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

21. dubna 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 21. dubna 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

Kapitola 1

Cíl práce

Cílem mé práce je doprogramování popisu YouTube videí metadaty v souladu s DublinCore. Jedná se o spárování popisků YouTube videa pro rychlejší a přesnější vyhledávání. Takto doplněné video bude možné stáhnout pomocí URL i s jeho metadaty, uložit ho do neobjektové databáze MongoDB a poté s ním lépe pracovat. Většina vyhledávačů pracuje na základě obsahu a metadat o něm pro zajištění nejlepších a nejrychlejších výsledků. V projektu NARRA budout videa dále zpracovávána. Je proto na místě umět nabídnout další videa pro střih, či úpravy, aby byl výsledný produkt v nejlepší možné kvalitě.

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 také veřejně dostupné API 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, které poskytuje práci pro více jak 30 studentů dokončujících Bakalářské a Magisterské studium.

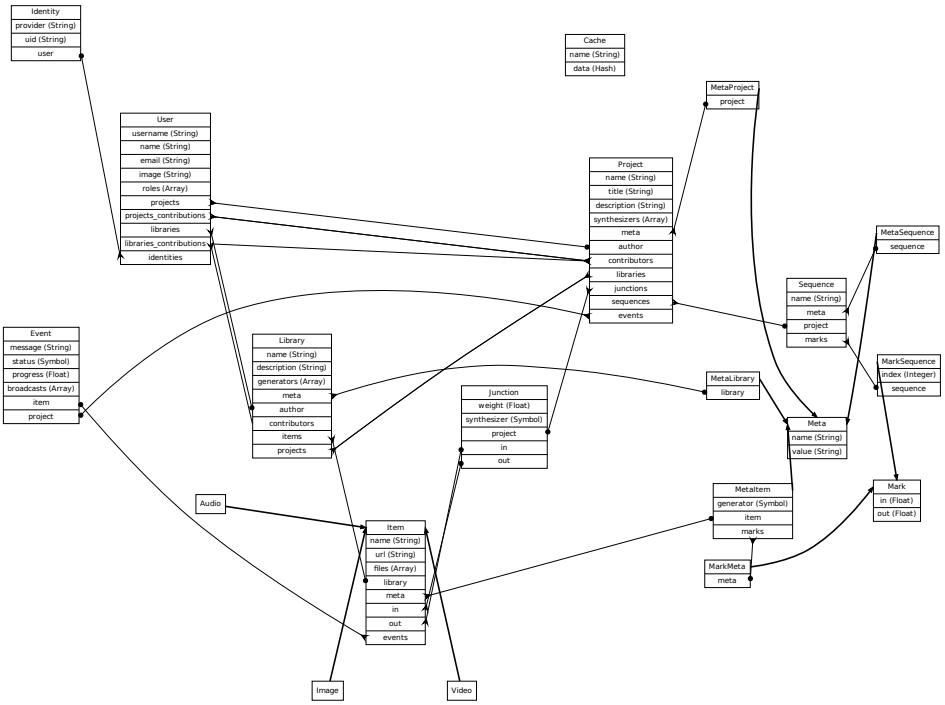
S první myšlenkou projektu Narra přišel v roce 2002 - 2003 Eric Rosenzveig a Willy LeMaitre spolu s dalšími mediálními umělci a programátory. Dílo bylo rozdělené na tři části. První "plaListNetWork"byl opensource software vyvinutý na základě konzultací s umělci o audiovizuálním obsahu a rozhraním pro 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

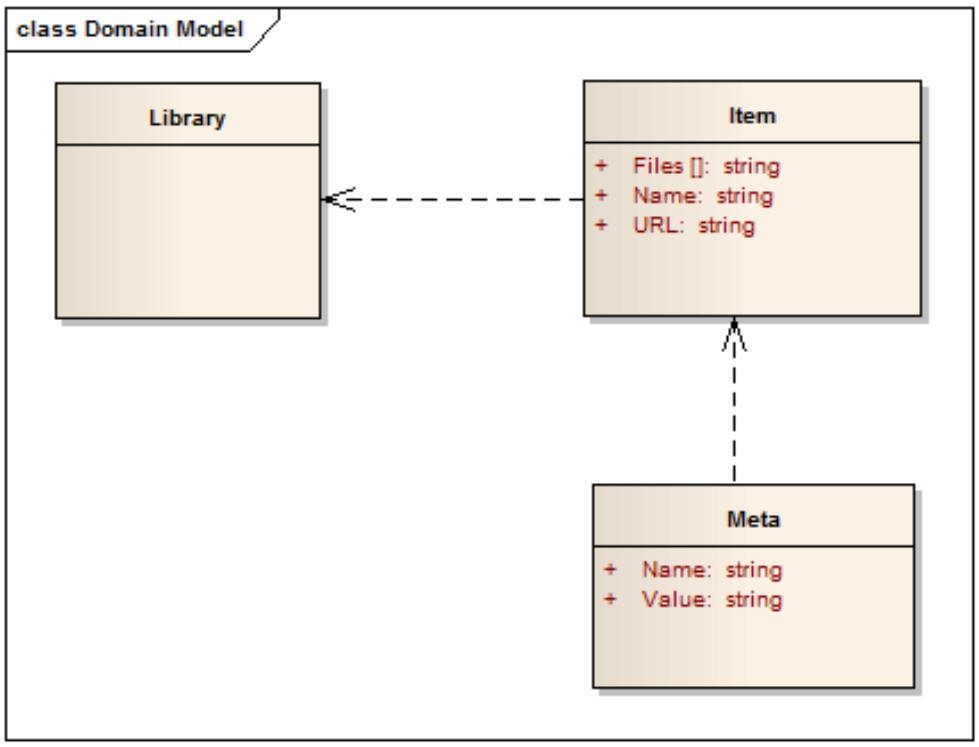
Narra je psaná v jazyce Ruby a poskytuje REST-API pro komunikaci se světem. Další použité technologie jsou Sidekiq, OmniAuth, MongoDB a Rails. Všechny tyto komponenty zajišťují stabilní jádro aplikace, na které je možné

připisovat další 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 má vazbu na projekt a knihovnu. Entita Item je reprezentována jménem souboru, url videa a vlastními staženými soubory. Dále obsahuje vazby na knihovnu ve které jsou uložena videa. Další vazba je na entitu MetaItem, která reprezentuje generátor pro metadata. MetaItem je statická třída, která obsahuje vazbu na Meta. Meta už obsahuje příslušné jméno a položky platné pro popis metadaty.



Obrázek 2.2: Doménový model mé části Narry

Moje část aplikace je tvořena entitami Item, Meta a Library. Entita library reprezentuje celou knihovnu videí, se kterou se bude lokálně pracovat. Item je jeden konkrétní prvek u kterého je potřeba provést validaci a inicializaci pomocí url. Jméno pro konkrétní prvek je reprezentováno stringem a url je také string. Pro jednoduché volání budu mít vytvořenou třídu Connector, která bude provádět inicializace, validaci, popis metadaty a stažení videa a případných titulků.

2.3 MongoDB

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

2.3.1 Formát BSON

Formát BSON 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 BSONu takto:

Tabulka 2.1: MongoDB příklad formátu BSON

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

2.4 YouTube API

YouTube Aplication Programming Interface je nástroj pro vývojáře, který umožňuje snadný přístup ke statistikám a datům na konkrétním YouTube kanálu. 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 na YouTube stránce.
- 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 APIs. Tato část rozhraní umožňuje začlenění funkcí 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 je potřeba si založit google účet a svůj vlastní API klíč. Vytvoření projektu a API klíče popíšu podrobněji v praktické části práce.

Celý koncept API umožňuje a vyžaduje získávání jen potřebných dílčích zdrojů, aby se zabránilo zbytečnému přenosu dat a nebyla přetěžována síť ani

procesor. Tento přístup zajišťuje efektivní práci a využití prostředků, proto jsou pořadavky rozděleny na části.

- snippet
- contentDetails
- fileDetails
- player
- processingDetails
- recordingDetails
- statistics
- status
- suggestions
- topicDetails

Díky rozkouskování do částí je možné se dotázat na specifické objekty a ušetřit si tak svou denní kvótu. V návaznosti dojde ke snížení latence a aplikace bude rychlejší.

2.5.2 Omezení

Každá aplikace má určitá omezení paměti, či časového kvanta, které uživateli přidělí. YouTube používá kvóty, 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 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. Ceny 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 stanovené množství kvót, neboť mohou číst a zapisovat odlišné množství čístí 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`. 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 a to je více než dostačující kapacita a proto nemusím omezovat metadata, která v mé aplikaci plánuji.

2.6 Testovací nástroj RSpec

2.6.1 Historie

RSpec[1] vznikl jako experiment Stevena Bakera, Davida Astelse a Aslaka Hellesøe. S vývojem začali v roce 2006, na Ruby on Rails. 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 mudulů, 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 testování je velmi podobná hovoru v angličtině. Používají se slova "describe"a "it", která mají stejný význam jako v mluveném slově.
"Describe an order."
"It sums the prices of its line items."

```
RSpec.describe Order do
  it "sums the prices of 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
```

Dále můžeme deklarovat vnořené skupiny pomocí describe, nebo context metod.

```
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*. Pro použití ve vývojářském režimu master je potřeba doplnit na začátek souboru tento kus kódu:

```
%w[rspec-core rspec-expectations rspec-mocks rspec-support].each do |lib|
  gem lib, :git => "git://github.com/rspec/#{lib}.git", :branch => 'master'
end
```

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 false hlášky 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 (?)
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

2.7 DublinCore

DublinCore je soubor metadatových prvků, jehož cílem je umožnit rychlé a snadné vyhledávání v elektronických zdrojích. Původně byl vytvořen jako popis metadat v 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.

2.7.1 Historie DublinCore

První setkání tvůrců DublinCore 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 metody 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 konali v říjnu 1997 v Helsinkách, kde byl kladen důraz na datum, pusobnost 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.

Tabulka 2.3: Schůzky DublinCore mezi lety 2000 - 2014

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

2.7.2 Struktura DublinCore

Pro mou bakalářskou práci jsem zvolil popis sekci 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. V následující tabulce je náhled na originální změní pořadavků, které jsem musel upravit. Více o této úpravě je napsáno v části Realizace, kapitola Popis metadaty.

2.8 Vyrovnávací paměť médií

2.9 Kodek H.264

Tabulka 2.4: Ujednocená struktura metadat

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 souřadnice mi, nebo lhůta či časové období.
Tvůrce (creator)	Subjekt zodpovědný za zpřístupnění zdroje.	Napříkald 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)	Primářní klíč zdroje.	Řetězec maximálně dvanácti znaků mezi ?v= a &, nebo koncem URL.
Jazyk	Jazyk zdroje.	YouTube API neposkytuje přímý přístup k jazyku, proto ho nebudu do mých metadat používat.
Vydavatel	To samé co Tvůrce.	Zde mají v metadatech duplicitu.
Vztah (relation)	Související zdroje.	V mám případě se jedná o categoryId.
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 v mých metadatech mít nebudu.
Předmět (subject)	Téma zdroje.	Toto je znovu entita, kterou YouTubeAPI neobsahuje.
Název (title)	Jméno dané zdroji.	Atribut title, který vracím samostatně pro lěpší práci s následným obsahem.
Typ (type)	Povaha nebo žánr zdroje	Toto je vždy video.

Kapitola 3

Realizace

3.1 Řešení spolupráce s YouTube API

3.1.1 YouTube Data API (v3)

API v3[?] umožňuje začlenění funkcí YouTube do vlastní aplikace. Proto jí 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`. 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. Dále je zde potřeba nechat si vygenerovat unikátní API klíč, pomocí kterého získávám metadata.

3.2 Třída connector

3.2.1 Založení aplikace

Před založením aplikace jsem několikrát navštívil R.U.R. v Praze, kde se projekt NARRA vyvýjí 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 `www.github.com`. Po předpřipravení projektu podle mého návrhu jsem si na githubu vytvořil účet, vlastní SSH klíč a mohl jsem si celou aplikaci k sobě natáhnout a začít programovat.

3.2.2 Validace a inicializace

Vlastní implementace je napsaná v jazyce Ruby a vývojovém 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 snažší 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. Nejdřívě jsem zkoušel do projektu zakomponovat metodu match. Ta ovšem vracela string místo booleovké hodnoty a proto jsem ji musel nahradit =~. Takto zkonstruovaný výraz by ovšem nefungoval úplně dokonale, neboť při funkční url by vrátil 0. Stačilo výraz lehce poupravit do tvaru !!(url =~RegExp) a vše fungovalo jak má.

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

Regulární výraz je matematický formalismus pro popis slov/vět jazyků. Jedná se tedy o způsob jak formálně popsat určité slovo, případně větu pomocí formálního vyjádření speciálními symboly a skupinami znaků. V ruby 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 standartní konstrukce, až na jednu vyjímku, která není až tak obvyklá. Jedná se o ?:výraz, což znamená že bude splněno při žádném nebo jednom výskytu výrazu. Všechny ostatní konstrukce jsou popsány v následující tabulce. Ruby má také velmi dobrou webovou podporu pro testování regulárního výrazu na stránkách `http://rubular.com/`.

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 youtube videu. Příkladem takové adresy je `http://goo.gl/TKMZjS`. Pouhým ověřením přes regulární výraz bych neměl šanci zjistit obsah a validitu odkazu.

Tabulka 3.1: Tabulka skupin symbolů pro regulární výraz

`[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 bílími znaky.
`\S`	Jakýkoli znak netvořený bílími 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.

3.2.3 Přesměrování

Přesměrování vyžadovalo novou knihovnu net/http, ze které jsem použil její zabudované metody. Při vytváření jsem nastavil horní hranici přesměrování na 20, neboť drtivá většina proběhne do tří požadavků. Dále jsem potřeboval zajistit, abych se netočil dokola na několika málo url, což by bylo neefektivní a hloupé. Pro maximum dvaceti prvků mi bohatě postačuje pole s lineárním procházením, neboť hash mapa by byla zbytečný luxus. Takto se mohu podívat zda jsem již nenavštívil nějakou url dvakrát, což by znamenalo zacyklené přesměrování a v mém případě vyhození příslušné vyjímky.

První úskalí knihovny net/http 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.

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, neboť každý jistě ví, že 2xx kód je symbolem úspěchu. Číslovka 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. Poslední skupina jsou kódy 4xx a 5xx značící chybu klienta a chybu serveru.

Po zjištění, zda je požadovaná url adresa validní, bylo potřeba ještě provést inicializaci. Zde vytáhnu z YouTube API všechny potřebné informace o videu, které budu dále zpracovávat. Zde je velké množství parametrů, které YouTube API nabízí k výběru. Dále je potřeba vygenerovaný API klíč, pomocí něhož YouTube pozná, komu ubrat denní kvótu za požadavek. V následující tabulce je příklad parametrů pro API.

Tabulka 3.2: Parametry YouTube API a jejich význam

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

Konrétní požavavek byl na adresu `https://www.googleapis.com/youtube/v3/videos?id=#{@videoid}&key=klíč&part=část`, kde @videoid je inicializovaná hodnota pro identifikátor videa, klíč je unikátní API klíč programátora a část je jeden, nebo několik parametrů oddělených čárkou. Zde je nutné vzít v potaz, že za vytížení YouTube serveru se platí a v jistých případech nemalou částí denní přidělené kvóty 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.

Snippet, který napíše o videu většinu informací. ContentDetails byl potřeba pro splnění pořadavků z DublinCore, statistics přidají do obsahu počty sledovaností a status zobrazí licenci a informace o sdílení videa. Tyto čtyři položky stačí pro pořadovaná metadata zadavatelem a při přidání dalších bych zbytečně omezoval maximální počet vrácených položek díky omezení YouTube

API a aplikace by pozbývala na efektivitě.

Při inicializaci je druhý parametr klíč, který slouží

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. Dalším požadavkem mateřkého projektu bylo vrácení typu videa :video. Tím jsem měl za sebou základní část a mohl jsem pokračovat s metadaty.

3.2.4 Metadata

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

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 objektu 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.

Tabulka 3.3: Metadata předávaná do NARRA

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.
dimension	Zda je video 2d, nebo 3d.
definition	Jaké je maximální možné rozlišení videa.
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.

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.

3.2.5 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 `https://www.youtube.com/v/`, za kterou stačí přiřadit identifikátor videa a požadovaný video stream je na světe.

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 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.

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

// integrační testy

// systémové testy

3.3.2 Testování třídy Connector

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 url. Zde jsem hledal různé url adresy, které nikam nepřesměrují a jsou validní. Dále jsem potřeboval najít nevalidní url, což stačilo vhodným způsobem modifikovat validní. Dále jsem potřeboval najít url, která se přesměruje na validní. Zde jsem použil funkci youtube, kde je možné zvolit sdílení videa, což vytvoří url, která se přesměruje. Poté jsem ještě použil google zkracovač, což je také velmi dobrá methoda pro ověření validace a přesměrování.

Další testy již probíhali pro určitý objekt konektoru. Zde jsem opět našel několik youtube videí na kterých jsem testoval příslušná metadata. Pro hezčí zápis testu jsem musel zápis metadat předělat z *{name:'channelId', value:'#{@channelId}'}* do zápisu *{'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.

V nástroji RSpec je v bloku before nekolik 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. To 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 kazdým blokem it, proto testy trvaly tak nehorázně dlouhou dobu. Po změně na :all se rázem čas testů dostal stabilně do deseti vteřin. Proto je potřeba dávat velký pozor, zda neděláme v programu stejné věci vícekrát.

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, neboť ve chvíli testování má již video statický obsah a již není vysíláno živě.

Dále jsem vytvořil jeden test pro vyzkoušení titulků. Zde jsem potřeboval ověřit zda mi program správě vyhodí vyjímku při neexistujících titulcích. Tato vyjímka 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 pomocí regulárního výrazu při validaci url a pro zkoušku vytažení všech dostupných dat.

Páté testované video bylo vytvoření kvůli možnosti stáhnout validní titulky k videu. Další test vyzkoušel časovou známku u titulků pro čas vytvoření. Blok číslo deset má na starost otestování možnosti stáhnout video, neboli dostat se na url, která obsahuje pouze video stream. Předposlední test znovu zkusil stažení titulků a oveření zda bude vyhozena vyjímka u neexistujících titulek. Závěrečný test je zde z důvodu omezení videí v různých zemích. Zde youtube dává hodnoty blockedIn v případě, že je video zakázáno v zemi, kterou vrací v poli. V případě, že video není nikde blokováno tento atribut zmizí. Znovu jsem proto musel otestovat správné reagování na omezení.

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 můj software obsahuje chyby na velmi malou. S otestovaným softwarem pomocí jednotkových testů mohu pokračovat v integračních a systémových testech s mateřskou aplikací.

Závěr

Literatura

[1] Chelimsky, D.: *RSpec. Behaviour Driven Development for Ruby. Making TDD Productive and Fun.* [cit. 2015-03-01].

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

Příloha **B**

Obsah přiloženého CD

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