, ve které se nachází požadovaná definice. Jedná se o nepovinný parametr, ale je doporučeno využívat ho při každém požadavku. Vyhledávání konkrétní definice je s tímto parametrem rychlejší a tedy i klientovi je rychleji poskytnuta výsledná vygenerovaná entita. Jedná se o parametr „skupina“, který uživatel zadává při tvorbě definice objektu, více je uvedeno v kapitole Definice objektů 3.4.
- Parametr `<url>` identifikuje konkrétní definici objektu a je zároveň identifikátorem výsledné vygenerované RDF entity (IRI). Definice objektů obsahuje parametr „regex“ - regulární výraz, který uživatel nastavuje tak, aby mu odpovídaly URL adresy, které mají identifikovat konkrétní definici a entitu. Tento regulární výraz je popsán v kapitole 6.1.2

### 6.5.1 Příklad definice a výsledné RDF entity

Pro shrnutí informací o generování a definic objektů bude sloužit příklad definice objektu `loggerTurtle`, který je dostupný v příloze ve složce ukázkových objektů `objects`, jedná se o soubor `logger_turtle.ttl`.

#### 6.5.1.1 Definice objektu

Objekt `loggerTurtle` je definován těmito parametry:

- **název**: loggerTurtle
- **skupina**: dynrdf
- **regulární výraz**: `loggerTurtle`
- **priorita**: 1
- **typ definice**: TURTLE
- **šablona definice**: šablona definice je uvedena v kódu 6.3
- **HTML šablona**: HTML šablona je uvedena v kódu 6.5.1.2

Regulární výraz `loggerTurtle` a struktura šablon předepisují očekávanou strukturu URL adresy. Tou může být URL adresa `http://example.com/data/?group=<group>&url=<url>`, kde parametry `group` a `url` obsahují hodnoty:

Listing 6.3: Šablona RDF entity v Turtle formátu pro objekt loggerTurtle

```
@base <[@0, "^([^/]*\/[^/]*\/[^/]*\/[^/]*\/[^/]*)\/"]> .
@prefix rlog:
<http://persistence.unileipzig.org/nlp2rdf/ontologies/rlog#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#[@6]> # log id
       a rlog:Entry ;
       rlog:level rlog:WARN ;
       rlog:date "[@7]T[@8]Z"^^xsd:dateTime ;
       rlog:className "[@9]";
       rlog:message "[@10]".
```

Listing 6.4: HTML šablona pro objekt loggerTurtle

```
<!doctype html>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Logger</title>
    <meta name="description" content="Dynrdf logger example">
  </head>

  <body>
    Log record: [@10] <br />
    Log ID: #[@6] <br />
    Level: Warn <br />
    Time: [@7] [@8] <br />
    Class name: [@9]
  </body>
</html>
```

- `<group>`: dynrdf
- `<url>`: `http://logservice.com/data/loggerTurtle/42/2016-05-01/10:00:23/org.dynrdf.Request/some exception msg`

V této URL adrese parametr `group` odpovídá parametru definice (`dynrdf`) a URL adresa v parametru `url` odpovídá regulárnímu výrazu `loggerTurtle`, protože je přímo v URL uveden text `../loggerTurtle/...` Vztah mezi URL adresou a regulárním výrazem je možné ověřit si online na adrese `http://regexr.com/3ddne`.

URL adresa se v šablonovacím systému dělí na následující atributy mezi lomítky, které může uživatel použít v šablonách:

- **@0**: celá URL adresa
- **@1**: http:
- **@2**: prázdný řetězec (mezi //)
- **@3**: logservice.com
- **@4**: data
- **@5**: loggerTurtle
- **@6**: 42
- **@7**: 2016-05-01
- **@8**: 10:00:23
- **@9**: org.dynrdf.Request
- **@10**: some exception msg

Pokud je v šabloně uvedeno záporné číslo parametru, nebo URL adresa po zpracování neobsahuje dostatečný počet parametrů, je uživateli vrácena odpověď s kódem `400 - Bad Request`. Pokud by v tomto příkladě URL adresa neobsahovala poslední parametr (zprávu výjimky „some exception msg“, nebo jinou zprávu), parametr `@10` by tedy nebyl nalezen a klientovi by byla odeslána odpověď se zmíněným kódem.

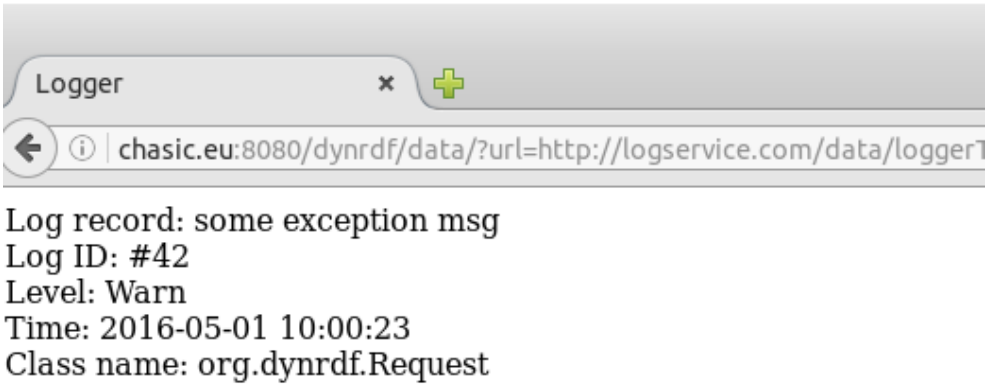
#### 6.5.1.2 Výsledná RDF entita

Uživateli je přístupem na URL adrese uvedené v kapitole 6.5.1.1 vygenerována výsledná entita. Typ serializace si vždy určuje klient technikou Content Negotioation [2]. V tomto příkladě je uveden výsledek ve formátu Turtle v kódu 6.5

Listing 6.5: Výsledná entita ve formátu Turtle pro příklad objektu loggerTurtle

```
@prefix rlog:
<http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog/rlog.ttl> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://logservice.com/data/loggerTurtle#42>
       a rlog:Entry ;
       rlog:className "org.dynrdf.Request" ;
       rlog:date "2016-05-01T10:00:23Z"^^xsd:dateTime ;
       rlog:level rlog:WARN ;
       rlog:message "some exception msg" .
```




Obrázek 6.3: Vygenerovaná HTML stránka pro příklad loggerTurtle

Přístupem klienta přes webový prohlížeč je vygenerována HTML stránka dle HTML šablony . HTML šablona v tomto příkladu obsahuje jen přehled atributů ekvivalentní RDF entity. Vzhled a obsah výsledné stránky si definuje každý uživatel při tvorbě definic sám. Tato stránka je zobrazena na obrázku 6.3.

## 6.6 Instalace serveru

Instalaci serveru je možné provést s již vytvořeným webovým archivem, který je k dispozici v příloze k práci (soubor `dynrdf.war`), nebo ze zdrojových kódů serveru, které jsou k této práci také přiloženy a které jsou také dostupné online v repozitáři `https://github.com/Chasic/dynrdf`.

### 6.6.1 Systémové požadavky

- Webový server pro webové aplikace v Javě Tomcat [28] nebo Glassfish [26]
- Java 8 [29]
- Maven [30]

### 6.6.2 Instalace přes webový archiv

Instalace serveru přes přiložený webový archiv je nejjednodušší variantou. Pro správný chod celého serveru je potřeba před nasazením nastavit konfiguraci serveru. Celá instalace má následující kroky:

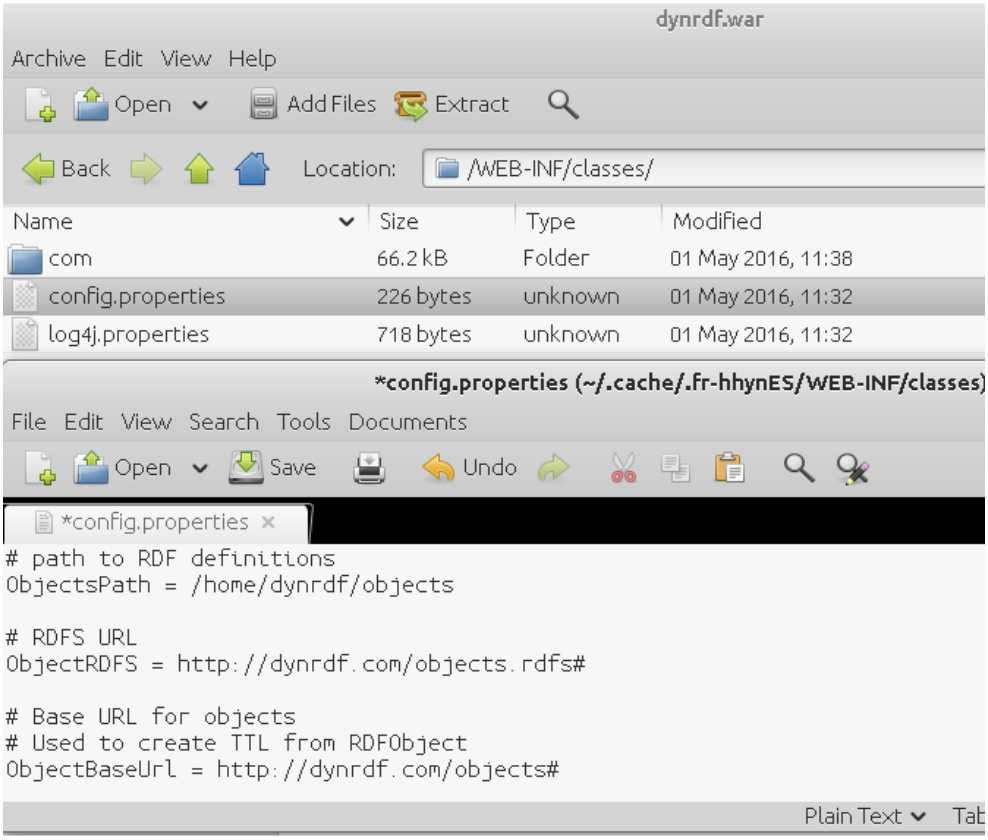
- V souborovém systému založte adresář pro definice objektů a zajistěte práva pro čtení a zápis pro webový server pod kterým aplikace běží

- Nakopírujte do tohoto adresáře ukázkové objekty z adresáře `objects`, který je k dispozici v příloze nebo online v repozitáři na adrese `https://github.com/Chasic/dynrdf/tree/master/objects`
- Otevřte přiložený soubor `dynrdf.war` v libovolném manažeru archivů
- V archivu otevřte soubor `/WEB-INF/classes/config.properties` dle obrázku 6.4 a nastavte zde tyto parametry:
  - `ObjectsPath`
    Cesta k adresáři s definicemi objektů
  - `ObjectBaseUrl`
    URL adresa na seznam objektů dostupný z API: `http://<host>/api/objects`, kde `<host>` je adresa serveru.
  - `ObjectRDFS`
    Cesta k RDF schématu, který definuje atributy definic, jako jsou typy definic, regulární výraz apod. Na této cestě by se měl vyskytovat obsah souboru `dynrdf.rdfs.ttl`. Ten je v základu aplikace dostupný z kořene webové aplikace na adrese `http://<host>/dynrdf.rdfs.ttl`, kde `<host>` je adresa serveru. Pro udržení konzistence v URL adresách je potřeba změnit tuto URL také v tomto souboru. Jedná se o řádek na začátku souboru:
    `@prefix dynrdf: <http://dynrdf.com/dynrdf.rdfs.ttl#>`
    Na tomto řádku změnte URL ve špičatých závorkách na stejnou adresu, na které je tento soubor dostupný.

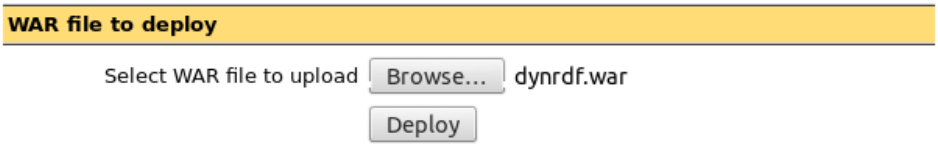
  Parametry `ObjectBaseUrl` a `ObjectRDFS` se používají při ukládání definic do Turtle formátu.
- V archivu otevřte soubor `/WEB-INF/classes/log4j.properties` a nastavte zde parametr `log4j.appender.file.File` na cestu, kam chcete ukládat logy. Můžeté také změnit styl logování, případně změnit i typ logování z `DEBUG` na jiný typ prostředí.
- Uložte upravený webový archiv
- Nahrajte soubor `dynrdf.war` na aplikační server. Na serveru Tomcat pomocí formuláře pro nahrání na obrázku 6.5.

### 6.6.3 Instalace ze zdrojových kódů

Pro instalaci ze zdrojových kódu je potřba mít nainstalovaný program Maven. Instalace má tyto kroky:




Obrázek 6.4: Nastavení parametrů v konfiguračním souboru




Obrázek 6.5: Nahrání webového archivu na server Tomcat (adresa /manager/html) - vyberte war soubor z adresáře a klikněte na deploy

- Nastavte parametry v konfiguračních souborech podobně jako při instalaci z připraveného war souboru. Konfigurační soubory se nacházejí ve složce `src/main/resources`.
- Vytvořte ze zdrojových kódů webový archiv pomocí příkazu `mvn package` v kořenu zdrojových kódu (tam kde je uložen soubor `pom.xml`).
- Webový archiv `dynrdf.war`, který najdete ve složce `target`, která se po provedení příkazu vytvořila, nahrajte na aplikační server stejně jako v případě instalace serveru z war souboru 6.6.2.

## 6.7 Nastavení serveru pro původ dat

Struktura URL adresy tak, jak je popsána v kapitole 3.3, podporuje možnosti pro nastavení webových serverů tak, aby vůči klientům vystupovaly jako servery, na kterých jsou data uložena, ikdyž jsou generované touto aplikací na jiném serveru (na jiné URL adrese). Tímto chováním je docíleno konzistence mezi URL adresou, na které je RDF entita dostupná a IRI této entity ve vygenerovaném souboru.

Pro demonstraci slouží následující příklad. Ten popisuje chování pro už dříve zmiňovanou definici RDF entity logovací služby. Předpokládejme, že webový server pro dynamické generování je v tomto příkladu dostupný na adrese `http://dynrdf.com` a server pro původ dat je dostupný na adrese `http://logservice.com`.

Ze šablony pro RDF entitu 6.6 je důležitý řádek s identifikátorem entity, konkrétně jde o placeholder v `<[@0]>`. Za placeholder `[@0]` se dosazuje celá URL adresa. Za URL adresu, kde má být přístupná požadovaná RDF entita nadále považujme URL `http://logservice.com/data/loggerTurtle/42/2016-05-01/10:00:23/org.dynrdf.Request/msg`.

Listing 6.6: Šablona RDF entity v Turtle formátu

```
@prefix rlog:
<http://persistence.unileipzig.org/nlp2rdf/ontologies/rlog#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<[@0]>
       a rlog:Entry ;
       rlog:level rlog:WARN ;
       rlog:date "[@7]T[@8]Z"^^xsd:dateTime ;
       rlog:className "[@9]";
       rlog:message "[@10]".
```

K nekonzistenci mezi URL adresou, kde je RDF entita dostupná a její IRI dochází v případě, kdy klient přistoupí přímo na URL adresu `http://dynrdf.com/data/?url=http://logservice.com/data/loggerTurtle/4111/2016-05-01/10:00:23/org.dynrdf.Request/msg`. V tomto případě je výsledná RDF entita vygenerována na dané URL, ale její IRI je rovno hodnotě GET parametru `url`.

Toto chování lze opravit pomocí nastavení webového serveru, kde je aplikace `http://logservice.com` spuštěna. Toto nastavení musí zařídit přeposílání požadavků na RDF entitu přes proxy na server pro generování. Například ve webovém serveru Apache [31] se dá toto přeposílání nastavit s použitím modulu `mod_rewrite` [32], jak je uvedeno v ukázce konfigurace 6.7.

Klíčový je zde řádek s `RewriteRule`, který ve volném překladu znamená: „Pro všechny požadavky na data přesměruj tyto požadavky na adresu serveru pro dynamické generování. Jako `url` parametr nastav adresu požadavku.“. Parametrem `[P]` je docíleno toho, že je použit modul `mod_proxy` [33] a nedochází tedy k přímému přesměrování, ale klientovi se jeví výsledná data tak, že pocházejí z dané URL adresy.

Pokud tedy klient přistoupí na URL adresu `http://logservice.com/data/loggerTurtle/42/2016-05-01/10:00:23/org.dynrdf.Request/msg`, získá výslednou RDF entitu a URL adresa i IRI entity se budou shodovat. K přesměrování požadavku na server pro vygenerování entity proběhlo na pozadí a klientovi se výsledná data jeví tak, že pochází se serveru `http://logservice.com`.

Listing 6.7: Část konfigurace webového serveru Apache sloužící k přeposílání požadavků přes proxy

```
RewriteEngine on
RewriteRule /data/(.*)
  http://dynrdf.com/data/?url=http://%{REMOTE_HOST}%{SCRIPT_FILENAME}
  [P]
```

# Závěr

Cílem této práce bylo navrhnout, implementovat a otestovat webový server pro poskytování dynamicky generovaných objektů v RDF formátech. Serveru byly určeny požadavky zadáním této práce a také dalšími v průběhu tvorby této práce. Dalšími požadavky se zabývala analytická část.

Všechny tyto požadavky a cíle byly splněny. Návrhu předcházela analýza účelu aplikace a podobných řešení. Závěrem této analýzy bylo, že neexistuje žádný server, který by poskytoval funkce implementovaného řešení. To dává této práci šanci být velmi využívanou.

Na základě analýzy byl proveden návrh celého systému a následně jeho implementace. Bylo použito několik pokročilých technologií jak z oboru Linked Data, tak i z pohledu implementace.

Testování proběhlo na několika úrovních. Základem byly unit testy, na které navázaly integrační testy v rámci celé aplikace. Aplikace byla také otestována na výkon.

Součástí této práce je také uživatelská dokumentace, která popisuje funkce aplikace z pohledu uživatele. Dokumentace obsahuje také pokyny pro instalaci.

## Využití aplikace a další vývoj

Využití této aplikace je plánováno na portálu LinkedPipes[4]. LinkedPipes je soubor RDF nástrojů, jako jsou LinkedPipes ETL[5] pro převádění dat do RDF a LinkedPipes Visualization[6], nástroj pro vizualizaci a prohlížení RDF dat. Tato aplikace se bude v rámci těchto nástrojů jmenovat LinkedPipes Generator (LP-GEN).

Vzhledem k využití této aplikace se další vývoj dotkne například vzhledu webové aplikace, který bude předělán na materiální design pomocí frameworku

[4] `http://linkedpipes.com/`
[5] `http://etl.linkedpipes.com/`
[6] `http://visualization.linkedpipes.com/`

Angular Material[7]. Tato aplikace díky této změně více zapadne mezi zmiňované nástroje.

Z hlediska implementace je plánováno přidat další možnosti při tvorbě definic objektů, jako je například rozšířené nastavení číslování v šablonách definic, nebo možnost nahrání vlastního souboru dat pro typy definic pracující se SPARQL dotazy.

[7] `https://material.angularjs.org/latest/`

# 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] Apache Software Foundation: *Apache Jena [online]*. [cit. 2016-05-08]. Dostupné z: https://jena.apache.org/

[13] Sesame: *Sesame [online]*. [cit. 2016-05-08]. Dostupné z: http://rdf4j.org/

[14] 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

[15] Oracle Corporation: *Jersey [online]*. [cit. 2016-05-13]. Dostupné z: https://jersey.java.net/

[16] Google: *A Java serialization/deserialization library that can convert Java Objects into JSON and back. [online]*. [cit. 2016-05-13]. Dostupné z: https://github.com/google/gson

[17] Apache Software Foundation: *Apache log4j 1.2 [online]*. [cit. 2016-05-13]. Dostupné z: https://logging.apache.org/log4j/1.2/

[18] Lightbend Inc.: *typesafehub/config: Configuration library for JVM languages [online]*. [cit. 2016-05-13]. Dostupné z: https://github.com/typesafehub/config

[19] JUnit: *JUnit - About [online]*. [cit. 2016-05-13]. Dostupné z: http://junit.org/junit4/

[20] Oracle Corporation: *Chapter 7. Jersey Test Framework [online]*. [cit. 2016-05-13]. Dostupné z: https://jersey.java.net/documentation/1.19.1/test-framework.html

[21] AngularJS, Google: *AngularJS — Superheroic JavaScript MVW Framework [online]*. [cit. 2016-05-13]. Dostupné z: https://angularjs.org/

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

[23] W3C: *Content-Type fields in MIME [online]*. [cit. 2016-05-13]. Dostupné z: https://www.w3.org/Protocols/rfc1341/4_Content-Type.html

[24] Bootstrap community: *Bootstrap [online]*. [cit. 2016-05-13]. Dostupné z: http://getbootstrap.com/

[25] Apache Software Foundation: *Apache JMeter [online]*. [cit. 2016-05-13]. Dostupné z: http://jmeter.apache.org/

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

[27] W3C: *RDF - Semantic Web Standards [online]*. [cit. 2016-05-08]. Dostupné z: https://www.w3.org/RDF/

[28] The Apache Software Foundation: *Apache Tomcat [online]*. [cit. 2016-05-12]. Dostupné z: http://tomcat.apache.org/

[29] The Apache Software Foundation: *Maven - Welcome to Apache Maven [online]*. [cit. 2016-05-08]. Dostupné z: https://maven.apache.org/

[30] Oracle: *Java 8 Central [online]*. [cit. 2016-05-08]. Dostupné z: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

[31] Apache Software Foundation: *The Apache HTTP Server Project [online]*. [cit. 2016-05-13]. Dostupné z: https://httpd.apache.org/

[32] Apache Software Foundation: *mod_rewrite - Apache HTTP Server [online]*. [cit. 2016-05-13]. Dostupné z: http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html

[33] Apache Software Foundation: *mod_proxy - Apache HTTP Server [online]*. [cit. 2016-05-13]. Dostupné z: https://httpd.apache.org/docs/current/mod/mod_proxy.html

Příloha **A**

# Seznam použitých zkratek

**RDF** Resource description framework

**URI** Uniform Resource Identifier

**IRI** Internationalized Resource Identifier

**URL** Uniform Resource Locator

**HTTP** Hypertext Transfer Protocol

**WWW** World Wide Web

**HTML** HyperText Markup Language

**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

**VPS** Virtual private server

Příloha **B**

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `install.txt` ..... instalační pokyny
- `dynrdf.war` ..... war soubor pro deploy na server
- `objects` ..... adresář s ukázkovými definicemi objektů
- `testObjects` ..... adresář s definicemi objektů pro testování
- `img` ..... adresář s obrázky webové aplikace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF