Sem vložte zadání Vaší práce.

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

Bakalářská práce

YouTube konektor pro projekt NARRA

Petr Kubín

Vedoucí práce: Ing. Petr Pulc

5. května 2015

Poděkování

Děkuji vedoucímu práce Ing. Petru Pulci za jeho odborné vedení a své rodině za podporu během celého mého 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 5. května 2015 .....

Odkaz na tuto práci

Kubín, Petr. *YouTube konektor pro projekt NARRA*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

YouTube, snad nejoblíbenější server pro sledování a sdílení videí po celém světě. V mé bakalářské práci se budu zabývat propojením videí na serveru YouTube a jejich popisků, neboli metadat. Takto spárované video a jeho metadata umožní vyhledávačům v projektu OpenNarrative lepší a rychlejší nabízení relevantního obsahu pro další práci s videem, jako například střih. Celý projekt je napsán v jazyce Ruby s využitím již dostupných funkcí z YouTube API.

Klíčová slova YouTube API, Ruby, NARRA, OpenNarrative, popis videa metadaty, formát videa H.264

Abstract

YouTube, perhaps the most popular server to watch and share videos worldwide. In my bachelor thesis I examine the interconnection between videos on YouTube and their labels, or metadata. Thus paired video and its metadata allows search engines in project OpenNarrative better and faster offering relevant content for further work in video, such as editing. The project is written in Ruby using the already available features of YouTube API.

Keywords YouTube API, Ruby, NARRA, OpenNarrative, metadata description of the video, H.264 video format

Obsah

Seznam obrázků

Seznam tabulek

Úvod

YouTube je největší webová stránka pro sdílení videí se sídlem v San Bruno, Kalifornie. Byla založena v únoru 2005 a odkoupena společností google v listopadu následujícího roku. YouTube podporuje WebM a H.264. WebM byl formát pro web společností google. H.264 se používá díky dobré kvalitě komprese a je firemním standartem s hardwarovou podporou. Obsahem YouTube videí může být videoklip, televizní pořad, naučná videa a další. Většina obsahu je tvořena videi od jednotlivců, je zde také možné narazit na soubory od společností například Vevo, nebo BBC.

NARRA, projekt centra audiovizuálních studií FAMU, je opensourcová webová služba umožňující společnou práci s audiovizuálním materiálem. Jádro projektu je tvořeno multimedíální databází obsahující metadata jednotlivých médií a informace popisující vazby mezi nimi. Mým cílem je rozšíření projektu NARRA o možnost práce s multimediální službu YouTube pro možnost zpracování médií, neboť YouTube je nejjednodušší způsob uložení médií pro umělce z FAMU, kteří budou software NARRA využívat.

Hlavním úkolem je vytvořit konektor umožňující práci s YouTube videi a provázání těchto videí metadaty, které mi poskytne YouTube API. Takto zpracované video pro stažení projektem NARRA bude přístupné pro další střih či úpravy. To povede k usnadnění práce editorů, kteří neznají dodaná data úplně do detailu a potřebují najít ta nejlepší videa pro střih.

Na začátku práce se seznámíte s historií vzniku projektu NARRA, jeho hlavní myšlenkou a použitými technologiemi. V další kapitole zpracuji NoSQL databázi MongoDB, která je součástí projektu NARRA. Dále se teorie posune k YouTube API, jeho možnostem, omezením a technologiím, které mi umožní mou práci dokončit. Uvedu i několik příkladů použití YouTube API spolu s vysvětlením.

V další kapitole se seznámíte s teorií testování a testovacím nástrojem RSpec, který je určený pro jazyk Ruby. Poté se teorie přesune k metadatům a DublinCore, kde vysvětlím historii a důležitost popisu elektronického materiálu pomocí metadat. Z teorie tak zbývají pouze dvě kapitoly o vyrovnávací

paměti médií, která je potřeba pro uložení videa na server a kodek VP8.

V praktické části se dozvíte více příkladů a krátký návod jak pracovat s YouTube API, validaci YouTube URL adresy pomocí regulárního výrazu, identifikaci a uskutečnění přesměrování v rámci HTTP. Dále se dočtete jak dostat z YouTube API informace o videu a jak je uložit ve formě metadat. Na konci práce jsem detailně rozebral teorii a praxi v testování softwaru, který bude součástí většího projektu.

Kapitola 1

Cíl práce

Cílem práce je vytvořit rozšíření pro systém NARRA, které umožní import médií z portálu pro sdílení videí YouTube a jejich popis metadaty v souladu s DublinCore. Toto rozšíření umožní zpřístupnění videa v systému pro Open Narrative a dokumentaristé spolu s dalšími tvůrci budou moci využívat YouTube jako své osobní primární úložiště, nebo i zdroj cizího materiálu.

Protože systém NARRA potřebuje přístup k multimediálním souborům pro vytvoření náhledu, je naším úkolem také zpřístupnit videosoubor pomocí URL (mezipaměť) pro zpracování systémem. O samotné vytvoření náhledů a uložení do databáze MongoDB se stará samotný systém, ale základní principy budou vysvětleny i v této práci. V projektu NARRA budou videa dále zpracovávána. Je proto na místě umět nabídnout další videa pro střih, či úpravy, což usnadní práci editorům, kteří neznají dodaná data úplně do detailu a tak sami netuší, jestli se pro další střih nenabízí něco lepšího.

Kapitola 2

Analýza a návrh

2.1 Projekt Narra

Narra je projekt s volně dostupným zdrojovým kódem, který se zabývá anotací a propojením audiovizuálních médií a textu. Podobně jako YouTube má dostupné API a po dokončení bude soužit umělcům, filmařům a dalším, kteří chtějí vytvářet otevřený příběh (Open Narrative), nebo editovat videa. Dokumentaristé mohou tento projekt využít pro rozsáhlejší díla, díky nabídnutému relevantnímu obsahu s metadaty. Projekt zastřešuje FAMU CAS a celý vývojářský tým tvoří 5 lidí, včetně studentů dokončujících Bakalářské a Magisterské studium.

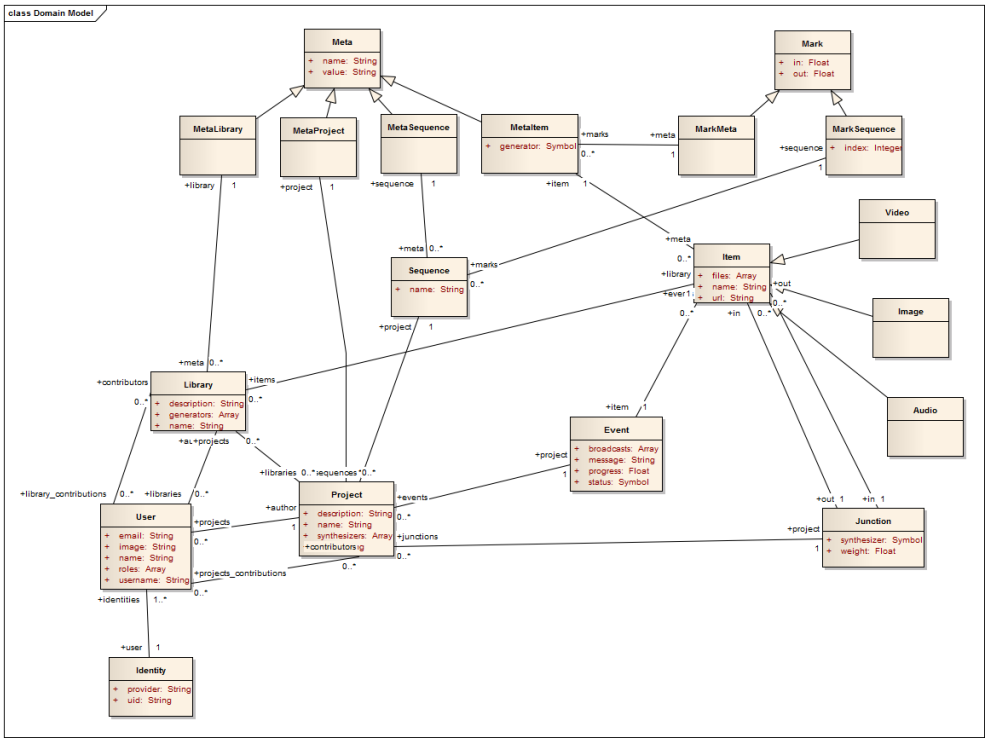
S první myšlenkou projektu Narra přišel v roce 2002 - 2003 Eric Rosenzveig a Willy LeMaitre spolu s dalšími mediálními umělci a programátory. Dílo bylo rozdělené na tři části. První „playListNetWork“ byl opensource software vyvinutý na základě konzultací s umělci o audiovizuálním obsahu a rozhraním pro vizualizaci. Umožňoval práci více uživatelů na různých místech a pomocí textových poznámek upravovat popisky skladeb a videí. Druhý „disPlayList“ bylo veřejně přístupné rozhraní pro streamování medií z playListNetWork. Jednalo se o webovou aplikaci, která vizualizovala výsledná videa do grafu, ze kterého šel pomocí klíčových slov tvořit další celek. „Ressemblage“, neboli poslední část byl výsledkem práce umělců používajících novou technologii práce s médii.

V projekt Open Narrative se zapříčinil nejvíce umělec Eric Rosenzveig a redaktor Tomáš Dobruška na FAMU v letech 2010 - 2015. Díky penězům z grantu mohou pokračovat ve vývoji spolu s KSI FIT ČVUT.

2.2 Technologie v projektu Narra

Narra je psaná v jazyce Ruby a poskytuje REST-API pro komunikaci se světem. Další použité technologie jsou Sidekiq, OmniAuth, MongoDB a Rails. Všechny tyto komponenty zajišťují stabilní jádro aplikace, na které je možné

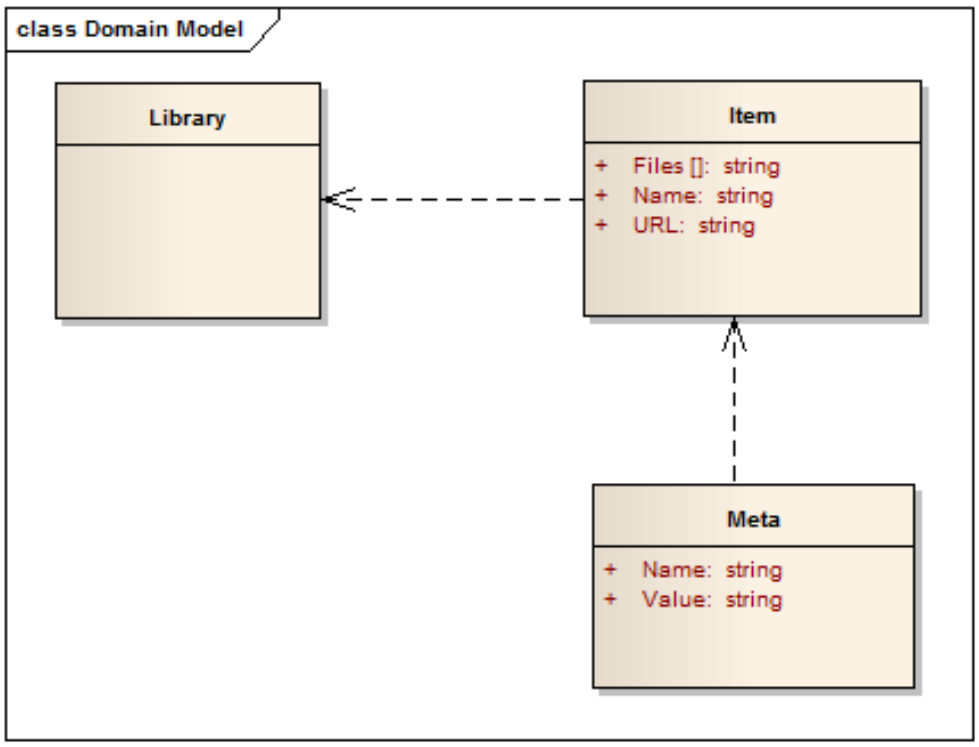
navazovat dalšími balíčky (gem), jako v případě mé bakalářské práce. Pro začátek jsem se musel seznámit s doménovým modelem celé aplikace.



Obrázek 2.1: Doménový model celého projektu Narra

User reprezentuje přihlášeného uživatele, který chce používat aplikaci. Každý uživatel může mít vazbu na projekt a knihovnu. Entita Item obsahuje informace o jménech souboru, url videa a vlastních stažených souborů. Dále obsahuje vazby na knihovnu ve které je uložen sám Item. Itemy se podle definice v modelu nemohou vyskytovat samostatně, musí být organizovány v knihovnách. V případě mého projektu nebudu vytvářet Item ale jeho potomka Video.

Další vazba je na entity MetaItem, které reprezentují generátor pro metadata. MetaItem je třídní potomek Meta, který přebírá atributy rodiče, jež jsou povinné. Při ukládání položek v Meta musí být vyplněno Meta.name i Meta.value.



Obrázek 2.2: Doménový model mé části Narry

Moje část aplikace se týká především entit Video, MetaItem a Library. Entita Library reprezentuje celou knihovnu videí, se kterou se bude lokálně pracovat. Video (potomek Item) je entita, které budu při vytváření asistovat poskytnutím URL při zadávání. Po zadání budu muset zpracovat obsah odkazu a vrátit metadata a cestu ke stažení videa. Jméno pro konkrétní prvek je reprezentováno řetězcem a url je také řetězec. Pro jednoduché volání budu mít vytvořenou třídu Connector, potomka Narra::SPI, která bude provádět inicializace, validaci, popis metadaty a stažení videa a případných titulků.

Zpracování videa probíhá vytvořením nového prvku (Item) v knihovně. Pro zpracování se použije POST požadavek s parametry v1/items/new (author, admin). Po zpracování dostaneme strukturu nově vytvořeného prvku (Item).

Příklad POST požadavku[1]:

```
POST v1/items/new
url: "http://example.org/00329O-051.mov"
library: "552a328961633276b1000000"
author: "Camera Guy"
metadata: {"description": "Some interesting description"}
```

Příklad odpovědi:


```
{"status":"OK","item":{
  "id":"552a338961633277b1000000",
  "name":"003290-051",
  "url":"http://example.org/003290-051.mov",
  "type":"video",
  "prepared":false,
  "library":{"id":"552a...b1000000","name":"Example Library"},
  "metadata":[
   {"name":"type","value":"video","generator":"source"},
   {"name":"name","value":"003290-051","generator":"source"},
   {"name":"url","value":"url","generator":"source"},
   {"name":"library","value":"test","generator":"source"},
   {"name":"author","value":"testovaci","generator":"source"},
   {"name":"description","value":"test","generator":"bob"}
  ]
}}
```


V odpovědi jsem musel zkrátit identifikátor knihovny, aby se mi vešel výraz na stránku. Ze stejného důvodu jsem byl nucen zkrátit hodnotu url. Zde je možné nastavit zda bude projekt veřejný. V takovém případě je možné dostat přístup pouze pro čtení za předpokladu, že je knihovna součástí projektu. Pro vyšší práva je nutností být contributor (spolupracovník) daného projektu. Získání výpisu přístupných knihoven se provádí GET požadavkem na zdroj/adresu v1/libraries(author/admin).

2.3 MongoDB

MongoDB[2] je multiplatformní NoSQL databáze. Nepoužívá klasické tabulky, založené na relační struktuře, ale má svá vlastní schémata ve formáto BSON[3]. Objektová databáze umožňuje rychlejší a lehčí integraci dat. Tvůrcem této NoSQL databáze je Americká společnost MongoDB[2] Inc. založená v roce 2007. V roce 2009 přešla společnost MongoDB[2] k open source řešení a stala se součástí významných společností, například BOSH.

2.3.1 Formát BSON

Formát BSON[3] je nejvíce využíván při ukládání a přenosu dat po síti v neobjektové databázi. Je velmi podobný již známému JSON formátu. Ukládání dat probíhá v binární formě, což je efektivnější z hlediska rychlosti i paměťové náročnosti. V některých případech bude BSON zabírat o něco více místa, neboť potřebuje hlavičku, ve které je uložena délka pole s daty. Například formát JSON bude uložený v BSON[3]u takto:

{"hello": "world"}	\x16\x00\x00\x00	// total document size
	\x02	// 0x02 = type String
	hello\x00	// field name
	\x06\x00\x00\x00world\x00	// field value
	\x00	// 0x00 = type EOO

Tabulka 2.1: MongoDB příklad formátu BSON

2.4 YouTube API

YouTube Aplication Programming Interface[4] je nástroj pro vývojáře, který umožňuje snadný přístup ke statistikám, datům na konkrétním YouTube kanálu a i k jednotlivým videím. API je rozdělené na tři hlavní části:

- Players and Player APIs, které umožňují uživatelům sledování videí ve vaší aplikaci a zjištění zpětné vazby od uživatelů.
- Data and Analytics APIs, zběžně popisuje rozhraní pro přístup k funkcím a datům uloženým v databázi YouTube.
- Buttons, Widgets, and Tools slouží k popisu všech nástrojů, které může vývojář použít pro svou aplikaci.

2.5 Data and Analytics API

2.5.1 YouTube Data API (v3)

Pro mou práci potřebuji druhou část Data and Analytics API[4][5]. Tato část rozhraní umožňuje začlenění informací YouTube do vlastní aplikace. Mám v plánu jí použít pro získání metadat, kde HTTP/GET požadavkem získám JSON objekt od YouTube API, s nímž budu nadále pracovat. Pro zahájení práce s API je potřeba si založit Google účet a získat vlastní API klíč. Vytvoření projektu a API klíče popíši podrobněji v praktické části práce.

API vyžaduje v požadavku seznam dílčích zdrojů, které si přejeme získat, aby se zabránilo zbytečnému přenosu dat a nebyla přetěžována síť ani procesor. YouTube API obsahuje omezení denní kvótou, o kterém se dozvíte více v následující kapitole. Tento přístup zajišťuje efektivní práci a využití prostředků, proto jsou požadavky rozděleny na části.

- snippet
- contentDetails
- fileDetails
- player

- processingDetails
- recordingDetails
- statistics
- status
- suggestions
- topicDetails

Díky rozkouskování do částí je možné se dotázat na specifické informace a ušetřit si tak svou denní kvótu. V návaznosti dojde ke snížení latence a aplikace bude rychlejší. Pro správnou funkčnosto požadavků je potřeba mít vlastní API klíč. Projekt NARRA zatím nemá získávání API klíče funkční nicméně je plánované automatické získávání API klíčů pro každého uživatele s Google identitou v systému NARRA.

2.5.2 Omezení

Každá aplikace má určitá omezení paměti[6][7], či časového kvanta, které uživateli přidělí. YouTube omezuje pomocí kvót počet požadavků, ve kterých měří využití výpočetního výkonu pro jednotlivé uživatele. Podporovány jsou čtyři typy operací.

- list
- insert
- update
- delete

Operace list vrátí GET požadavek spolu žádným, či více výsledky. Insert vytvoří pomocí pažadavku POST nový prostředek. Update změní již existující prostředek a nahradí ho novým. Poslední delete vymaže existující prostředek, který jsme specifikovali. Operace insert, update a delete vyžadují autorizaci uživatele, nejčastěji pomocí jeho vlastního API[4] klíče. Operace list funguje i v případech bez autorizace, tak s autorizací.

Další důvod pro používání kvót je zajištění jisté úrovně efektivity u vývojářů softwaru používajících Data API, a ne vytvářet aplikace, které omezují ostatní a snižují tak kvalitu poskytovaného softwaru. Výše jednotlivých kvót se pro požadavky liší podle náročnosti operací, která se má provést. Jsou zde dva základní faktory, které ovlivňují z většiny cenu požadavku. Při načtení ID z videa zaplatíte 1 jednotku. Operace zápisu stojí přibližně 50 jednotek. Nejdražší je nahrání videa, které se pohybuje okolo 1 600 jednotek za jedno video. To se Vám může zdát jako velmi vysoká cena, ale není tomu tak.

Operace čtení a zápisu nemají přesně stanovenou kvótu, neboť mohou číst a zapisovat odlišné množství částí videí. Pro tento účel YouTube API vytvořilo několik odlišných kategorií, aby umožnilo využít pouze nezbytně malou část kvóty pro požadavek.

Po tomto krátkém úvodu se dostáváme k přidělenému počtu jednotek pro jedno aplikaci. Každá aplikace dostane 50 000 000 jednotek na den, což odpovídá přibližně 1 000 000 operací čtení, kde má každý zdroj dvě části, nebo 50 000 operací zápisu a 450 000 dalších operací čtení, kde má každý zdroj znovu dvě části. Poslední příklad je přibližně 2 000 nahraných videí, 7 000 operací zápisu a 200 000 operací čtení, kde má každý zdroj tři části.

Předchozí odstavec byl jen letmý příklad, kolik si YouTube API účtuje jednotek za své služby. Pro detailnější informace je potřeba nahlédnout do své Google konzole na adrese `https://developers.google.com/youtube/v3/determine_quota_cost`[6]. Zde jde velmi snadno zjistit konkrétní cenu požadavku aplikace pomocí připravené tabulky. Předběžně vypočítané cena u mé aplikace je 9 jednotek na jeden požadavek, což znamená více jak pět milionů zpracovaných videí za jeden den. To je více než dostačující kapacita a proto nemusím omezovat metadata, která v mé aplikaci plánuji z YouTube získávat.

2.6 Testovací nástroj RSpec

2.6.1 Historie

RSpec[8] je samostatný testovací nástroj psaný v Ruby, který vznikl jako experiment Stevena Bakera, Davida Astelse a Aslaka Hellesøe. S vývojem začali v roce 2006 na Ruby on Rails, neboť okolo roku 2006 byly Ruby on Rails nejpoužívanější Ruby aplikace. Pomocí RSpec lze testovat libovolný kód v Ruby. První vydaná verze 1.0 vyšla o rok později a obsahovala funkce, které má RSpec dodnes. Měl ovšem několik ne úplně efektivně naimplementovaných částí a proto musel být přepsán.

Koncem roku 2008 zabudoval Chad Humphries „Micronauta“, který v sobě obsahoval systém metadat a poskytoval mnohem lepší flexibilitu, než RSpec 1.0. V roce 2010 začali David a Chad pracovat na verzi 2. Chtěli celý projekt rozdělit do modulů, které by pak mohli pužívat samostatně a navazovat jedním na druhý. Jako jádro posloužil Micronaut, na který navazovali moduly.

Listopad roku 2012 byl pro vývor RSpecu zlomový. David Asteles se rozhodl od projektu odpojit a věnovat se jiným věcem. Lídrem pro RSpec se stal Myron Marston a pro rspec-rails byl nominován Andy Linderman. Nově vytvořený tým začal pracovat na verzi RSpec 3, která byla spojením a vyčištěním všech předchozích verzí dohromady. Po vydání RSpec 3 odešel Andy Linderman do důchodu. Dodnes se RSpec rozvíjí díky velké komunitě spolupracovníků.

V mé Bakalářské práci budu pracovat s jádrem testovacího nástroje RSpec a očekáváními, neboli `expectations`. V následujících dvou podkapitolách

uvedu postup instalace a ukázky použití daných komponent.

2.6.2 RSpec core

RSpec core je samotné jádro aplikace, na které navazují další moduly. Je nezbytnou částí pro testování. Instalace je složená ze tří příkazů *gem install rspec*; *gem install rspec-core*; *rspec –help* pro nápovědu k nově nainstalovanému softwaru. První příkaz nainstaluje rspec-core, rspec-expectations a rspec-mocks, což je kompletní balíček pro testování. Druhý příkaz nainstaluje pouze rspec-core, který neobsahuje všechny funkčnosti.

Základní struktura popisu testů je velmi podobná hovoru v angličtině. Používají se slova „describe“ a „it“, která mají stejný význam jako v mluveném slově.

Dále můžeme deklarovat vnořené skupiny pomocí klíčových slov `describe`, nebo `context`. Tato klíčová slova nakonec zavolají příslušné metody, což je pro programátora skryto jazykem DSL, kterým popis testů v systému RSpec rozhodně je.

```
RSpec.describe Order do
  context "with no items" do
    it "behaves one way" do
      # ...
    end
  end

  context "with one item" do
    it "behaves another way" do
      # ...
    end
  end
end
```

Další ukázky z rspec-core podrobněji rozeberu v části testování ke konci mé práce.

2.6.3 RSpec expectations

Instalace balíčku expectations je naprosto stejná jako instalace rspec-core, ba i jednodušší. Stačí napsat pouze *gem install rspec*, pro použití s rspec core. V případě testování jinými nástroji, které podporují expectations, je příkaz lehce odlišný *gem install rspec-expectations*.

Použití je velmi intuitivní, neboť je naprosto shodné s projevem v angličtině. Velmi hrubá forma je expect(z čeho).operace(s čím). Souvislý kód poté vypadá například takto:

```
RSpec.describe Order do
  it "sums the prices of the items in its line items" do
    order = Order.new
    order.add_entry(LineItem.new(:item => Item.new(
      :price => Money.new(1.11, :USD)
    )))
    order.add_entry(LineItem.new(:item => Item.new(
      :price => Money.new(2.22, :USD),
      :quantity => 2
    )))
    expect(order.total).to eq(Money.new(5.55, :USD))
  end
end
```

Zde máme metodu Order, ve které vytvoříme dvě položky. První má hodnotu (1.1, :USD) a druhá (2.2, USD), kterou jsme ovšem vytvořili pomocí *:quantity => 2* dvakrát. Proto můžeme otestovat zda součet těchto tří prvků je roven (5.5, :USD). Návratová hodnota testování pomocí expect je true/false. V případě negativního / neočekávaného výsledku oznámí terminál co očekával a na dalším řádku co dostal od programu. Velmi snadno se tedy pozná, kde nastala chyba.

Zabudované komparátory	**Význam**
expect(actual).to eq(exp)	Rovná se (==)
expect(actual).to eql(exp)	Rovná se (eql?)
Identita	
expect(actual).to be(exp) / to equal()	Zda je identické
Porovnání	
expect(actual).to be(exp) > / < / >= / <= expected	Operace porovnání
Regelární výrazy	
expect(actual).to match(/exp/)	Zda výraz odpovídá exp
Třídy	
expect(actual).to be_an_instance_of(exp)	Jestli se aktuální třída == exp
expect(actual).to be_a(exp)	Alias k předchozímu
expect(actual).to be_an(exp)	Alias k předchozímu
expect(actual).to be_a_kind_of(exp)	Alias k předchozímu
Boolovské true / false	

expect(actual).to be_truthy	Projde když actual != nil OR false
expect(actual).to be true	Projde když actual == true
expect(actual).to be_falsy	Projde když actual == nil OR false
expect(actual).to be false	Projde když actual == false
expect(actual).to be_nil	Projde když actual == nil
expect(actual).to_not be_nil	Projde když actual != nil
Očekávání errorů	
expect { ... }.to raise_error	Očekávání, že ... vyvolá error
expect { ... }.to raise_error(ErrorClass)	Očekávání, že ... vyvolá error z ErrorClass
expect { ... }.to raise_error("message")	Očekávání, že ... error bude stejný jako "message"
expect { ... }.to raise_error(ErrorClass, "message")	Kombinace druhé a třetí varianty
Vyhození chyby	
expect { ... }.to throw_symbol	Očekávání vyhození libovolného symbolu
expect { ... }.to throw_symbol(:symbol)	Očekávání vyhození symbolu :symbol
expect { ... }.to throw_symbol(:symbol, 'value')	Vyhození symbolu :symbol s hodnotou 'value'
Členství v kolekci	
expect(actual).to include(expected)	Splněno, když actual obsahuje expected
expect(actual).to start_with(expected)	Actual začíná expected
expect(actual).to end_with(expected)	Actual končí expected

Tabulka 2.2: Různé způsoby testů pomocí RSpec

2.7 Ruby balíčky(gems)

RubyGems[9] je obdoba linuxového manageru balíčků pro programovací jazyk Ruby. Umožňuje snadné stažení a instalaci balíčků do systému. Stejně jako linuxové balíčky hlídá i tento balíčkovací systém závislosti, verze. Na adrese `https://rubygems.org/` se dají nainstalovat publikované balíčky, popřípadě pomocí aplikace bundler, lze získat i nepublikovaný software z repozitáře na GitHubu.

2.8 DublinCore

DublinCore[10] je doporučení pro označení metadat, jehož cílem je umožnit rychlé a snadné vyhledávání v elektronických zdrojích. Původně byl vytvořen jako popis metadat webových stránek, postupně zaujal vyšší instituce a experty z různých odvětví, například muzeí, knihoven a dalších komerčních ogranizací. Vedení sídlí ve státě Ohio v Severní Americe.

2.8.1 Historie DublinCore

První setkání tvůrců DublinCore[10] bylo v březnu 1995. Jejich cílem bylo popsat elektronická data, na základě sémantických pravidel a byla zde uvedena problematika vyhledávání v elektronických dokumentech.

Druhý seminář se konal o rok později v dubnu 1996. Tvůrce tentokrát hostilo město Warwick ve Velké Británii. Po přiblížení problematiky mezinárodní komunitě se zaměřili na problém syntaxe a sémantiky, kterou by byly schopné efektivně zpracovat webové aplikace. Rychlé zavedení technologie DublinCore do webových aplikací vedlo k rychlému rozšíření této metodiky do světa. Na tomto setkání byl vytvořen základ architektury metadat: Warwick Framework.

V září 1996 byl další seminář přesunut do Spojených států amerických do města Dublin. Hostil převážně experty na grafiku, kteří spolu s tvůrci DublinCore diskutovali o spojení mezi vizuální a textovou částí metadat. Měli za cíl spojit požadavky DublinCore a Warwick Frameworku.

Další krok ve vývoji se odehrál v roce 1997 ve městě Canbery v Austrálii. Hlavní myšlenou bylo učinit popis více minimalistický a lépe ho strukturovat. Šlo o zjednodušení a upravení popisu pro následná další rozšíření, neboť každý jistě ví, že na hliněných nohách se kvalitní barák postavit nedá. Proto celé specifikaci dali stejnou strukturu a učinili jí minimalistickou.

Další semináře se konaly v říjnu 1997 v Helsinkách, kde byl kladen důraz na datum, působnost a vztah. Ve Washingtonu v roce 1998 bylo svolané setkání, pro sjednocení různých implementací DublinCore. Další seminář byl rekordní v počtu odborníků. Sešlo se jich 120 z 27 zemí. Další semináře už nastíním pouze pomocí tabulky.

Rok	Město	Země
2000	Otawa	Kanada
2001	Tokyo	Japonsko
2002	Florencie	Itálie
2003	Seatle	Washington, USA
2004	Sanghai	Čína
2005	Madrid	Španělsko
2006	Manzanillo, Colima	Mexico
2007	Singapur	Singapur
2008	Berlin	Německo
2009	Soul	Jižní Korea
2010	Pittsburgh	Pennsylvania, USA
2011	Haag	Nizozemí
2012	Kuching, Sarawak	Malaysie
2013	Lisabon	Portugalsko
2014	Austin	Texas, USA

Tabulka 2.3: Schůzky DublinCore mezi lety 2000 - 2014

2.8.2 Struktura DublinCore

Pro mou bakalářskou práci jsem zvolil popis sekce 3 z `http://dublincore.org/documents/dcmi-terms/`, kterou jsem ještě upravil, abych využil všechna metadata co mi YouTube nabízí a zároveň se zbavil duplicit, které se v této sekci nacházejí. Veškeré popisy jsou dostupné na `http://dublincore.org/documents/dcmi-terms/`.

Struktura objektu s daty o YouTube videu se lehce lišila od stanovené struktury z DublinCore[10]. V následující tabulce je náhled na originální změní požadavků, které jsem musel upravit. Více o této úpravě je napsáno v části Realizace, kapitola Popis metadaty.

Popisek	Definice	Komentář
Přispěvatel (contributor)	Subjekt zodpovědný za zdroj příspěvku.	Například jméno osoby, organizace, nebo služby.
Krytí (coverage)	Prostorová a časová použitelnost zdroje.	Například označení místa geografickými souřadnicemi, nebo lhůta či časové období.
Tvůrce (creator)	Subjekt zodpovědný za zpřístupnění zdroje.	Například jméno osoby, organizace, nebo služby.
Datum (date)	Doba spojená s událostí v životním cyklu zdroje.	Vyjádření časové informace.
Popis (describtion)	Popis dané entity.	Popis může být textový nebo grafický.
Formát (format)	Formát souboru, fyzický nosič, nebo rozměry zdroje.	Například délka trvání videa.
Identifikátor (identifier)	Primární klíč zdroje.	Parametr v HTTP GET požadavku na stránku s videem.
Jazyk, řeč	Jazyk zdroje.	YouTube API neposkytuje přímý přístup k jazyku, proto ho nebudu do mých metadat používat.
Vydavatel	Osoba která vydala dílo.	Zde by mohl být vydavatel YouTube, který se dle mého názoru spíše platforma či služba, proto v mých metadatech autor nebude.
Vztah (relation)	Související zdroje.	Tuto informaci YouTube API neposkytuje.
Práva (rights)	Informace o právech obsahu.	Tento atribut je v YouTubeAPI popsán dvěmi entitami. LicensedContent a License.
Zdroj (source)	Přibuzný zdroj od kterého je odvozen popsaný zdroj.	Tento atribut YouTube API neposkytuje.
Předmět (subject)	Téma zdroje.	Toto je u videa reprezentováno číslem kategorie.
Název (title)	Jméno dané zdroji.	Atribut title, který vracím samostatně kvůli požadavkům na rozraní konektoru.
Typ (type)	Povaha nebo žánr zdroje	Toto je vždy video.

Tabulka 2.4: Ujednocená struktura metadat

2.9 Vyrovnávací paměť médií

2.9.1 Obecný popis

Vyrovnávací paměť[11] slouží ke zrychlení systému pomocí „nakešování“ informací, které by byly čteny například ze vzdáleného zdroje. Zde je zrychlení docíleno díky různým přístupovým časům mezi vzdáleným diskem a lokálním diskem. Přístup na vzdálený disk disk trvá řádově vteřiny, zatímco čtení z lokálního disku trvá o poznání kratší dobu. Velmi rychlou matematikou se dá vypočítat, že je cashe několik řádů rychlejší v porovnání se vzdáleným úložištěm, proto je potřeba uložit často používané informace přechodně do vyrovnávací paměti. Mezi nevýhody pamětí cashe patří jejich velikost. Zatímco vzdálený disk, či claudové úložiště je řádově TB (v případě YouTube nemá šanci jeho velikost smrtelník odhadnout), lokálního úložiště je řádově GB ~ TB a je cenově nákladnější.

V projektu NARRA je také potřeba pracovat s vyrovnávací pamětí médií, neboť je zde vytěžován server, datové linky a další komponenty projektu. Řešení spočívá ve vytvoření jednorázového média v náhledové kvalitě. Náhledová kvalita postačuje pro zjištění obsahu videa a zároveň rychlou práci pro střih, zatímco po dokončení práce se již vyšlou z projektu příslušné požadavky a zajistí celé sestříhané video v nejvyšší možné kvalitě.

Uložení náhledového videa má i další důvody: uživateli je možné poskytnout takové zdroje, ke kterým nemá přímo přístup a dále se video v náhledové kvalitě neztratí v případě, že YouTube původní video zablokuje, případně bude odstraněno původním vlastníkem. Tento jev se děje velmi často a může být spojen s porušováním autorských práv. V případě YouTube je extrémně důležité mít multimédium ve vyrovnávací paměti, neboť není dostupné jako soubor a tak by bylo třeba ho při každém požadavku znova stahovat a zpracovávat.

2.9.2 Realizace v projektu NARRA

Výsledná realizace spočívá v těchto krocích:

- Můj konektor poskytne informaci, kde se nachází soubor s multimediálním obsahem.
- Pracovní server NARRA navštíví tuto adresu (hostovanou na serveru `http://neptun.avc-cvut.cz/`), čímž dojde ke stažení videa, uložení na cestu dotupnou přes http a zaslání hlavičky 303 s lokací souboru.
- Pracovní server tedy náseduje přesměrování.
- Pracovní server přepočítá videosoubor do všech formátů potřebných v NARRA (WebM ve vysoké a nízké kvalitě; zvukový soubor ve formátu OGG Vorbis). Odkazy na tyto soubory sám předá do databáze NARRA.

Uvnitř NARRY se při vytvoření entity Item vytáhnou z konektoru, který umí danou URL obsloužit (existují uvnitř zabudované konektory pro multimediální soubory dostupné přímo přes HTTP), všechny potřebné informace včetně adresy pro stažení fyzického multimediálního souboru. Jako poslední krok po uložení Itemu dojde ke spuštění zpracování (narra-core/lib/narra/core/items.rb:88). To znamená, že se do fronty úkolů v systému SideKick zařadí úloha překódování videa do požadovaných formátů (definováno v nastavení instance, viz dotaz na API v1/settings, 1.5.2 v dokumentaci API).

Dále jsem musel pro moje účely vytvořit způsob jak stáhnout a dočasně poskytnout YouTube video jako soubor. Vše je postaveno na serveru nginx. Tento kód je pouze návrhem kódu, který poběží na serveru. Podílel jsem se pouze na myšlence tohoto kódu, nikoli na skutečné implementaci. Implementací se zabývali programátoři z projektu NARRA. Z mého konektoru jsem pouze musel zajistit, že bude identifikátor videa v pořádku.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import web, subprocess, os

urls = ("/.*", "youtube")
app = web.application(urls, globals())

class youtube:
  def GET(self):
    data = web.input(id="0")
    video = data.id
    if video == "0":
       raise web.notfound()

    filename = subprocess.check_output(['youtube-dl',
        '--get-filename','-o','"%(id)s.%(ext)s"',video])
    filename = filename.strip('"\n')

    os.system("youtube-dl -o /data/%s %s >/dev/null" % (filename, video))

    raise web.seeother('/'+filename)

if __name__ == "__main__":
  web.wsgi.runwsgi = lambda func, addr=None: web.wsgi.runfcgi(func, addr)
  app.run()
```

2.10 Kodek VP8 a ukládání videí v NARRA

Pro správné pochopení významu kodeku VP8[12] je potřeba trocha teorie k WebM[13]. WebM je otevřený formát multimediálních souborů používaný na webu. WebM soubory se skládají z obrazových toků komprimovaných právě kodekem VP8, nebo VP9. V projektu NARRA je použita komprimace pomocí VP8. V aktuálním nastavení se videa počítají do formátu WebM (video v kodeku VP8, audio ve Vorbis) ve dvou kvalitách: 720p s bitrate 1Mbps a 180p s bitrate 300kbps. Navíc je počítán čistě zvukový náhled ve Vorbis (kontejner OGG) a pět náhledů v rozlišení 350x250 ve formátu PNG.

Pro uložení videa se používá kontejner (například WebM, MP4, AVI, MOV, ...), tyto formáty definují vnitřní strukturu a složení jednotlivých datových proudů do výsledného souboru. Tento datový proud je potřeba zkomprimovat. K účelu komprimace slouží právě kodek (MPEG, VP8, H.264, ...). Komprimace slouží pro zmenšení datového toku pro záznam videa.

2.10.1 VP8

Jak již bylo zmíněno VP8 je formát pro kompresi dat vlastněný společností Google. Je založen na knihovně `libvpx`, která jediná umí zakódovat VP8 video stream. Dekódování probíhá také pomocí google knihovny `libvpx`.

Kapitola 3

Realizace

3.1 Řešení spolupráce s YouTube API

3.1.1 YouTube Data API (v3)

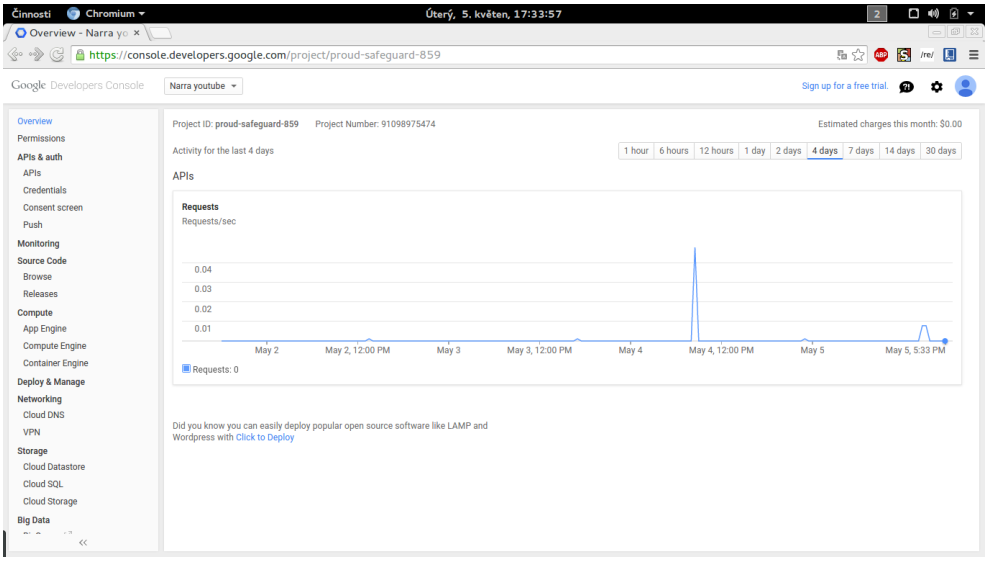
API v3[5] umožňuje začlenění informací YouTube do vlastní aplikace. Proto ho používám pro získání metadat. Nejprve jsem si musel vytvořit google účet a zaregistrovat aplikaci. Pro vytvoření google účtu a zaregistrování slouží `https://console.developers.google.com/project`[7]. Každý takto vytvořený projekt má u sebe statistiky s počtem dotazů, počtem chyb, identifikačním řetězcem a názvem.

Po kliknutí na název mého projektu je možné se dozvědět podrobnější informace a změnit konfiguraci projektu. Základní náhled mi poskytuje graf s počtem požadavků, kde vidím jak moc vytěžuji YouTube API[5]. Dále je zde potřeba nechat si vygenerovat unikátní API klíč, pomocí kterého získám přístup k API a tak mohu získávat metadata. Následující obrázek slouží jako ukázka pro úvodní stránku projektu.

3.2 Třída connector

3.2.1 Založení aplikace

Před založením aplikace jsem několikrát navštívil Michala Mocňáka, který se podílí na vývoji projektu NARRA a po vymyšlení mé části aplikace jsem požádal vedoucího práce, aby mi daný projekt předpřipravil, neboť na `fork` projektu z NARRA jsem neměl dostatečná práva. Pro lehčí kontrolu mého postupu a možnost verzování jsem zvolil službu GitHub `www.github.com`[14]. Po předpřipravení projektu podle mého návrhu jsem si na GitHubu vytvořil účet, přidal vlastní SSH klíč a mohl jsem si celou aplikaci k sobě natáhnout a začít programovat.



Obrázek 3.1: Ukázka projektu na google konzoli

3.2.2 Validace a inicializace

Vlastní implementace je napsaná v jazyce Ruby. Pro můj vývoj jsem si vybral vývojové prostředí RubyMine. Pro vyřešení metadat jsem si vytvořil jednu třídu, kterou jsem napojil na `narra-core`. Dále jsem potřeboval knihovny `Net/HTTP` a `JSON` pro snazší práci.

```
require 'narra/core'
require 'net/http'
require 'json'

module Narra
  module Youtube
    class Connector < Narra::SPI::Connector
```

Tímto kusem kódu jsem vytvořil nový modul, který je potomkem Narra::SPI::Connector. Další věc jsem musel řešit validaci url. V případě nevalidní url mi stačilo vrátit false, při úspěchu jsem vracel true. Booleovskou hodnotu jsem použil, neboť musím umět říci konektoru zda umím či neumím danou url obsloužit. Nejdřívě jsem zkoušel do methody validace URL zakomponovat metodu match. Ta ovšem vracela řetězec, který se shodoval, nebo hodnotu `nil` místo booleovké hodnoty a proto jsem ji musel nahradit `=~`. Takto zkonstruovaný výraz by ovšem nefungoval úplně dokonale, neboť při funkční url by vrátil 0, což značí pozici od které se výrazy shodují. Stačilo výraz lehce

poupravit do tvaru !!(url =~ RegExp) a už jsem měl požadovaný booleovský výraz s hodnotami true/false.

```
!!(url =~ /^(?:http:\/\/|https:\/\/)?(www\.)?(youtu\.be\/|youtube
\.com\/(?:embed\/|v\/|watch\?v=|watch\?.+&v=))((\w|-){6,11})(\S*)
?$/
```

3.2.2.1 Regulární výraz pro validaci

Regulární výraz[15] je matematický formalismus pro popis slov/vět jazyků**Připomínka:** Přímo definice od Ing. Jana Trdličky. Jedná se tedy o způsob jak formálně popsat určité slovo, případně větu pomocí formálního vyjádření speciálními symboly a skupinami znaků. V Ruby[16] je popis regulárním výrazem uvozen / na začátku a / na konci. Pomocí dvou lomítek řekneme překladači, že zápis uvnitř lomítek má považovat za regulární výraz.

V předchozím regulárním výrazu jsem použil standardní konstrukce, až na jednu výjimku, která není až tak obvyklá. Jedná se o ?:(výraz), což znamená že bude splněno při žádném nebo jednom výskytu (výrazu). Tento způsob zápisu umožňuje kvantifikovat výskyt výrazu uvozené závorkami. Například url `https://edux.fit.cvut.cz`, ve které mě nezajímá protokol, vytáhnu regulárním výrazem `(?:http/https)(edux.fit.cvut.cz)`. Následující tabulka popisuje ostatní konstrukce regulárních výrazů v jazyce Ruby. Pro otestování regulárních výrazů existuje velmi hezky zpracovaná stránka `http://rubular.com/`[17].

Pro správnou funkčnost ověření, zda je url validní či ne, bylo potřeba vyřešit přesměrování. Například url adresa, která nevypadá ani z části jako validní může vést k videu na serveru YouTube. Příkladem takové adresy je `http://goo.gl/TKMZjS`. Pouhým ověřením přes regulární výraz bych neměl šanci zjistit obsah a validitu odkazu.

3.2.3 Přesměrování

Přesměrování[18] vyžadovalo novou knihovnu `NET/HTTP`, ze které jsem použil její zabudované metody. Při vytváření jsem nastavil horní hranici přesměrování na 20. Dále jsem potřeboval zajistit detekci smyček v přesměrování, k čemuž by docházet nemělo a je to patologický příklad, ale je dobré smyčku zdetekovat co nejdříve. Řešení pomocí pole je pro takto málo prvků efektivnější, neboť HashMapa má větší nároky na inicializaci v porovnání s jednoduchým polem.

Číselné porovnání vypadá takto: vkládání do pole je v konstantním čase ($\mathcal{O}(1)$), kdežto HashMapy je v nejhorším až $\mathcal{O}(n)$. Na druhou stranu vyhledávání v poli je vždy $\mathcal{O}(n)$ kdežto HashMapy je průměrně $\mathcal{O}(nlog(n)$ a nejhůře také $\mathcal{O}(n)$. Protože u výpočtu složitosti jsou u HashMapy vysoké konstanty je paměťově a časově lepší zvolit pole. Takto se mohu podívat zda jsem již nenavštívil nějakou url dvakrát, což by znamenalo zacyklené přesměrování

`[abc]`	Právě jeden znak z množiny: a, b, c.
`[^abc]`	Právě jeden znak z doplňku množiny: a, b, c.
`[a-z]`	Právě jeden znak z rozsahu a-z.
`[a-zA-Z]`	Právě jeden znak z rozsahu a-z nebo A-Z.
`^`	Znak pro začátek řádky.
`$`	Znak pro konec řádky, např `[a-z]+$` bude uspokojen všemi řetězci tvořenými znaky a-z, který se vyskytne alespoň jednou a bude před koncem řádky.
`\A`	Začátek řetězce.
`\z`	Konec řetězce.
`.`	Jakýkoli znak.
`\s`	Jakýkoli znak tvořený bílými znaky.
`\S`	Jakýkoli znak netvořený bílými znaky.
`\d`	Číslice.
`\D`	Vše krom číslice.
`\w`	Jakýkoli znak z množiny (písmeno, číslo, podtržítko).
`\W`	Opak `\w`.
`\b`	Shoda musí nastat na hranici číselného a nečíselného znaku.
`(...)`	Musí se shodovat přesně s výrazem v (), např (http)* značí nula až nekonečno opakování http.
`(ab)|`	Znak a nebo b, a i b mohou být i skupina znaků.
`a?`	Žádný, nebo právě jeden výskyt a.
`a*`	Žádný, nebo více výskytů znaku a.
`a+`	Jeden, nebo více výskytů znaku a.
`a{3}`	Přesně tři výskyty znaku a.
`a{6,}`	Šest a více výskytů znaku a.
`a{3,6}`	Mezi třemi a šesti výskyty znaků a.

Tabulka 3.1: Tabulka skupin symbolů pro regulární výraz

a v mém případě vyhození příslušné vyjímky, neboť vím, že takovouto URL nemohu nikdy zpracovat.

První úskalí knihovny `NET/HTTP`[18] nastalo v okamžiku, kdy url neměla v názvu protokol. V tomto případě nebyla schopna rozpoznat server a celý proces zkolaboval. Řešením bylo přidat k url bez protokolu protokol http, který se v případě potřeby přesměruje na https. Kdybych přidal místo pouhého http rovnou https, mohlo by se stát, že některé stránky nebudou fungovat, neboť není zaručené zpětné přesměrování z šifrovaného protokolu na nešifrovaný. Tohle bude platit hlavně u „zkracovačů“ URL (URL shorteners), které nepotřebují navazovat zabezpečené spojení. Pro vysvětlenou: HTTPS je stále protokol HTTP, nad kterým je spouštěna vrstva SSL šifrování.

Poslední část přesměrování spočívala ve zjištění příslušného kódu, kterým mi stránka odpověděla na `GET` pořadavek. Při kódu 2xx je vše v pořádku a URL lze rovnou vrátit. Stavový kód začínající trojkou je ovšem zajímavějším protože se jedná o přesměrování. V tomto případě musí programátor zjistit, na kterou stránku se dostane a proces opakovat, než dostane kód 2xx, nebo než zjistí, že je v cyklu, dostane chybový kód 4xx/5xx, či vyprší počítadlo přesměrování. Poslední skupina jsou kódy 4xx a 5xx značící chybu klienta nebo chybu serveru. Při chybách 4xx a 5xx vyhodím vyjímku ve které uporozním na výskyt kódu z tohoto rozsahu.

3.2.4 Inicializace

Po zjištění, zda je požadovaná URL adresa validní, bylo potřeba ještě provést inicializaci. Při inicializaci se vytvoří instance objektu `Connector`, která zanikne až v momentě, kdy se všechna data přelijí do struktur databáze. Zde vytáhnu z YouTube API všechny potřebné informace o videu, které budu dále zpracovávat. YouTube API nabízí parametrů, ze kterých si můžu vybrat. Dále je potřeba použít API klíč[4], pomocí něhož YouTube pozná, komu ubrat denní kvótu za požadavek. Toto řešení je dočasné do doby, než v jádře projektu NARRA doimplementují ověření pomocí `OAuth` a každý uživatel bude mít kvůj vlastní vygenerovaný klíč. V následující tabulcetabulce je příklad parametrů pro API.

Parametr	**Význam**
snippet	Zobrazí hlavní informace o videu
contentDetails	Zobrazí detaily obsahu
fileDetails	Zobrazí detaily souboru
player	Zobrazí detaily přehrávače
processingDetails	Zobrazí podrobnosti zpracování
recordingDetails	Zobrazí podrobnosti nahrávání
statistics	Zobrazí statistiky videa
status	Zobrazí status
suggestions	Zobrazí návrhy
topicDetails	Zobrazí detaily o tématu videa

Tabulka 3.2: Části dat YouTube API a jejich význam

Konrétní požavavek z mé aplikace na YouTube API byl na adresu `https://www.googleapis.com/youtube/v3/videos?id=#{@videoid}&key=klíč&part=část`[19], kde `@videoid` je inicializovaná hodnota pro identifikátor videa, klíč je unikátní API klíč programátora, nebo v budoucnosti uživatele systému NARRA a část je seznam požadovaných částí dat oddělených čárkou. Zde je nutné vzít v potaz, že za vytížení YouTube serveru se platí a v jistých případech nemalou částí denní přidělené kvóty[20] jednotek. Detailní výpočet jsem

popsal již v kapitole o YouTube API, zde se o tomto omezení zmiňuji podruhé, neboť jsem nepoužil všech deset parametrů, ale jen čtyři. Cena použitých čtyř částí je 9 jednotek denní kvóty.

Část `snippet`, vypíše o videu většinu informací. `ContentDetails` byl potřeba pro splnění požadavků plynoucích z popisu v DublinCore, `statistics` přidají do obsahu počty sledovaností a `status` zobrazí licenci a informace o sdílení videa. Tyto čtyři položky stačí pro požadovaná metadata zadavatelem a při přidání dalších bych zbytečně omezoval maximální počet vrácených položek díky omezení YouTube API a aplikace by ztrácela na efektivitě.

Při inicializaci je druhý parametr klíč, který slouží k autentizaci v rámci YouTube API. Je nezbytný pro funkčnost, neboť při pořadavku na informace o videu musí mít YouTube možnost snížit konkrétnímu účtu denní kvótu.

3.2.5 Parsování JSON objektu

Nyní již mám k dispozici JSON objekt a můžu se pustit do práce. První pokus o rozparsování proběhl ručně. Vždy jsem si pomocí methody split rozdělil objekt na pole o dvou částech a druhý index jsem rozdělil znovu podle čárky a odřádkování. Pro lepší představu o vizuální stránce kódu sem dám ukázku vytažení obsahu title.

```
pom = @youtube_json_object_snippet.split('"title": "')[1]
@name = pom.split("\",\n")[0]
```

Toto řešení bylo vcelku jednoduché, rozdělení podle ",\n bylo v pořádku, neboť YouTube v popiscích provedlo escapacování těchto znaků a nemohlo dojít k nechtěnnému rozdělení ve špatném místě. Kód ovšem vypadal naprosto hrozně a proto jsem zvolil již hotovou variantu JSON parseru. Pro porovnání ukázka kódu s knihovnou JSON.

```
my_hash = JSON.parse(@youtube)
my_hash["items"][0]["snippet"]["title"]
```

Toto řešení je mnohem přehlednější a další programátor má usnadněné pochopení vnitřní struktury JSON objektu. Po této odbočce se dostáváme zpátky k řešení, kde druhým zmiňovaným způsobem vracím název videa samostatně a ne v komplexní struktuře metadat. Tato alternativa byla zvolena záměrně díky ukládání videí v mateřském projektu. Poskytnutí názvu videa v podobě separátní metody je nutností, protože Item (i potomci) musí mít název, kdežto metadata mít nemusí. Dalším požadavkem mateřského projektu bylo vrácení typu videa :video. Je to z toho důvodu, že většina skriptů, které s médiem dále pracují potřebují vědět, jestli je to video, zvuk nebo obrázek. Tím jsem měl za sebou základní část a mohl jsem pokračovat s metadaty.

3.2.6 Metadata

Pro popis metadaty jsem vycházel ze struktury DublinCore, která ovšem ne úplně 100% odpovídala mé představě ani představě YouTube vývojářů a proto jsem celou kostru musel upravit.

Název	**Obsah**
videoId	Jednoznačný identifikátor videa.
channelId	Jednoznačný identifikátor kanálu, pod kterým je video k dispozici.
channelTitle	Název kanálu, pod kterým je video k dispozici.
publishedAt	Přesný čas vydání a zvěřejnění videa.
description	Popis k videu.
categoryId	Číslo kategorie, do které patří dané video.
liveBroadcastContent	Booleovská hodnota, zda je obsah ve videu vysílaný živě.
viewCount	Počet shlédnutí videa.
likeCount	Počet udělení líbí se.
dislikeCount	Počet udělení nelíbí se.
favouriteCount	Počet přidání do oblíbených.
commentCount	Počet komentářů k videu.
duration	Čas trvání ve formátu `ISO_8601`.
dimension	Zda je video 2d, nebo 3d.
definition	Sd případně hd.
caption	Booleovská hodnota, zda video obsahuje či neobsahuje titulky.
licensedContent	Zda obsah videa podléhá licencování.
regionRestriction	Zda je video zakázané v nějaká zemi.
uploadStatus	Zda je nahrané video již kompletní, či ještě ne.
privacyStatus	Informace o soukromí videa.
license	Kdo vlastní licenci k videu.
embeddable	Zda je možné toto video použít k vložení.
publicStatsViewable	Booleovská hodnota o zobrazitelnosti veřejných statistik.
timestamp	Čas ve formátu utc, kdy byla metadata pořízena.

Tabulka 3.3: Metadata předávaná do NARRA

Jak jste si mohli povšimnout v této tabulce mi chybí titulek videa. Toto řešení je součástí návrhu, kde vracím název videa samostatně pro lepší následné ukládání v mateřském projektu. Dále mám staticky zadefinovaný typ videa, který se nemění.

Pro správné pochopení, jak extrhahovat metadata ze struktury JSON ob-

jektu je potřeba zjistit jak přesně vypadá. Celý objekt je jeden prvek obsahující pole items, ze kterého používám nultý prvek. V této úrovni rozhoduji, zda vyberu data z položky snippet, statistics, contentDetails nebo status. Po zvolení například snippet se dostanu o úroveň hlouběji a mohu vybrat konkrétní položku, například channelId. Stejným způsoben jsou dostupná všechna metadata z JSON objektu, pouze u restrikce zemí vzacím složitější strukturu než string.

Poslední prvek metadat timestamp nenajdu v DublinCore ani v YouTube API, je ovšem důležité ho do dat zařadit, kvůli udržovatelnosti. V mateřské aplikaci bude nejspíš také časový otisk, který není v této chvíli ještě dodělán a proto jsem ho umístil do metadat. Je zde kvůli kontrole, jak staré jsou statistiky u videa a například pro automatizovanou kontrolu metadat starších než dva týdny se tento údaj hodí. Ještě jsem přemýšlel zda bude dobré řešení vložit časový otisk přímo do metadat, nebo raději mimo, ale řešení s otiskem v metadatech vyhrálo. Nejpádnější důvod byl, že při aktualizaci se zavolá pouze metoda metadata a nebude se muset volat žádná další, neboť znovu stahovat video nemá cenu, v případě přidání titulků se zavolá ještě download_subtitles. Také jméno, identifikátor, url a typ videa se nebudou měnit.

Celkem můj gem poskytuje k jednomu videu 24 metadat a odděleně také jméno videa, typ videa. V součtu se jedná o dvacet šest položek, které umožní rychlejší vyhledávání a relevantní obsah pro každého uživatele, který moje rozšíření využije.

Pro správné pochopení, jak extrhahovat metadata ze struktury JSON objektu je potřeba zjistit jak přesně vypadá. Celý objekt je jeden prvek obsahující pole items, ze kterého používám nultý prvek, protože se ptám na konkrétní video. V této úrovni rozhoduji, zda vyberu data z položky snippet, statistics, contentDetails nebo status. Po zvolení například snippet se dostanu o úroveň hlouběji a mohu vybrat konkrétní položku, například channelId. Stejným způsoben jsou dostupná všechna metadata z JSON objektu, pouze u restrikce zemí vzacím složitější strukturu než řetězec (HashMapa s prvky tvořenými polem).

Poslední prvek metadat timestamp nenajdu v DublinCore ani v YouTube API, je ovšem důležité ho do dat zařadit, kvůli uživatelům. V mateřské aplikaci je také časový otisk pro každé metadato, a proto jsem ho umístil do metadat, kde jde o uživatelky viditelnou informaci. Je zde kvůli kontrole, jak staré jsou statistiky u videa a každý uživatel si tak může ověřit jak moc aktuální či neaktuální metadata jsou. V případě, že by nastala chyba v automatizované kontrole stáří metadat je také možné provést namátkovou oční kontrolu. Poslední metodou je metoda pro stažení titulků download_subtitles. Při aktualizování titulků předpokládám, že pokud budou autoři YouTube opravdu používat jako úložiště, tak si nebudou najednou měnit svůj systém značení médií.

Celkem můj gem poskytuje k jednomu videu až 24 metadat a odděleně také jméno videa, typ videa. V součtu se jedná o dvacet šest položek, které

umožní rychlejší vyhledávání v systému NARRA a poskytuje relevantní obsah pro každého uživatele, který moje rozšíření využije.

3.2.7 Dokončení

Na závěr mi zbývalo vrátit YouTube url ve formátu, kde bude pouze video stream bez ostatních elementů, neboť ze standartní url by to bylo moc práce navíc pro jádro aplikace. Pro tento účel souží adresa `#{env}/youtube_dl?id=#{@videoid}`. Proměnná env zde zastupuje proměnnou prostředí, která je platná pro NARRA_YOUTUBE_SERVER a videoid je již dobře známý identifikátor videa.

Poslední kus kódu patřil stažení titulek. Na první pohled se to zdálo jako velmi jednoduchý úkol, ovšem stažení titulek stojí 200 jednotek. Proto je potřeba autentizace API klíčem. Tímto klíčem je potřeba být přihlášen již v jádru aplikace při spuštění a na můj konektor se jen dotázat na stažení titulek. Jelikož je ještě autentizace pomocí OAuth v mateřském projektu nedořešená, předpřipravil jsem můj kus pro stažení titulek pouze pro aktuální funkčnost, která zabezpečí, že po autentizaci pomocí OAuth začne mé stažení titulek fungovat.

Titulky k videu jsou k dispozici z `https://www.googleapis.com/youtube/v3/captions/`, za kterou se opět přiřadí identifikátor videa. Bez autorizace ovšem nahlásí stránka chybový kód: „Login Required“.

3.3 Testování YouTube konektoru

3.3.1 Teorie testování

Pro správné pochopení teorie testování[21] si musíme uvědomit, že pomocí testů prokážeme že software obsahuje chyby při nesplnění testu. Při splnění všech testů nemůžeme dokázat, že je software stoprocentně bez chyb, neboť může existovat chyba, kterou testy neodhalily. Proto je potřeba se důkladně věnovat testování, abychom snížili riziko chyby na nejnižší možnou úroveň.

3.3.2 Jednotkové testy

První druh testů jsou jednotkové testy[21]. Ty se zabývají konkrétní funkčností jednotlivých metod a tříd. Testování probíhá pro jednu konkrétní třídu, obvykle běží krátkou dobu a testování probíhá na lokálním počítači vývojáře. Tyto testy pokrývá v Ruby nástroj RSpec, který jsem popsal na začátku práce. V mých testech jsem se zaměřil na zjištění správné odezvy metod a získání správných výsledků.

3.3.3 Integrační testy

Integrační testování[21] se provádí po dokončení jednotkových testů a slouží k bezchybnému začlenění nového kusu aplikace do stávajícího projektu. Těmito testy se ověřuje nejen integrace komponentů, ale i spolupráce se softwarem případně hardwarem při složitějších projektech, které potřebují specifický hardware nebo software.

Testování probíhá začleněním jedné z komponent a postupném přidávání dalších komponent k celému projektu. Integrační testování je často zanedbáváno díky rozpočtu projektu. Takto nezachycené chyby se ovšem projeví v dalších testech a je proto dobré integrační testy nepřeskočit.

3.3.4 Systémové testy

V pořadí již třetí zkouška funkčnosti projektu se zabývá chováním celého projektu jako celku. Provádí se zde simulace scénářů a kroků, které mohou po nasazení projektu nastat. Je to poslední testování před předáním projektu zákazníkovi a proto obvykle probíhá několikrát. Systémové testování[21] je nezbytné pro výstupní kontrolu projektu.

3.3.5 Akceptační testy

Akceptační testy jsou prováděné zákazníkem spolu se zástupcem firmy na předem dohodnutých scénářích. Jedná se o otestování, zda zadavatel správně pochopil požadavky zákazníka. Odhalení závažných nedostatků v této poslední fázi bývá nejhorším možným scénářem pro vývojáře, který musí následně aplikaci předělávat. Můj gem jsem testoval jednotkovými a integračními testy, neboť systémové a akceptační testování zajišťují programátoři jádra projektu.

3.3.6 Testování třídy Connector

3.3.6.1 Testování URL

První část testu kontroluje vytváření a validaci URL. Před první částí testů jsem si nastavil testovanou URL a zjistil, zda se správně vytvoří objekt dané třídy. Poté jsem zkontroloval tři globální proměnné, zda se nezměnil jejich obsah. Tyto dva testy zajistili primární funkčnost konektoru a mohl jsem zkontrolovat konkrétní URL. Zde jsem hledal různé URL adresy, které nikam nepřesměrují a jsou validní. Dále jsem potřeboval najít nevalidní URL, přičemž stačilo vhodným způsobem modifikovat URL validní.

Poté jsem potřeboval najít URL, která se přesměruje na validní. Zde jsem použil funkci YouTube, kde je možné zvolit sdílení videa, což vytvoří zkrácenou URL, která se přesměruje. Poté jsem ještě použil Google zkracovač, což je také velmi dobrá metoda pro ověření validace a přesměrování.

3.3.6.2 Testování objektu konektoru

Další testy již probíhaly pro určitý objekt konektoru. Zde jsem opět našel několik YouTube videí na kterých jsem testoval příslušná metadata. Pro hezčí zápis testu jsem chtěl zápis metadat převést z formátu *{name:'channelId', value:'#{@channelId}'}* do formátu *{'channelId'=>'#{@channelId}'}*, což mi umožnilo lepší a přehlednější přístup k testování metadat. Url adresy jsem volil pro pokrytí speciálních znaků v popiscích, pro neobvyklá metadata, které jsou u videí jen zřídka a pro pokrytí všech metadat.

Zde byl problém s proměnlivými statistikami u videa, kde například u počtu shlédnutí jsem musel při testování stále upravovat testovanou hodnotu. Proto jsem vizuálně otestoval s JSON objektem shodu v proměnlivých počtech u statistik a poté jsem test zakomentoval. Tento problém bych mohl vyřešit nastavením intervalu, což také není úplně nejlepší neboť bych se musel znovu podívat zda je to číslo opravdu dobře nebo ne.

V nástroji RSpec je v bloku `before` několik možností parametrů v závorkách. V první části testování, kdy jsem měl ještě málo testů, trval celý blok okolo deseti vteřin. V závěrečné fázi práce, kdy jsem otestoval vše co mě napadlo trval test i přes jednu minutu. Takto dlouhé testování se mi nezdálo a proto jsem se důkladně zaměřil na parametrizaci v bloku `before`. Při parametru `:each`, který jsem měl ze začátku se celá inicializace provedla před každým blokem it, proto testy trvaly tak dlouhou dobu. Po změně na `:all` se rázem čas testů dostal stabilně pod deset vteřin. Proto je potřeba dávat velký pozor, zda neděláme v testu stejné věci zbytečně vícekrát.

3.3.6.3 Testy na videích

Pro první testované video jsem zvolil „Zimní montáže - 1. díl“ z kanálu mrk.cz. Zde byl zajímavý prvek ve formě odřádkování v popisu videa. Druhé testované video bylo znovu v duchu rybolovu s zajímavým popiskem, který byl tentokrát prázdný. Dále jsem chtěl na druhém videu otestovat živé vysílání neboť toto video bylo vysíláno živě a přesto je parametr `liveBroadcastContent` roven hodnotě `false`. Hodnota false byla v metadatech o videu správně, neboť po živém odvysílání video změní svůj stav a není vysíláno živě.

Dále jsem vytvořil jeden test pro otestování titulků. Zde jsem potřeboval ověřit, zda mi program správě vyhodí výjimku při neexistujících titulcích. Tato výjimka je velmi opodstatněná, neboť každý download titulků stojí 200 jednotek z YouTube API kvóty a bylo by nežádoucí nechat uživatele několikrát zkusit stáhnout neexistující titulky. Další test spočíval ve správném zpracování dlouhé URL adresy, obsahující další informace za identifikátorem videa, pomocí regulárního výrazu při validaci URL a pro zkoušku vytažení všech dostupných dat.

Páté testované video bylo vytvořeno kvůli možnosti stáhnout validní titulky k videu. Další test vyzkoušel časovou známku u titulků pro čas vytvo-

ření. Blok „should check download video“ měl na starost otestování možnosti stáhnout video, neboli dostat se na URL, která obsahuje pouze video stream. Tento test jsem nemohl provést, neboť potřebuji proměnnou prostředí, kterou dostanu až při dotazu na server. Předposlední test znovu zkusil stažení titulků a ověření, zda bude vyhozena výjimka u neexistujících titulků.

Závěrečný test je zde z důvodu omezení videí v různých zemích. Zde YouTube předává hodnotu `blockedIn` v případě, že je video zakázáno v nějaké zemi. Seznam zemí dále vrátí pomocí pole. V případě, že video není nikde blokováno tento atribut zmizí, protože v databázi nemohou být prázdné hodnoty. Znovu jsem proto musel otestovat správné reagování na omezení.

3.3.6.4 Shrnutí jednotkových testů

V jednotkových testech jsem se snažil pokrýt všechny možné nástrahy, které mi je schopen uživatel nadělit a otestovat, zda můj program reaguje podle předpokladů. Všechny testy proběhly bez chyb a proto jsem snížil pravděpodobnost, že moje rozšíření obsahuje chyby na přijatelnou úroveň. S otestovaným softwarem pomocí jednotkových testů mohu pokračovat v integračních a systémových testech s mateřskou aplikací.

3.3.7 Integrační testy

Integrační testování odhalilo několik ošetření, které je potřeba provádět v mém balíčku a nespoléhat na jádro aplikace. Jednalo se o zařazení metadat, která by měla hodnotu nil, nebo prázdný řetězec. V takovém případě jsem nesměl přidat danou položku s metadaty k výslednému videu, neboť ošetřování obsahu metadat při zpracovávání videa by bylo v rozporu s modelem. Další chybu integrační testy neodhalily a má část aplikace funguje dle očekávání.

Závěr

Cílem mé práce bylo vytvoření konektoru pro YouTube, který bude umět přidat k videu metadata a poskytnout video k dalšímu zpracování v projektu NARRA. Nejdříve jsem si prostudoval technologie a strukturu projektu, neboť jsem na něj navazoval. Po doménovém návrhu mého konektoru jsem si nastudoval možnosti a omezení YouTube API. YouTube má velmi přehledně strukturovaný popis, což mi usnadnilo kontrolu a návrh omezení na můj konektor.

V průběhu práce jsem využil již hotové knihovny pro síťovou komunikaci (NET/HTTP) a pro práci s objekty ve formátu JSON. V žádné z knihoven jsem neodhalil nedostatky, které by znemožnili jejich bezproblémové použití a proto jsem si nemusel psát například vlastní JSON parser. Pro přepis meta informací do systému NARRA jsem se seznámil s formou pro popis metadaty DublinCore a byl jsem překvapen, jak daleko do historie sahají základy této metody.

Dále jsem měl za úkol zpřístupnění videosouboru pro práci se systémem pomocí URL.

Literatura

[1] Pulc, P.: NARRA, [online], Naposledy navštíveno [30. 4. 2015]. Dostupné z: https://github.com/CAS-FAMU/narra/blob/master/doc/narra.pdf

[2] MongoGB, I.: *MongoDB Manual*. [online], Naposledy navštíveno [6. 3. 2015]. Dostupné z: https://www.mongodb.org/

[3] MongoGB, I.: *JSON AND BSON*. [online], Naposledy navštíveno [6. 3. 2015]. Dostupné z: http://www.mongodb.com/json-and-bson

[4] Google Developers: *Getting Started with the YouTube Data API*. [online], Naposledy navštíveno [10. 3. 2015]. Dostupné z: https://developers.google.com/youtube/v3/getting-started

[5] Google Developers: *YouTube Data API (v3)*. [online], Naposledy navštíveno [14. 4. 2015]. Dostupné z: https://developers.google.com/youtube/v3/?hl=cs

[6] Google Developers: *YouTube Data API (v3) - Determining Quota Costs*. [online], Naposledy navštíveno [10. 3. 2015]. Dostupné z: https://developers.google.com/youtube/v3/determine_quota_cost

[7] Google Developers: *Google Developers Console*. [online], Naposledy navštíveno [10. 4. 2015]. Dostupné z: https://console.developers.google.com/project

[8] David, C.: *RSpec. Behaviour Driven Development for Ruby. Making TDD Productive and Fun.* [online], Naposledy navštíveno [14. 3. 2015]. Dostupné z: http://rspec.info/documentation/

[9] Dirk, E.: RubyGems. *LINUX Journal*, May 2006, [online], Naposledy navštíveno [2. 5. 2015]. Dostupné z: http://www.linuxjournal.com/article/8967

[10] Dublin Core Metadata Element Set, Version 1.1: *Dublincore.org*. [online], Naposledy navštíveno [20. 3. 2015]. Dostupné z: http://dublincore.org/documents/dces/

[11] doc. Ing. Hana Kubátová, C.: Testování aplikací, [online], Naposledy navštíveno [5. 5. 2015]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-SAP/_media/lectures/11/sap-11-cache.pdf

[12] Bankoski, J.: VP8 Data Format and Decoding Guide, November 2011, [online], Naposledy navštíveno [4. 5. 2015]. Dostupné z: http://datatracker.ietf.org/doc/rfc6386/

[13] The WebM Project: *Welcome to the WebM Project*. [online], Naposledy navštíveno [4. 5. 2015]. Dostupné z: http://www.webmproject.org/

[14] Tomáš Bartoň, P. P., Jan Vlnas: *Git [MI-RUB Programování v Ruby]*. FIT, ČVUT, [online], Naposledy navštíveno [20. 3. 2015]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-RUB/lectures/start

[15] Trdlička, J.: *Regulární výrazy, grep, awk, sed.* FIT, ČVUT, [online], Naposledy navštíveno [12. 1. 2015]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-PS1/_media/lectures/06/bi-ps1-p06-regexpr-01.pdf

[16] Index of Files, Classes & Methods in Ruby 2.2.2 (Ruby 2.2.2): *Ruby-doc.org*. [online], Naposledy navštíveno [8. 2. 2015]. Dostupné z: http://ruby-doc.org/core-2.2.2/

[17] Michael, L.: *Rubular a Ruby regular expression editor*. [online], Naposledy navštíveno [1. 4. 2015]. Dostupné z: http://rubular.com/

[18] SitePoint: *Looking at Ruby's Net::HTTP Library*. [online], Naposledy navštíveno [20. 4. 2015]. Dostupné z: http://www.sitepoint.com/ruby-net-http-library/

[19] Google Developers: *Videos: list*. [online], Naposledy navštíveno [10. 2. 2015]. Dostupné z: https://developers.google.com/youtube/v3/docs/videos/list

[20] Google Developers: *Google Developers Console*. [online], Naposledy navštíveno [10. 4. 2015]. Dostupné z: https://console.developers.google.com/project/proud-safeguard-859/apiui/apiview/youtube/quotas

[21] Mlejnek, J.: Testování aplikací, [online], Naposledy navštíveno [30. 4. 2015]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-SI1/_media/lectures/09/09.prednaska.pdf

Příloha **A**

Seznam použitých zkratek

FAMU Filmová a televizní fakulta Akademie múzických umění

CAS Centrum audiovizuálních studií

KSI Katedra softwarového inženýrství

FIT Fakulta informačních technologií

ČVUT České vysoké učení technické v Praze

MU Masarykova univerzita

API Programovací rozhraní aplikace

EOO End of object(Konec objektu)

DSL Doménově specifický jazyk

Příloha **B**

Obsah přiloženého CD

- `narra-youtube` ..... složka s implementační částí
  - `lib/narra` ..... konektor NARRA
  - `spec` ..... testování
- `zadani.txt` ..... zadání práce
- `thesis` ..... složka se textovou částí práce
  - `BP_Kubin_Petr_2015.pdf` ..... text práce ve formátu PDF
  - `BP_Kubin_Petr_2015.tex` .... zdrojová forma práce ve formátu LaTeX