Publishing Linked Data in Ruby on Rails

JACOB BROMS ENGBLOM

DM228X, Master’s Thesis in Media Technology (30 ECTS credits)
Degree Progr. in Media Technology 300 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Hannes Ebner
Examiner was Haibo Li

TRITA-CSC-E 2012:068
ISRN-KTH/CSC/E--12/068--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se/csc
Abstract

Through a prospective case study, revolving around the specific application Beakit, this investigation tries to establish what solutions are available and what the optimal route is for publishing Linked Data through the web framework Ruby on Rails. Based on Beakits technical specifications, three solutions were found fitting to test: RDF/XML-templating, JSON-LD, and rTriplify.

With rTriplify discovered as non-functioning, no solution was found allowing automatic mapping to triples and also fitting the needs of Beakit. Instead, manual mapping through either RDF/XML-templates or through JSON-LD was found to be the least intrusive, requiring little code refactoring, and simple way of publishing Linked Data. This process was further aided by Ruby on Rails content-negotiation capabilities and RESTfulness. Additionally, creating an ontology and adhering to some of the more peripheral principles of Linked Data was found to be time-consuming, and the former should be avoided unless absolutely necessary in favor of reliance on already existing ontologies.

Sammanfattning


rTriplify visar sig otillämpbart, och i och med detta kunde ingen lösningar hittas som tillåt automatisk mappning till triples och som samtidigt passade Beakits behov. Manuell mappning genom RDF/XML-templates eller JSON-LD visade sig istället vara de lösningar som fordrade minst ändringar av existerande källkod och som var minst invecklade. Dessa lösningars lämplighet visade sig också behjälpt av Ruby on Rails innehållsförhandling samt RESTfulness. Att skapa en ontologi och följa mer perifera Linked Data principer visade sig också tidsödande, och det förstnämnda bör undvikas om inte absolut nödvändigt till fördel för redan existerande ontologier.
Table of Contents

1 Introduction....................................................................................................................5
  1.1 Objectives and Aim.................................................................................................6
    1.1.1 Problem Definition.........................................................................................7
    1.1.2 Delimitations.................................................................................................7
    1.1.3 Motivation.......................................................................................................8
  1.2 General Background..............................................................................................8
    1.2.1 Prototyp AB....................................................................................................8
    1.2.2 Beakit.............................................................................................................8
    1.2.3 Beakit Specifications.......................................................................................9

2 Background and Theory............................................................................................11
  2.1 Publishing Linked Data..........................................................................................11
    2.1.1 RDF - Resource Description Framework.....................................................11
    2.1.2 Using Linked Data.........................................................................................12
  2.2 Ruby on Rails.........................................................................................................12
  2.3 Related Work and Available Tools........................................................................13
    2.3.1 Triplestores....................................................................................................13
    2.3.2 Relational Databases and RDF.................................................................13
    2.3.3 Ruby on Rails and Linked Data......................................................................15
    2.3.4 Overview.......................................................................................................16

3 Method........................................................................................................................18
  3.1 Case: Beakit...........................................................................................................18
    3.1.1 Available Approaches....................................................................................18
  3.2 Evaluation................................................................................................................18
  3.3 Method Motivation................................................................................................19

4 Implementation...........................................................................................................20
  4.1 Approaches.............................................................................................................20
  4.2 Application Data.....................................................................................................20
    4.2.1 Freebase........................................................................................................20
  4.3 Manual Templating: RDF/XML............................................................................21
    4.3.1 Vocabularies and Data Structure...............................................................23
  4.4 JSON-LD................................................................................................................24
    4.4.1 Vocabulary and @context..............................................................................25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5 Dataset Description</td>
<td>25</td>
</tr>
<tr>
<td>4.6 rTriplify</td>
<td>25</td>
</tr>
<tr>
<td>5 Results</td>
<td>27</td>
</tr>
<tr>
<td>5.1 General Results</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Manual Mapping</td>
<td>27</td>
</tr>
<tr>
<td>5.2.1 RDF/XML Templating vs. JSON-LD</td>
<td>28</td>
</tr>
<tr>
<td>5.3 rTriplify</td>
<td>29</td>
</tr>
<tr>
<td>5.4 SPARQL Endpoint</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>29</td>
</tr>
<tr>
<td>6 Discussion and Analysis</td>
<td>30</td>
</tr>
<tr>
<td>6.1 Beakit and Linked Data</td>
<td>30</td>
</tr>
<tr>
<td>6.1.1 Accomplished: Obtained Functionality</td>
<td>30</td>
</tr>
<tr>
<td>6.1.2 Mapping to Triples</td>
<td>30</td>
</tr>
<tr>
<td>6.1.3 Ruby on Rails</td>
<td>31</td>
</tr>
<tr>
<td>6.1.4 Other Scenarios</td>
<td>31</td>
</tr>
<tr>
<td>6.2 Shortcuts: Decreasing Complexity</td>
<td>32</td>
</tr>
<tr>
<td>6.2.1 Ontologies and Vocabularies: Leaving Things Out</td>
<td>32</td>
</tr>
<tr>
<td>6.3 JSON and the Semantic Web</td>
<td>33</td>
</tr>
<tr>
<td>6.3.1 JSON-LD and RDF</td>
<td>33</td>
</tr>
<tr>
<td>6.3.2 APIs and the Semantic Web</td>
<td>34</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>35</td>
</tr>
<tr>
<td>7.1 Recommendations</td>
<td>35</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>36</td>
</tr>
<tr>
<td>8 References</td>
<td>37</td>
</tr>
<tr>
<td>9 Glossary</td>
<td>38</td>
</tr>
<tr>
<td>10 Appendices</td>
<td>38</td>
</tr>
<tr>
<td>10.1 Appendix 1</td>
<td>38</td>
</tr>
<tr>
<td>10.2 Appendix 2</td>
<td>42</td>
</tr>
<tr>
<td>10.3 Appendix 3</td>
<td>45</td>
</tr>
<tr>
<td>10.4 Appendix 4</td>
<td>46</td>
</tr>
<tr>
<td>10.5 Appendix 5</td>
<td>48</td>
</tr>
<tr>
<td>10.6 Appendix 6</td>
<td>49</td>
</tr>
<tr>
<td>10.7 Appendix 7</td>
<td>53</td>
</tr>
</tbody>
</table>
1 Introduction

The Semantic Web is a bold vision of freeing and unifying data (1). It can be described as an enhancement of the present-day web that mainly consists of human-readable documents, and focuses on instead providing and publishing machine-readable documents. This is usually done in the form of RDF\(^1\) (Resource Description Framework), in accordance with the set of best practices summarized by the term Linked Data (2).

These practices make use of standardized protocols and specifications already in place as building blocks of the web like HTTP, URIs and RDF, and sets up the groundwork for a sort of standardized API. This enables easier exchange and usage of data across platforms and applications, and is a step towards the vision of transforming today’s Web into a more accessible global database.

So how would this be done in practice for a smaller application? Beakit\(^2\) is a service for saving media recommendations, primarily via e-mail. A friend recommends a movie and you simply send an e-mail off with the title in the subject line and it’s added to your Beakit list with some corresponding information such as a short description and where you can obtain this title. The application would after awhile have quite a significant amount of data. Being built on a smaller framework, how would an application such as this make its data available on the Semantic Web?

\(^1\) http://www.w3.org/TR/rdf-primer/  
\(^2\) http://www.beakit.com (note that code for this thesis is live at http://watchlist-stg.herokuapp.com)
1.1 Objectives and Aim

The technologies and standards orbiting the Semantic Web are numerous and to an extent alien to individuals not engrossed in the field. Even though these technologies and standards can be considered mature, it can be a daunting task to publish and use Linked Data, especially if your data is stored in the commonly used relational database. There are few out of the box solutions and developers are generally unfamiliar with the underlying technologies.
The purpose of this investigation is to evaluate different methods of publishing Linked Data for a specific application with specific needs and requirements, Beakit. This could give insight on available pragmatic solutions, what shortcuts can be taken, and the consequences they might have. In other words, this thesis will through a case study try to investigate, clarify and possibly establish more pragmatic paths for publishing Linked Data through a certain web framework, Ruby on Rails.

1.1.1 Problem Definition

The main question could be expressed as “What is the best solution, in terms of complexity, speed, and obtained functionality, for publishing Linked Data in Ruby on Rails?”

Publishing Linked Data here relates to making application data available publicly in RDF in the manner of the best practices associated with the term Linked Data. A solutions “..complexity, speed, and obtained functionality” refers to factors such as implementation time in regards of how much of the application had to be restructured, obtained functionality in relation to the Linked Data practices, and if additional expertise outside the Ruby on Rails skill set was required.

To handle the above it also becomes necessary to answer the question of what technologies and tools are available for publishing Linked Data with Ruby on Rails and what their requirements and perks are. Even if this investigation is locked to specific application it can be mentioned that the background and theory will not only handle technologies suited for Beakit, but a wider range of semantic technologies available for an application built on Ruby on Rails and whatever data layer it might use.

1.1.2 Delimitations

Publishing Linked Data will be the focus for this investigation, not consuming Linked Data (through a SPARQL endpoint or such). Beakit will not be going through a complete “semantification”.

As this is a case study with the purpose of looking at ways of publishing Linked Data from an already existing application, solutions that require changing to much of the underlying structure of the application will be considered unsatisfactory as an approach for this investigation. Changing out the relational database Beakit uses for a triplestore would be an example of such a change.

It can be additionally noted that the intent for this investigation is not to concoct a general step-by-step guide for publishing Linked Data in Ruby on Rails, and this text should not be used as one, and should instead be regarded more as a general evaluation of solutions befitting Ruby on Rails.

1.1.3 Motivation

With the steady and fast growth of the number of open APIs (3), it becomes increasingly
interesting to incorporate a larger portion of these with the Semantic Web vision. This vision, and the Linked Data practices, are not an easily accessible domain and it can be hard to navigate through accessible solutions, something that is certainly not helping with the growth of the Semantic Web.

In addition to this, most available more complete frameworks for handling Linked Data are Java-based heavier solutions, solutions not to attractive to commercial parties with smaller datasets who often use more lightweight frameworks such as .NET MVC, PHP frameworks, and Ruby on Rails.

An investigation revolving around a live application and with more commercial requirements was judged to be a good way to sift through possible solutions for publishing Linked Data in this scope, and search for possible shortcuts that could help in motivating developers to go the extra mile for creating an application that is integrated with the Semantic Web. Ruby on Rails was chosen partly based on the background of Beakit, but also because of rising popularity and to investigate its usefulness in the context of Linked Data.

**1.2 General Background**

To provide context to this investigation, here follows some information about the application Beakit and its setting. If unfamiliar with a term or acronym here or further on in this text, please consult the Glossary.

**1.2.1 Prototyp AB**

This thesis was done in collaboration with Prototyp AB, formerly Sanocore AB. Situated in Stockholm, Prototyp has seven employees and works primarily with client driven web development, both front-end and back-end. The company has also been known to develop its own products such as officegame.se, a game for helping office employees to stay active.

**1.2.2 Beakit**

The application which this thesis will orbit around is called Beakit. In brief terms, it’s a tool for quickly noting interesting media recommendations and bits of information. A user sends a keyword or two to the application server via e-mail and an entry is created. This entry is an intelligent guess of the desired media-item based on submitted keywords, and contains links to further information about the item. In Beakits infancy a media-item would either be a book, a movie or a TV-series. The application is thus at the start of this investigation built on APIs and data sources such as IMDB, iTunes, and Amazon, and would provide affiliation links to at least the latter two.

Beakit also endeavors to provide information about relations between different media items, such as two movies starring the same actor. This data is to be mined from freebase³ and interleaved and linked with more specific Beakit-oriented data. These sets are then

³ [http://www.freebase.com](http://www.freebase.com)
what would be published as Linked Data and serves as the fodder for this investigation. Further information on this is provided under Implementation.

1.2.3 Beakit Specifications

Beakit is of course a Ruby on Rails application. It uses Rails 3.1 with Ruby 1.93 and is deployed on the Heroku\(^4\) hosting service running a Thin\(^5\) web-server. The application is relational and is in production run on a SQLite3 database.

Basic Functionality

After a user signs up for the service, he or she can add Beakit-items, a book, movie or tv-serie, to a user specific list. This is either done by searching for an item while logged in and manually adding to the list by choosing from the results, or by sending an email with a search term to either movie, tv, or book@beakit.com depending on which media type is wanted. The service then tries to match a media item to the search term, and if successful adds the best match to the user list.

This item displays year of release, title, a short plot description, an attributed image, and affiliation links to the item on major retailers. An individual item can also be “ticked-off”, indicating ownership or consumption of that item, and can then be rated by the user on a scale of one to five.

These items are listed chronologically by when they were added and the list can be filtered to show items based on a text string, by whether it's been rated or ticked-off, or to show all removed items. Multiple filters can be applied at the same time.

Data Structure and Models

The main resources for the application are Beakit items and media items. A Beakit item is owned by a user and has all the user specific properties such as ticked-off or rated. It is then linked to one specific media item, which has properties regarding information about this item: title, year of release, description, and further coupling to affiliation link resources. All of this done in conjunction with the MVC and Active Record pattern resting atop a relation database.

At the end of this investigation, a media item will have even further data coupled to it, such as starring actors. See Application Data under Implementation.

\(^4\) http://www.heroku.com
\(^5\) http://code.macournoyer.com/thin/

Figure 2: Simplified view of model relations from the Beakit application. Full Active Record schema can be found under Appendix 1.
for further details.

**Application Processes**
The main bulk of processes active in Beakit are related to media item lookups of different kinds. When a user sends an email with a search term, it is routed to an application controller via Mailgun\(^6\). This controller checks to what address the email was sent, creates a Beakit item in accordance, and tries to match it to a local already existing media item. If unsuccessful, the service queues a Delayed_job\(^7\) to attempt to find it at a remote location. When a media item is created, Active Record callbacks are utilized to queue additional asynchronous jobs, matching additional data such as affiliations links to the media item. The whole process is very similar for the on-site search feature.

**Backbone.js**
Resources, i.e Active Record models, are outputted RESTfully and serialized as JSON, which are then handled by backbone.js\(^8\) providing javascript modeling and interaction. HTML is rendered by the client-side templating system icanhaz.js\(^9\) (based on the logic-less templating library mustasch.js\(^{10}\)) in conjunction with backbone.js.

This means that the client basically gets served some framing html and initiating javascript, but all interaction and object HTML rendering is done client side against a JSON interface, and a sort of crude REST-based API.

---

6  https://mailgun.net/
7  https://github.com/tobi/delayed_job
8  http://www.backbonejs.org
9  http://www.icanhazjs.com/
10 http://mustache.github.com/
2 Background and Theory

It is important to clearly establish the underlying criteria for this paper, the main one being that this investigation will accommodate to the needs and specifications of the application that is in focus for this trial: Beakit. This is to provide a more "genuine" developing situation including static requirements, clients and fewer possibilities to use cutting edge and not widely supported or heavily customized technologies.

Even if Beakit might not be able to utilize all of it, the background and tools presented below will as mentioned provide a sort of lay of the land for developing semantic applications, more specifically ones based on Ruby. This will in turn hopefully give some insight on what routes for publishing Linked Data that might be attractive under other conditions.

2.1 Publishing Linked Data

The use of the term “Linked Data” varies. The W3C definition could be summarized as “[a] collection of interrelated datasets on the Web” that is “available in a standard format, reachable and manageable by Semantic Web tools”\(^ {11}\). The standard format for providing information is most widely considered to be some flavor of W3C specified RDF, and this thesis will adhere to this definition and the best practices attributed to it.

A more succinct definition might be the four principles or rules that went with the original coinage of the term Linked Data by Tim Berners-Lee (2):

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).
4. Include links to other URIs. so that they can discover more things.

For more in-depth references to these practices and principles one can explore Bizer et als. tutorial (4).

To publish Linked Data you would thus need some “interrelated datasets” and then make these available in RDF for others to query. Some would also require a SPARQL endpoint to be set up to enable easier querying, but this investigation will not consider an endpoint vital for the publishing of Linked Data although it is a variable to handle when evaluating the results of the investigation.

2.1.1 RDF - Resource Description Framework

Designed as data model for describing resources on the web and uses triples for describing these. A triple is a sort of sentence, describing a resource by the means of

\(^{11}\) http://www.w3.org/standards/semanticweb/data
subject, predicate and object. RDF is a thoroughly mature and standardized specification of a content format and comes in many serialization formats, i.e. ways to output these triples. RDFa and RDF/XML are considered the most popular although there are many more like Turtle, N-Triple, and RDF/JSON. RDFa is here a bit different from the rest, as it is injected into HTML-views as opposed to being outputted separately. This thesis will default to RDF/XML as a serialization format unless specified otherwise.

2.1.2 Using Linked Data

Although this investigation will not directly handle consuming Linked Data, it can be worth mentioning the main reasons for its existence. The Linked Data practices provide access to stores of structured data via well-established technologies, such as HTTP URIs, enabling a base for a standardized API-structure and the interlinking of these. In turn, this paves for richer and more dynamic applications as larger and more structured stores of data become available to them.

This Semantic Web vision does however depend on large amounts of data actually being published as Linked Data, which in turn require simple ways for applications and developers to provide this data. As we will see, Ruby on Rails seems aptly enabled for handling and publishing Linked Data, giving the main motivation for this investigation.

SPARQL

One specific tool for using Linked Data is the RDF query language SPARQL\textsuperscript{12}. Given a set of RDF graphs, SPARQL queries enable interaction with this stored structured data. Formulated queries can be posted to one or more so-called SPARQL endpoints, which then return relevant data for further processing.

If no endpoint is provided for a certain dataset, a RDF-dump is usually available for download. This dump can then be queried against via a RDF-API, a toolset manipulating RDF, specific examples of which can be founded under Related Work and Available Tools.

2.2 Ruby on Rails

Ruby on Rails\textsuperscript{13}, often shortened RoR, is an open source web application framework based on the older object-oriented programming language Ruby.

It uses the Model/View/Controller design pattern (MVC), often in combination with the Active Record pattern. Apart from validating input and output from models, this enables easy manipulation of database objects by acting as a sort of API for accessing these through SQL, packaging a database table into a model class.

With its architecture, in using conventionally CRUD-functions among other things, Ruby on Rails also encourages and by default implements RESTful practices which helps in keeping URIs clean and dereferenciable. In combination with being MVC-based, and

\textsuperscript{12} http://www.w3.org/TR/rdf-sparql-query/
\textsuperscript{13} http://www.rubyonrails.org/
having a separated View object, outputting data in other formats than HTML is also an easy task. All of these aspects make Ruby on Rails especially interesting as a web framework for working with and handling Linked Data.

Ruby on Rails is far from the most popular web development framework out there, but it has been growing quickly and demand for the expertise in the domain has been rising (5). Since a few years back, in tandem with its rising popularity, cloud based hosting services for the framework have started appearing. Heroku is one of these, and is where Beakit is hosted.

2.3 Related Work and Available Tools

There is some literature on publishing Linked Data, although they are not case studies but more of general how-to’s and explanations of the principles of Linked Data (4,6). These are highly recommended for better getting to know the technologies involved.

2.3.1 Triplestores

“As RDF and RDFS have gained ground, the need for repositories that can store RDF content has grown. These so-called triplestores vary in their capabilities. Some focus on providing a rich means to reason over the triples (for example, see http://jena.sourceforge.net), while others focus on storing large quantities of data (see http://sourceforge.net/projects/threestore).” - Berners-Lee et al., 2006 (7).

A triplestore is a database built with the purpose of storing triples (instead of the usual relational data), triples which in conjunction form a RDF-graph. They not only vary in capabilities, but in code base. Some are built from the ground up while some are based on existing relational database system such as MySQL or PostgreSQL. These latter require RDF-oriented query languages (such as SPARQL) used to retrieve triples to be converted to SQL, which is an active and tough field of research (8–10).

Triplestores are often bundled with SPARQL servers (as the case for example 4store) but also in what can be described as Semantic Web Frameworks, a few which are covered below under the section Open Source Semantic Web Frameworks.

2.3.2 Relational Databases and RDF

Most applications today are based on some type of relational database, so publishing Linked Data will in the majority of cases involve converting the content of a relational database to RDF in some manner. There is an abundance of research concerning conversion or mapping of relational databases to RDF and optimizing this process (8,10–15), as it can be considered vital for the growth of the Semantic Web. A mapping language standard is even in the works from W3C\textsuperscript{14}. There are, however, not nearly as many easily accessible tools for the job.

\textsuperscript{14} http://www.w3.org/TR/r2rml/
Mapping Relational Databases

There are some surveys and investigations attempting to structure and bring clarity to what available solutions for mapping relational databases exist and what their specifications are. A W3C survey (16) from 2009 provides a status update on current approaches to mapping relational databases as well as how far along they have come. Hert et. al (17) go further and try to provide a more detailed comparison of mapping languages in obtained functionality. Below follow some of the more complete and distributed solutions for mapping relational content into Linked Data views.

**D2R Server**

D2R Server\(^\text{15}\) is a publishing solution based on the D2RQ mapping language (14) and through it provides a Linked Data view over the content of a non-RDF relational database. This view is a real time mapping of relational content to virtual RDF graphs and can be interfaced against through SPARQL and, as it is Java based, with the Jena and Sesame APIs mentioned below.

The real time mapping requires a D2RQ mapping specification file to be provided, that states instructions on how the content of the non-RDF relational database should be mapped.

**Triplify**

Triplify\(^\text{16}\) is an attempt to decrease the complexity for the process of publishing Linked Data. Based on a configuration file specifying common queries within the application, it takes the approach of mapping common HTTP-URI requests to mentioned SQL-queries and transforming the results into published Linked Data in the form of a RDF-dump (13).

Written primarily for widely distributed PHP-based applications, such as phpBB and Wordpress, and distributed as a lightweight “plugin”, it is a pragmatic solution with the aim of quickly publishing Linked Data. For custom solutions it becomes slightly less easily applicable, as the configuration file has to be rewritten for each new application.

There does exist a Ruby on Rails port of Triplify called rTriplify\(^\text{17}\), last updated 2010. It is written for Rails 2.

**Open Source Semantic Web Frameworks**

There are several tool sets available that bundle many sorts of capabilities. These Semantic Web Frameworks often consist of a triplestore, a RDF API, a SPARQL engine as well as an assortment of database mapping tools.

**Jena and Sesame**

One of the most mature and widely mentioned Semantic Web frameworks is the Java based Jena toolkit\(^\text{18}\). It consists of several components including a RDF API, a RDBMS

\(^{15}\) http://d2rq.org/d2r-server

\(^{16}\) http://www.triplify.org/

\(^{17}\) http://github.com/rtriplify/rtriplify

\(^{18}\) http://jena.sourceforge.net
(relational database management system) based triplestore, a SPARQL compliant query engine and servers enabling the publishing of RDF.

Sesame\textsuperscript{19} is quite alike in being Java based as well as having similar functionality in enabling interaction with and storage of RDF. The two do interface against RDBMS differently, the details of which will not be covered in this text.

**Redland**

Another well established RDF framework, Redland offers much the same components as Jena and Sesame with the difference of being written in C and having language bindings for Perl, PHP, Python and Ruby. It provides an in-memory or on-disk RDBMS based triplestore, SPARQL capabilities and a RDF API with support for multiple RDF syntaxes.

### 2.3.3 Ruby on Rails and Linked Data

Although general guides exist, little to no research done on the topic of publishing Linked Data with specifically Ruby on Rails exists. However, there are some tools relevant to Ruby and RDF APIs, and although this would be mostly of interest when using data already stored as RDF, they will here be briefly covered.

**ActiveRDF**

ActiveRDF is an object-oriented RDF API (18). Its authors suggest that high-level programming languages like Ruby are especially suited for an object-oriented API for the triple-oriented RDF because, among other reasons, of not being strongly typed.

Although ActiveRDF seems promising for handling RDF in Ruby, and has stated support for Rails and Active Record, the project and its documentation seems to be abandoned and unmaintained. As a probable consequence of this, ActiveRDF seems to be incompatible with Rails 3.

**RDF.rb**

The most frequently updated RDF API, RDF.rb\textsuperscript{20} is arguably the best library/gem for handling RDF in Ruby as of this writing moment. It is a pure Ruby library, and has no Active Record bindings.

As mentioned, RDF APIs are built for interfacing with RDF stores and handling SPARQL query answers. They are less useful when publishing Linked Data, but could fill a role when “manually” formatting data from non-RDF sources to RDF-output.

**JSON-LD**

This Linked Data format\textsuperscript{21}, which has a ruby implementation made to be coupled with

\textsuperscript{19} http://www.openrdf.org/
\textsuperscript{20} http://rdf.rubyforge.org/
\textsuperscript{21} http://www.json-ld.org/
RDF.rb, is extra interesting in the context of Beakit. As mentioned in the Beakit Specifications, backbone.js is used which entails a JSON interface and JSON output from some controllers instead of HTML output. JSON-LD could be a more integrative way, however unstandardized, of publishing Linked Data and at the same time using it in-application as opposed to having a separate RDF/XML output.

The approach of JSON-LD is different to RDF/JSON, since it does not require altering the structure of existing JSON-output as well as the applications using it. This is because the format is more lightweight than RDF/JSON, it isn’t RDF, and is constructed to enable simple appending of semantic markup.

**RDFa on Rails**

RDFa on Rails is a library that gives access to some helper methods for outputting RDFa in views. It comes with presets for outputting RDFa coupled to Dublin Core and FOAF, but has support for using custom vocabularies written in RDFS.

**Redland with Ruby Bindings**

See section Redland above.

### 2.3.4 Overview

<table>
<thead>
<tr>
<th></th>
<th>SPARQL-endpoint</th>
<th>Java-based</th>
<th>RDF API</th>
<th>Automated Mapping Scheme</th>
<th>Bundled triplestore</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D2R Server</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, via Jena</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Jena</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Sesame</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Redland</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, but with bindings for additional languages</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

_Table 1: Java-based tools and/or requiring triplestore_
<table>
<thead>
<tr>
<th>Tool</th>
<th>SPARQL-endpoint</th>
<th>RDF API</th>
<th>Automated Mapping Scheme</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveRDF</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Abandoned and unmaintained.</td>
</tr>
<tr>
<td>RDF.rb</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>JSON-LD</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Also a RDF-format.</td>
</tr>
<tr>
<td>RDFa on Rails</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>For writing RDFa.</td>
</tr>
<tr>
<td>rTriplify</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Port of Triplify for Ruby on Rails.</td>
</tr>
</tbody>
</table>

*Table 2: Ruby-oriented tools*
3 Method

This investigation is a prospective case study revolving around a particular Ruby on Rails application. By looking at a specific application and available semantic technologies, a pragmatic selection of possible solutions to the task of publishing Linked Data are to be chosen and evaluated. This selection and evaluation will provide qualitative data, such as obtained functionality and ease of implementation, concerning Ruby on Rails and the Linked Data practices. The focus on this type of data, instead of for example more quantitative benchmarking, is that these generally tell you very little about the implementation process, something here considered important because of its impact on who chooses to publish Linked Data.

3.1 Case: Beakit

The case study will handle the specific application Beakit. As mentioned in the delimitations, solutions that entail changing to much of the core structure of the application will be discarded as overly complex. This is to provide a more realistic simulation of a commercial development approach where it is unlikely that a decision would be made to overhaul the whole data layer simply for the purpose of publishing Linked Data. It is not the question of publishing Linked Data at any cost.

3.1.1 Available Approaches

Based on the introductory study for this text, possible approaches of publishing Linked Data from Beakit are to be chosen for evaluation. The selection will take into account the Beakit specifications and weigh the implications of these in conjunction with available technology. The concretization of this selection can be found under Implementation/Approaches.

3.2 Evaluation

The available approaches are to be evaluated, as specified in the problem definition “...in terms of complexity and speed as well as obtained functionality...”. Quantitative measurements for these, especially a solutions “complexity”, are hard to establish. Partly because of a web applications many components and interlocking technologies, but also due to a developers subjectivity and the environment he or she uses.

Because of this, “complexity” will here be a more loosely descriptive measurement including such factors as implementation time, the amount of steps involved and how much of the application structure that had to be changed. The “obtained functionality” can be more concretely defined and concerns what parts of the Linked Data practices that are achieved. Some possible discerning factors could be whether a SPARQL-endpoint is obtained or whether new data is automatically published or needs manual updating.

In short, the applied evaluation criteria could be organized and described as:
• a solutions complexity and speed of implementation
  • in terms of how much the application had to be restructured.
  • in terms of what additional knowledge was required outside basic Ruby on Rails acquaintance.

• obtained functionality in relation to the recommended Linked Data practices
  • providing data via dereferencable HTTP URIs.
  • functionality in accordance to the Linked Data Publishing Checklist (4).
  • RDFa or RDF/XML output.
  • SPARQL-endpoint available.

Basic Ruby on Rails acquaintance is here regarded as the basic knowledge necessary for constructing a web service in Ruby on Rails and includes basic familiarity with HTTP, SQL, HTML/XML, JSON, CSS, and JavaScript, as well as the Active Record- and MVC-pattern.

3.3 Method Motivation

As mentioned in the Method introduction this study will provide qualitative data with the intent of clarifying paths for simplified implementation of solutions for publishing Linked Data from the framework in question. Because of the lack of meaningful quantitative measurements for this angle of “simplified implementation”, for example what would implementation time in minutes for a certain developer convey, a more qualitative prospective approach was chosen. As indicated by the evaluation criteria above, the implementation trials spawned from this prospective approach are then to be broken down into more concrete elements to indicate how and why a solution is, for example, complex and time-consuming.

Constructing this investigation as a case study offers its risks and benefits. The main reason for orbiting this trial around the specific application Beakit is to provide real criteria and boundaries for what solutions are acceptable and which are not. With an application already in place it generally becomes harder to use cutting edge technology and more heavily customized solutions and a more realistic development situation is simulated. Of course, a risk then becomes that this specific application has some quirk limiting the amount of possible solutions and making the investigation unapplicable in the majority of other cases. The Discussion and Analysis of this text will try to isolate such quirks and handle their possible consequences.
4 Implementation

This section contains the specification of the approaches for publishing Linked Data from Beakit as well as development steps for the different approaches. All development was done on a MacBook Pro running OSX 10.6.8.

4.1 Approaches

As one can gather from the background above, there is a fair amount of tools for working with and publishing Linked Data. One could also summarize that there is more to choose from if one is willing to implement or convert to a triplestore or if using Java. These solutions, Jena for example, also generally do a lot more than help with publishing Linked Data, although you could certainly pick what components to use.

Since Beakit is neither Java based or could with ease be converted to running on a triplestore, the choice of usable tools for publishing RDF diminishes. Beakit as mentioned also uses backbone.js, which affects the possible solutions since it entails client rendered HTML. This means that RDFa becomes an impossible serialization format to use, as it would be invisible to crawlers and machines that do not parse javascript. On the other hand, it makes JSON-LD a more attractive choice, as JSON is already being outputted in a Linked Data like fashion, with dereferencable URIs.

The options left to choose from were decided to be:

- manually mapping data through Active Record into RDF/XML templates, possibly with the help of RDF.rb or Redland with Ruby bindings, and maintaining URIs to point to entities.
- outputting and/or implementing JSON-LD.
- implementing rTriplify (entails writing a triplify configuration file).

One could argue that for instance running a D2R server might also be a possible solution, but maintaining a separate server interfacing the data layer of a cloud based application does not seem very practical.

4.2 Application Data

Before the actual venture of publishing Linked Data some preparations and decisions were made concerning the test data to be published and the structure of the same.

4.2.1 Freebase

To provide additional data for the application to use and more importantly publish, Beakit media items of the type “film” were appended with additional data obtained from the community driven graph database Freebase. Functionality that made use of the Active Record callbacks was implemented so that whenever a new media item was created,
additional data about it was retrieved via the Freebase API based on its IMDB ID.

Data fetched, and which for new models were created for, was:

- actors, producers, writers and directors for the film.
- languages and countries of origin.
- locations of interest.
- genre belonging.
- production companies.

This data was linked to media items through several many-to-many relation, as well as with direct links to freebase entities. The freebase link, in combination with data originating Beakit itself, was what was decided to be published as Linked Data, since all the other data originated from freebase, was already structured, and could be accessed via that link.

Data that was found pertinent to publish was

- the name of the media item.
- number of Beakit lists a media item is on.
- average Beakit user rating for a media item.
- the media item on freebase.
- affiliation links, i.e the media item on iTunes and Amazon.

With this in mind, we continue on with descriptions of the attempted implementations specified under Approaches.

### 4.3 Manual Templating: RDF/XML

The first step to publishing Linked Data with this approach was to enable controllers to respond to RDF-requests. This means registering RDF as a MIME-type:

```ruby
Mime::Type.register_alias, "application/rdf+xml", :rdf in /config/initializers/mime_types.rb and then in each relevant controller and controller method adding a respond_to block. For the URI beakit.com/media_item/:id mapped to the controller method show it would be:
```

```ruby
respond_to :html, :rdf

def show
  if params[:format]
    coder = HTMLEntities.new
    @item = MediaItem.find_by_id(params[:id]) #Active Record object

    respond_to do |format|
      format.rdf do
        coder.encode @item
      end
    end
  end
end
```
This means that a GET request to, for example, the URI beakit.com/media_item/157 would check the HTTP Accept header and do a HTTP Status 303 redirect to either beakit.com/media_item/157.html or beakit.com/media_item/157.rdf. These routes would then return either JSON-output (see chapter backbone.js) or return RDF output for the item with an internal ID of 157. The RDF output would be templated through /app/views/media_item/show.rdf.erb as RDF/XML:

```xml
<rdf:Description rdf:about="<%= @item.get_URI %>">
  <rdfs:label>Description of the media item <%= @item.name %></rdfs:label>
  <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"><%= @item.created_at %></dcterms:created>
  <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"><%= @item.updated_at %></dcterms:modified>
  <rdf:type rdf:resource="beakit:Item"/>
  <foaf:primaryTopic rdf:resource="<%= @item.get_URI %>">
    <beakit:name rdf:resource="<%= @item.name %>">
      <% if (@item.get_FB_URI) %><owl:sameAs rdf:resource="<%= @item.get_FB_URI %>">
      <% end %>
      <% if (WatchlistItem.where(:media_item_id => mediaItem.id).average('rated')) %>
        <beakit:averageBeakitRating><%= @WatchlistItem.where(:media_item_id => mediaItem.id).average('rated') %></beakit:averageBeakitRating>
      <% end %>
      <% if (@item.get_FB_URI) %><owl:sameAs rdf:resource="<%= @item.get_FB_URI %>">
      <% end %>
      <beakit:onAmountofLists><%= @item.watchlist_items.count %></beakit:onAmountofLists>
      <% if (@item.affiliate_links.where(:affiliate_type => 1).nil?) %>
        <beakit:amazonAffiliationLink><%= @item.affiliate_links.where(:affiliate_type => 1).first.url %></beakit:amazonAffiliationLink>
      ```
As a last step, before finally being served, the above populated template is inserted into a application wide RDF/XML-template with name-spacing for different ontologies and vocabularies, including an application specific one, the description of which follows under Vocabularies and Data Structure.

An alternative to this manual templating would be the use of a RDF API, such as RDF.rb. The notions expressed in the background, regarding the primary usefulness of these mainly manifesting when interfacing against RDF stores, were here however verified. It became obvious that transforming relational content (Active Record objects) to triples and inserting a middle layer in the form of RDF.rb, simply to get some generated RDF was a very cumbersome, illegible, and excessive approach for simply publishing Linked Data as most of RDF.rb capabilities would be remain unused.

4.3.1 Vocabularies and Data Structure

RDF in itself does not provide thoroughly descriptive information about data structure. As Bizer et al. (4) describe it:

“RDF provides a generic, abstract data model for describing resources using subject, predicate, object triples. However, it does not provide any domain-specific terms for describing classes of things in the world and how they relate to each other. This function is served by taxonomies, vocabularies and ontologies expressed in SKOS (Simple Knowledge Organization System), RDFS (the RDF Vocabulary Description Language, also known as RDF Schema) and OWL (the Web Ontology Language).”

One should try to re-use existing well-established ontologies when possible, such as FOAF\(^\text{23}\) and Dublin Core\(^\text{24}\), but since a Beakit item is a new type of entity it was deemed necessary to create some custom terms and classes. For this implementation RDFS was chosen, as it is considered more suitable for simpler ontologies, often called vocabularies.

The Beakit vocabulary consisted of just one Class, a Beakit Item, with properties corresponding to the data specified to be published under chapter Application Data (full vocabulary file can be found under Appendix 2). This vocabulary file was then set up to be served at /vocab and referenced together with other more prominent ontologies, like Dublin Core and FOAF, in the application wide RDF/XML-template (see Appendix 3 for details).

Sample RDF-output for this implementation story can be found under Appendix 4. This

\(^{23}\) http://www.foaf-project.org/
\(^{24}\) http://dublincore.org/documents/dces/
output was validated via the W3C Validation Service\textsuperscript{25}.

### 4.4 JSON-LD

For the same reason that RDF.rb was considered excessive in the context of manual templating to RDF/XML, so was the ruby implementation of the JSON-LD processor, the JSON-LD Reader/Writer. The format in itself though, was applicable for the case of Beakit.

As seen in Code block 1, a HTTP GET request with a HTML HTTP Accept header renders and outputs JSON based on the return value of a method `mergeTables`. This method takes an Active Record object and builds a view model, based on data from related tables, to render as JSON. This output can easily be transformed to comply with JSON-LD syntax, in its simplest form, by appending `@context-` and `@id-properties` indicating a sort of vocabulary and the subject of a triple, respectively.

As variables starting with "@" are reserved in Ruby as instance variables, the view model class to be serialized to JSON had to be rewritten to inherit Hash, and then constructed using hash keys instead of instance variables as seen below.

```ruby
def mergeTables(mediaItem, coder=HTMLEntities.new)
  mi = FilmViewModel.new
  mi['@context'] = inlineVocab #method constructing @context JSON
  mi['@id'] = mediaItem.get_URI
  mi['@type'] = "http://#{ENV['domain']}/vocab#Item"

  mi['name'] = mediaItem.name
  mi['freebase'] = mediaItem.get_FB_URI
  mi['created'] = mediaItem.created_at
  mi['updated'] = mediaItem.updated_at

  mi['onAmountofLists'] = mediaItem.watchlist_items.count
  mi['averageBeakitRating'] = WatchlistItem.where({media_item_id => mediaItem.id}).average('rated')

  mi['amazonAffiliationLink'] = mediaItem.affiliate_links.where({:affiliate_type =>1}).first.try(:url)
  mi['itunesAffiliationLink'] =

\textsuperscript{25} http://www.w3.org/RDF/Validator/
mediaItem.affiliate_links.where(:affiliate_type => 2).first.try(:url)
...

    return mi
end

Code block 3, /app/controllers/media_item_controller.rb

The JSON-LD MIME type “application/ld+json” was also registered and the respond_to blocks were also slightly modified to incorporate JSON-LD requests. Both legacy JSON requests and JSON-LD requests were set to receive JSON-LD responses, as JSON-LD is also valid legacy JSON. See Appendix 5 for details.

4.4.1 Vocabulary and @context

The JSON-LD context can be described as a sort of intermediate vocabulary, enabling JSON key-value pairs to be interpreted as triples by a JSON-LD processor. There are several syntax’s for using a context, however only the first mentioned and non-compact version was implemented for this investigation. With an ontology already in place (see Vocabularies and Data Structure), constructing the @context-output here consisted of simply specifying a predicate for each JSON-key.

The @context-output and sample JSON-output for this implementation story can be found under Appendix 6. This output was validated via the JSON-LD playground^26.

4.5 Dataset Description

To be full compliant with the Linked Data practices, metadata about the dataset itself was to be provided. In accordance with the instructions from Heath & Bizer (4) this entailed a semantic sitemap and a URI concerning the dataset itself, in this case sitemap.xml and /datasets/media_items respectively. These were both files consisting of fairly static data such as RDF-dump locations and were thus created as static files in XML and RDF/XML. Each individual media item was also decorated with a dcterms:isPartOf, indicating its belonging to the media_item dataset, as well as dcterms:publisher and dcterms:licensing, providing provenance and licensing data. The sitemap file can be found in Appendix 7.

A RDF-dump of the dataset in question was also set up, both in RDF/XML and JSON-LD. This process was in essence the respective implementation as stated above, but in a for-loop and served at /dumps/media_items.

4.6 rTriplify

The ruby port of Triplify is unfortunately written for Rails 2 and uses some deprecated

---
^26 http://json-ld.org/playground/
methods incompatible with a Rails 3 application like Beakit. The first step was thus an attempt to unpack the rTriplify gem and convert it to a Rails 3 plugin. A brief amount of time was spent on this endeavor, the details of which will be briefly covered here.

To gain access to routing, which is essential for the output of Linked Data, the plugin was rewritten to be explicitly mounted as a Rails engine. In Rails 2, this was automatically done. This lead to the plugin at least outputting a N-triplet typed file (“data.n3”), albeit empty. It was hard to disclose whether this was some incompatibility issue, or whether the mappings-file was the issue. There were no error-messages. The prudent approach was judged to be to thoroughly rewrite it as a Rails 3 gem, since large structural differences were apparent.

The implementation attempt was here halted, since it was deemed unlikely that in a real development situation the task of rewriting rTriplify as a Rails 3 gem would be undertaken.
5 Results

As this was a more qualitative investigation, evaluation in accordance with the method criteria will not be presented in respect of, for example, the specific amount of minutes an implementation took. Instead, it will gauge solutions complexity relative to each other, and indicate which steps were particularly time consuming and intricate as well as what obtained functionality could be attributed to it.

5.1 General Results

In the quest of publishing according to the Linked Data practices, tasks were undertaken that were necessary for more than one of the implemented solutions. A vocabulary for the Beakit data model was one of these.

Ontologies and vocabularies as well as the technologies concerning them can be considered complicated. In the case of Beakit, where it was judged necessary use a custom vocabulary instead of solely reuse existing ones, the creation of this vocabulary was the most complex step as well as the most time consuming. Unless very familiar with OWL, RDFS and ontologies in general, this will most likely be the result for most developers. If not constructing a more semantic application, with semantic browsers in mind, this could also be seen as the least rewarding step as the vocabulary will not be fully utilized.

The static dataset descriptions and the provision of triple-dumps were also a task undertaken for the two approaches in the manual category. Although a JSON-LD dump was implemented, JSON-LD dataset descriptions were not, although it certainly was doable, since the process would be near identical. Although not complex, overly time consuming or with any implications for the application in the whole, this step should still be taken into account time wise.

With a vocabulary and dataset descriptions in place these requirements from the Linked Data publishing checklist were satisfied, for all attempted solutions:

- providing provenance metadata.
- providing licensing metadata.
- dereferencable URIs of proprietary vocabulary terms.
- mapping of proprietary vocabulary terms to other vocabularies.
- dataset-level metadata provided.
- references to additional access methods.

5.2 Manual Mapping

Except for rTriplify, in terms of obtained functionality and compliance with Linked Data practices the implemented solutions were very similar. In essence the different solutions implemented could be divided into two general categories, the plugin/configuration category of rTriplify and the more manually oriented category of JSON-LD and RDF/XML.
With the content negotiation capabilities and general specifications of Ruby on Rails, the two in the latter category could be handled very similarly and also got some functionality for “free”.

With backbone.js and the RESTfulness of the application, media item URIs were basically dereferencable, based on the their database table ID, from start. With content negotiation in place, simply handled in Ruby on Rails with an added MIME type and some redirects, the largest step became mapping, converting and serializing data to triples in different formats. This was not something requiring any significant restructuring of existing code and could not be considered overly complex, although identifying and applying the right predicates and data types was time-consuming as well as chore-like. The different Linked Data formats, JSON-LD and RDF/XML, also diverged implementation wise in this step.

### 5.2.1 RDF/XML Templating vs. JSON-LD

The RDF/XML mapping process entails a slightly more cumbersome route, requiring the creation of separate view-template files and writing markup, and opening up for syntactical errors. It can however be noted that this implementation method also results in some flexibility, as the process would be virtually identical for any type of RDF-serialization as you simply convert the markup in the template.

The mapping process for JSON-LD was instead a sort of embellishment of already existing JSON-output, one not altering the JSON-interface and thus not affecting the workings of backbone.js and the application itself. This was slightly less time consuming than the RDF/XML mapping process, facilitated by the legibility of JSON and the Ruby on Rails JSON serialization functionality.

However, it must be noted that JSON-LD is not a direct representation of RDF and thus publishing in this format can be considered a deviation from the Linked Data practices. This could be seen as positive and negative, and these two formats set against each will be handled further under Analysis and Discussion.

Another blemish on the JSON-LD approach was the discrepancy between JSON output used by the application and output: not all output specified in the @context was used by the application resulting in useless database queries, and not all JSON output was incorporated in the @context. The latter, however, must be considered a minor issue, since the JSON-LD processor seems to ignore keys unspecified in the @context.

In addition to providing data, in the form of triples, via dereferencable HTTP URIs this “last” step also met these requirements from the publishing checklist:

- linking data to other datasets.
- using terms from widely deployed vocabularies.

With this, both of these two approaches have to be considered to be in full compliance with the Linked Data practices, with reservation for JSON-LD not being a pure RDF serialization.
5.3 rTriplify

As rTriplify was never successfully implemented under this trial, nothing conclusive can be said for this solution as it can’t be fully evaluated according to the method criteria, especially the parts concerning obtained functionality. During the attempts of implementation however, some indications of complexity were found, especially for the composition of the mappings file, in rTriplify a .yml file. How this file was to be constructed is largely undocumented and required a lot of time to get familiar with, especially if not very comfortable with more complex YAML.

Of course, this whole process was made significantly harder since the actual data-output was not functioning, resulting in no feedback on whether something was being done correctly or not. It can also be noted that except for a few lines of code concerning routing, nothing had to be changed in the workings of the original Beakit application.

5.4 SPARQL Endpoint

A SPARQL endpoint was not obtained for any of the successfully implemented approaches, and for Triplify it is specifically mentioned that no endpoint is provided. Since no triplestore is implemented in the case of Beakit, this is not surprising. It is also not of huge consequence and, as mentioned before, not here considered a requirement for publishing Linked Data. An alternative way for using SPARQL could for instance be for a user to fetch RDF-dumps, and process and query it on their end.

5.5 Summary

A more brief and concrete summary for the results of this investigation:

- in the case of Beakit, constructing a vocabulary was found to require a quite significant amount of time as well as, and because of, the requirement of knowledge outside of the Ruby on Rails toolset.
- the processes for manually mapping data to triples, as in the case of JSON-LD and RDF/XML templating, were similar both in terms of obtained functionality and complexity, and greatly facilitated by the applications RESTfulness and Ruby on Rails content negotiation capabilities.
- JSON-LD was more legible and faster to implement than templating RDF/XML due to a existing JSON interface and Ruby on Rails serialization functionality.
- the rTriplify trial was unsuccessful and inconclusive but indications of complexity and hardships concerning the mappings file and syntax were experienced.
- no SPARQL endpoint was obtained for any of the attempted implementations.
6 Discussion and Analysis

As one can suspect, from these results it is hard to provide an absolute answer to what is the “best” way of publishing Linked Data in Ruby on Rails since the answer to this question is in heavy relation to what application it concerns. However, from the case of publishing Linked Data from Beakit, some possible shortcuts were discernible as well as indications of Ruby on Rails suitability for handling Linked Data.

6.1 Beakit and Linked Data

Beakit might not have been the best case for this investigation, since it produces relatively little of its own data to be made publicly available. Even if this can be seen to slightly undermine the realisticity and the authenticity of the case, the insight achieved on the amount of work required for publishing just a small amount of data can be seen as a compensating factor. The case of Beakit was also interesting in the perspective of an application that does not fully embrace semantic technologies such as a triplestore.

6.1.1 Accomplished: Obtained Functionality

With a completed implementation, Beakit now grants access to data concerning film media-items through the Linked Data practices. With the manual mapping approaches, full compliance with the criteria for obtained functionality, except the lack of a SPARQL-endpoint. This means that a media item is linked to other data sources, as well as to Beakit specific data, and is available for web-crawlers and applications through simple HTTP GET requests. As this is done in the standardized fashion of Linked Data, it improves search listings as well as data clarity. Since there is a RDFS vocabulary in place, this dataset should be usable through a semantic browser as well.

The lack of a SPARQL-endpoint however, makes it slightly more awkward to interface with the dataset in an API-like fashion. One has to retrieve the entire dump to be able to search and query the data, something that is generally facilitated in more effective ways in the context of web APIs.

6.1.2 Mapping to Triples

Sadly, with rTriplify not functioning in a Rails 3 environment, there seems to be a lack of a solution that can be simply latched on to an existing application and from there provide access to Linked Data. Looking at the specifications of Triplify, it also appears that it follows the Linked Data practices far from obsessively, and in essence only provides a RDF-dump of relational data and thus does not enable dereferencable URIs, making it inferior to manual mapping in terms of obtained functionality.

Fortunately, the conventions and capabilities of Ruby on Rails make a manually oriented mapping approach, at least for an application that follows these conventions as the case for Beakit, a relatively swift process. Especially so for JSON-LD, where the attempt was
very nearly reduced to simply “latching on” Linked Data functionality because of the JSON interface already in place.

In summary this means that, in terms of complexity and implementation time, even if there was no completely non-intrusive solution available, the application had to undergo very little restructuring for both of the manual mapping approaches. However, as it is the question of manual mapping, these still require knowledge about the domain of triples, serialization formats, and existing vocabularies, increasing their complexity.

### 6.1.3 Ruby on Rails

The main aspects of Ruby on Rails that can be seen as benefiting in the process of publishing Linked Data all orbit around the frameworks RESTfulness, which core idea is giving an ID to every resource. This of course rhymes very well with the most essential of the Linked Data practices. The way that REST is thoroughly integrated into the framework, as defaults essentially, makes it a speedy ordeal to have dereferencable URIs up and running. This in combination with easily implemented server-driven content negotiation makes it easy to respond with a RDF serialization of some sort, and with ontologies disregarded, the process can be reduced to simply inserting predicates between subject and objects.

Even if this manual mapping process can be seen as providing the bare bones of publishing Linked Data, it is only a part of the Linked Data practices, and there are other complicating factors that are not significantly improved by the capabilities of Ruby and Rails. These will be further handled under Shortcuts.

It should also be stated that Ruby on Rails is in no way unique or far superior to the assortment of web frameworks out there in these respects. One can however conclude that the facilities and conventions of Ruby on Rails make publishing Linked Data, in this more manual respect, significantly easier.

### 6.1.4 Other Scenarios

The specifications and requirements of Beakit meant some solutions for publishing Linked Data were discarded. To provide some nuance it will here be briefly handled, when under a different situation, some of these might be applicable.

**RDFa**

In the case of Beakit, because of its client side templating, there will be a discrepancy between the XHTML-representation a user versus a machine experiences, unless the machine can parse javascript. Because of this, RDFa was discarded as a solution to be tested in this investigation. In the scope for other applications however, including other Ruby on Rails applications, it should be noted that RDFa is a quite suitable, more lightweight option for publishing Linked Data that should be taken into account. In relation to the above mentioned manual mapping of relational data to triples, RDFa can be see as the most integrated way of doing this, as one does the mapping in already existing views.
Triplestores, RDF APIs and Non-Ruby Applications

When building a semantic application from scratch or wanting to publish large already existing datasets, the situation one stand before is very different from the case of Beakit. Little can be said based on this investigation for radically different situations such as this.

If using Java one could look at D2RQ, which seems to be the most mature plug and playable approach for mapping relational data to triples and Linked Data. There are, as covered in the background, several other Java-based semantic toolkits and RDF APIs to choose from, the ups and downs of which can not be covered here.

In the case of Beakit, Linked Data was never consumed or used in any way. For consuming and processing Linked Data in Ruby on Rails, via SPARQL or through parsing RDF-dumps, the at the writing moment most well maintained and frequently updated RDF API seems to be RDF.rb, as well as the JSON-LD processor coupled to it.

6.2 Shortcuts: Decreasing Complexity

From this investigation, it is hard to isolate any drastic shortcuts for publishing Linked Data, paths that would dramatically reduce intrusiveness and the need for domain specific knowledge and thus lessen complexity and implementation time. It is not possible to identify any solution where the process is fully automatized, requiring no insight on the technologies of the Semantic Web.

Manual mapping via templating forces developers to tackle the domain of triples and even if the Triplify approach attempts to circumvent this to an extent, if it should have worked, you still have to write a configuration file if you have a custom data model.

However, some simplifications can be done in terms of simply leaving some parts out from the Linked Data practices. Heath & Bizer (4) state:

“The primary means of publishing Linked Data on the Web is by making URIs dereferencable, thereby enabling the follow-your-nose style of data discovery. This should be considered the minimal requirements for Linked Data publishing.”.

As mentioned, not all parts of the Linked Data practices were especially facilitated by Ruby on Rails, but these minimal requirements were. The time consuming process of constructing a Beakit specific ontology was one that was not.

6.2.1 Ontologies and Vocabularies: Leaving Things Out

Of course, since handling triples one will still have to interact with vocabularies to some extent, but an effort should be made to avoid writing your own. Spending time locating as well as using widely adopted ontologies and their terms is a good investment. When absolutely impossible to find and use existing terms, using undeferencable predicate URIs created on the fly can be seen as another pragmatic action. A deviation from the Linked Data practices to be sure, but a relatively rare occurrence, of no huge consequence, and a great reduction of complexity as knowledge about the intricacies of creating ontologies
becomes unnecessary.

Leaving out dataset descriptions, alternative access methods (data dumps etc.) and licensing information, or providing these in some other non-standard format could also be seen as admissible shortcuts if pressed for time, leaving just the absolute essentials of Linked Data in place.

6.3 JSON and the Semantic Web

As an easy way for representing structured data, JSON has quickly risen to prominence. Although less verbose than the well-established XML, its ease of parsing and processing has contributed to its rising popularity. The fact that parsing capabilities are often built into frameworks, as in the case of Ruby on Rails, and the formats javascript friendliness is also a boon, as utilized by backbone.js and in the extension Beakit itself.

However, it is hard to proclaim the death of XML as a format, at least concerning APIs. It’s rather a matter of developers providing access to data in both the formats (19). Frameworks tend to be able to serialize to XML just as well as to JSON. Concerning Linked Data however, and JSON-LD versus RDF/XML, the situation is a bit different.

6.3.1 JSON-LD and RDF

For publishing Linked Data, implicitly RDF in some serialization is suggested for data representation. There are JSON representations for RDF, the Talis specification being the most notable (4) and a standardization being in the works27. However, one has to make a distinction between JSON-LD and RDF/JSON, both being Linked Data formats, but the former only able to be expressed in RDF after transformation through the JSON-LD API.

What can be seen as redeeming for this downside is the fact, as mentioned, that the format requires no restructuring of existing output, making JSON-LD less expressive than RDF/JSON but easier to implement, at least in the more manual mapping scenario. These factors seem to be on the radar of the Linked Data community, and the W3C are discussing perks of JSON-LD in the scope for the further development of JSON and RDF (20).

If then looking at JSON-LD versus RDF/XML in terms of “a solutions complexity and speed of implementation”, the former seems slightly more convenient (and less complex) in the case of Beakit. Because of an existing JSON-interface, the application had to undergo minimal restructuring. The legibility of JSON versus XML also has to be seen as a boon, enabling developers to focus on other things than bulky syntax and the ambiguity of XML.

JSON was, as mentioned, in the context of Beakit also extra useful because of backbone.js. Although this framework is likely to see additional growth, with developers wanting to create more complex user interfaces, it is far from entering any sort of mainstream and being useful in the majority of development scenarios. The format is

27 http://www.w3.org/2010/09/rdf-wg-charter.html
however seeing broad adoption in other applications interesting in relation to web
development and the Semantic Web, namely Web APIs.

6.3.2 APIs and the Semantic Web

It can be hard to see that parties would go out of their way just to publish Linked Data for applications similar to Beakit, as it produces very little of its own data. However, if broadening the scope to include the endeavor of creating a more complete RESTful Web API and wrapping it to serve Linked Data, there are more incentives. There is probably more pertinent data to be published for starters, as well as the fact that the process of setting up the web API involves a lot of the groundwork for publishing Linked Data.

The amount of listings on the API directory Programmable Web indicate a quite significant increase of open web APIs becoming available under the last few years (3). This can be seen as a positive trend, but as Heath & Bizer state, “…while Web APIs make data accessible on the Web, they do not place it truly in the Web…” (4). According to the same listings, 1.8% of these web APIs are semantic and 1% support RDF. On the other hand nearly half support JSON and 67% adopt RESTful protocols (21). With these figures, JSON-LD appears an attractive alternative for making a larger percentage of these APIs part of the Semantic Web because of all its beneficiary traits for wrapping RESTful JSON APIs to serve Linked Data.
7 Conclusions

The best solution for publishing Linked Data in Ruby on Rails in the case of Beakit was manual mapping to RDF/XML templates or the usage of JSON-LD. These were close to identical in terms of obtained functionality. If not strictly adhering to Linked Data principles, JSON-LD has to be seen as the superior alternative, although not by much, as it was the least complex and time consuming to implement due to legibility and its non interfering structure. JSON-LD could also be seen as extra interesting in the context of creating a web API.

Of course, part of the investigation was choosing solutions that fit the case application, Beakit. This meant thinning out some possible solutions that might have been attractive for a Ruby on Rails application under different circumstances. RDFa might be the most notable of these, but one can mention the D2R server as well, although this would have to be run parallel from the Rails application. This investigation can say nothing conclusive about the implementation time or prospects of these, but one can note the possibility of their use in the scope for a Ruby on Rails application.

Even if this investigation was a case study, it still provided some over-bridging conclusions that should be applicable to most scenarios involving Linked Data. The content negotiations capabilities Ruby on Rails, and its inherent RESTfulness were both great benefits to the process of publishing Linked Data, at least if manually mapping to triples, and should be so in most cases. Another shortcut that could be acknowledged in most situations is the fact that handling your own vocabularies and ontologies will for most developers be a time consuming exercise. It is best avoided either by making an extra effort to use existing vocabularies or simply by utilizing non dereferencable predicates.

rTriplify, the alternative that appeared the least intrusive in terms of implementation, did not support Rails 3 and it was deemed to big a task to execute an adaptation for that version. This means that at the writing moment, there are no solutions for simply publishing Linked Data that work in the fashion of “plug-and-play” and that are specific to Ruby on Rails.

This probably means that parties with small datasets probably would not publish data in most commercial scenarios, at least not according to Linked Data principles, because it entails to much going out of their way. However, if in the context of creating a RESTful web API, which seems to be becoming an increasingly common occurrence, it becomes more likely as much groundwork for this process overlaps the endeavor of publishing Linked Data. In a situation such as this, Ruby on Rails and JSON-LD seem like great choices.

7.1 Recommendations

For publishing Linked Data in Ruby on Rails the speediest route would in most cases be, if adhering to REST, manually mapping relational content to JSON-LD. Note that this may not be true depending on the nature of the application and the structure of the data. To further simplify this process, avoid creating your own ontologies unless absolutely necessary.
7.2 Future Work

A natural step for a continuation of the theme for this investigation would be looking into publishing larger datasets as well looking at Ruby on Rails in the scope of building a more fully semantic application. Creating a more extensive Ruby on Rails web API that adheres to Linked Data practices, and possibly looking at measures for transforming already existing ones, could also give interesting insights on whether the approaches tested here are applicable to that scenario.

Finally, a study with a more quantitative method and with a aim of benchmarking more technical aspects of the approaches tested here, would give valuable insight on possible performance issues of these vis-à-vis more established semantic web toolkits.
References


2. Berners-Lee T. Linked Data - Design Issues [Internet]. 2006; Available at: http://www.w3.org/DesignIssues/LinkedData

3. DuVander A. 5,000 APIs: Facebook, Google and Twitter are Changing the Web [Internet]. 2012; Available at: http://blog.programmableweb.com/2012/02/06/5000-apis-facebook-google-and-twitter-are-changing-the-web/ [accessed May 2012]


6. Hausenblas M, Cyganiak R. Publishing and consuming linked data with RDFa [Internet]. 2009; Available at: http://ld2sd.deri.org/ldod-ng-tutorial/


20. SW Coordination Group Telco [Internet]. 2010; Available at: http://www.w3.org/2010/11/swcq-minutes.html#item03 [accessed April 2012]

9 Glossary

**RDF (Resource Description Framework)** – please see 2.1.1 RDF - Resource Description Framework.

**SPARQL (SPARQL Protocol and RDF Query Language)** – please see 2.1.2 Using Linked Data.

**URI (Uniform Resource Identifier)** - A URI is a more general incarnation of a URL and describes and identifies a resource for example: [http://www.site.com/animals/snake/1](http://www.site.com/animals/snake/1). A dereferencable URI is then simply a URI that can be dereferenced, i.e the resource in question is returned by the web server after a request.

**API (Application Programming Interface)** – An API is an interface for communication between services or software elements. Functions in the API are bound to certain actions that can be a method call or similar, according to the API specifications. These specifications are not only for functions, but for data structures, objects and variables. A RDF API is here a library for interfacing with RDF-stores, i.e reading RDF, or for writing/outputting RDF in various serializations.

**MVC (Model-View-Controller)** – The MVC design pattern consists of the separation of data and user interaction. A model holds data and its structure, a view display different interactions and data from the model, and a controller delegates interaction between the view and the model.

**REST (Representational State Transfer)** – A web service design model that, in short, revolves around the principles of assigning a specific URI for each application resource, that each resource has a global interface for delegating actions between client and server via standard HTTP requests (POST, GET, PUT, DELETE), and the fact that each resource can be requested and obtained in different formats (HTML, XML, JSON etc.).

**CRUD (Create Read Update Destroy)** – A reference to all the major actions used on data models in a relational application. Each action is mapped to a SQL-query and HTTP request: INSERT/POST, SELECT/GET, UPDATE/PUT, DELETE/DELETE.
10 Appendices

All references to www.beakit.com should be accessed as watchlist-stg.herokuapp.com as the thesis code branch has not, as of this writing moment, been merged into the production code base.

10.1 Appendix 1

Active Record schema after implementation. Note that “Beakit item” is here named “Watchlist item”.

ActiveRecord::Schema.define(:version => 20120229154732) do
  create_table "affiliate_links", :force => true do |t|
    t.integer  "affiliate_type"
    t.string   "url"  
    t.integer  "media_item_id"
    t.datetime "created_at"
    t.datetime "updated_at"
  end
  add_index "affiliate_links", ["media_item_id"], :name => "index_affiliate_links_on_media_item_id"

  create_table "companies", :force => true do |t|
    t.string   "fid"
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "countries", :force => true do |t|
    t.string   "fid"
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "delayed_jobs", :force => true do |t|
    t.integer  "priority", :default => 0
    t.integer  "attempts", :default => 0
    t.text     "handler"
    t.text     "last_error"
    t.datetime "run_at"
    t.datetime "locked_at"
    t.datetime "failed_at"
    t.string   "locked_by"
    t.string   "queue"
    t.datetime "created_at"
    t.datetime "updated_at"
  end
  add_index "delayed_jobs", ["priority", "run_at"], :name => "delayed_jobs_priority"

  create_table "directors", :force => true do |t|
    t.integer  "person_id"
    t.integer  "media_item_id"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "example_list_items", :force => true do |t|
    t.string   "item"
    t.datetime "created_at"
    t.datetime "updated_at"
create_table "genre_belongings", :force => true do |t|
  t.integer  "genre_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "genres", :force => true do |t|
  t.string   "fid"
  t.string   "name"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "languages", :force => true do |t|
  t.string   "fid"
  t.string   "name"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "locations", :force => true do |t|
  t.string   "fid"
  t.string   "name"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "media_items", :force => true do |t|
  t.text     "description"
  t.string   "name"
  t.integer  "media_type"
  t.string   "source_id"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.string   "cover_file_name"
  t.string   "cover_content_type"
  t.integer  "cover_file_size"
  t.datetime "cover_updated_at"
  t.date     "publication_date"
  t.string   "image_url"
  t.integer  "runtime"
  t.string   "fid"
  t.datetime "freebase_lookup_failed_at"
end

create_table "origins", :force => true do |t|
  t.integer  "country_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "people", :force => true do |t|
  t.string   "fid"
  t.string   "name"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "performances", :force => true do |t|
  t.integer  "person_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "producing", :force => true do |t|
  t.integer  "person_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "production_companies", :force => true do |t|
  t.integer  "company_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "spoken_languages", :force => true do |t|
  t.integer  "language_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "takes_places", :force => true do |t|
  t.integer  "location_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "users", :force => true do |t|
  t.string   "email",                                 :default => "", :null => false
  t.string   "encrypted_password",     :limit => 128, :default => "", :null => false
  t.string   "reset_password_token"
  t.datetime "reset_password_sent_at"
  t.datetime "remember_created_at"
  t.integer  "sign_in_count",                           :default => 0
  t.datetime "current_sign_in_at"
  t.datetime "last_sign_in_at"
  t.string   "current_sign_in_ip"
  t.string   "last_sign_in_ip"
  t.integer  "created_at"
  t.datetime "updated_at"
end

add_index "users", ["email"], :name => "index_users_on_email", :unique => true
add_index "users", ["reset_password_token"], :name => "index_users_on_reset_password_token", :unique => true

create_table "watchlist_items", :force => true do |t|
  t.string   "raw_input"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.integer  "user_id"
  t.integer  "media_type"
  t.date     "failed_to_match_at"
  t.datetime "processed_at"
  t.integer  "media_item_id"
  t.integer  "rated"
  t.datetime "deleted_at"
  t.datetime "ticked_off_at"
end

create_table "writers", :force => true do |t|
  t.integer  "person_id"
  t.integer  "media_item_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end
10.2 Appendix 2

http://www.beakit.com/vocab

vocabulary.rdf.erb

<?xml version="1.0"?>

<rdf:RDF

xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:rdfs="http://www.w3.org/TR/WD-rdf-schema#"

xmlns:owl="http://www.w3.org/2002/07/owl#"

xmlns:dcterms="http://purl.org/dc/terms/"

xmlns:cc="http://creativecommons.org/licenses/by-sa/3.0/"

xmlns="http://<%= ENV['domain'] %>/vocab#">

<rdf:Description>

  <rdf:type rdf:resource="owl:Ontology"/>

  <rdfs:label>Beakit Vocabulary</rdfs:label>

  <dcterms:publisher

  rdf:resource="http://www.prototyp.se/company.rdf#prototyp"/>

  <dcterms:date rdf:datatype="http://www.w3.org/2001/XMLSchema#date">2012-05-07</dcterms:date>

  <dcterms:license

  rdf:resource="http://creativecommons.org/licenses/by/3.0/"/>

</rdf:Description>

<rdf:Description rdf:ID="Item">

  <rdf:type rdf:resource="rdfs:Class"/>

  <rdfs:label>Beakit Item</rdfs:label>

  <rdfs:comment>A Beakit Item which in turn is a media item</rdfs:comment>

</rdf:Description>
<rdf:Property rdf:ID="name">
    <rdf:type rdf:resource="owl:DatatypeProperty"/>
    <rdfs:subPropertyOf rdf:resource="rdfs:label"/>
    <rdfs:domain rdf:resource="#Item"></rdfs:domain>
    <rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
</rdf:Property>

<rdf:Property rdf:ID="onAmountOfLists">
    <rdf:type rdf:resource="owl:DatatypeProperty"/>
    <rdfs:domain rdf:resource="#Item"></rdfs:domain>
    <rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
    <rdfs:comment>The amount of Beakit lists this media item is on</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:ID="averageBeakitRating">
    <rdf:type rdf:resource="owl:DatatypeProperty"/>
    <rdfs:domain rdf:resource="#Item"></rdfs:domain>
    <rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
    <rdfs:comment>The average Beakit rating (1.0-5.0) by Beakit users</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:ID="amazonAffiliationLink">
    <rdf:type rdf:resource="owl:DatatypeProperty"/>
    <rdfs:domain rdf:resource="#Item"></rdfs:domain>
    <rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
</rdf:Property>

<rdf:Property rdf:ID="itunesAffiliationLink">
    <rdf:type rdf:resource="owl:DatatypeProperty"/>
    <rdfs:domain rdf:resource="#Item"></rdfs:domain>
    <rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
</rdf:Property>
<rdf:type rdf:resource="owl:DatatypeProperty"/>
<rdfs:domain rdf:resource="#Item"></rdfs:domain>
<rdfs:range rdf:resource="rdfs:Literal"></rdfs:range>
</rdf:Property>

</rdf:RDF>
10.3 Appendix 3

application.rdf.erb

```xml
<?xml version="1.0" encoding="utf-8"?>

<rdf:RDF xmlns:rdf      = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs     = "http://www.w3.org/2000/01/rdf-schema#"
     xmlns:foaf     = "http://xmlns.com/foaf/0.1/"
     xmlns:dc       = "http://purl.org/dc/elements/1.1/"
     xmlns:dcterms  = "http://purl.org/dc/terms/"
     xmlns:void     = "http://rdfs.org/ns/void#"
     xmlns:fb       = "http://rdf.freebase.com/ns/"
     xmlns:owl      = "http://www.w3.org/2002/07/owl#"
     xmlns:beakit   = "http://<%= ENV['domain'] %>/vocab#"  
 <%= yield :namespaces %>>

 <%= yield %>

</rdf:RDF>
```
10.4 Appendix 4

http://www.beakit.com/media_item/66.rdf

sample RDF-output for media item with internal ID of 66

<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf      = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs     = "http://www.w3.org/2000/01/rdf-schema#"
         xmlns:foaf     = "http://xmlns.com/foaf/0.1/"
         xmlns:dc       = "http://purl.org/dc/elements/1.1/"
         xmlns:dcterms = "http://purl.org/dc/terms/"
         xmlns:beakit   = "http://beakit.com/vocab#"
         xmlns:void    = "http://rdfs.org/ns/void#"
         xmlns:fb      = "http://rdf.freebase.com/ns/"
         xmlns:owl      = "http://www.w3.org/2002/07/owl#"
         xmlns:beakit   = "http://beakit.com/vocab#"
>
  <rdf:Description rdf:about="http://beakit.com/media_item/66">
    <rdfs:label>Description of the beakit item "Snakes on a Plane"</rdfs:label>
    <dcterms:created
datatype="http://www.w3.org/2001/XMLSchema#dateTime">2012-03-01 13:09:03 UTC</dcterms:created>
    <dcterms:modified
datatype="http://www.w3.org/2001/XMLSchema#dateTime">2012-03-01 15:38:41 UTC</dcterms:modified>
    <dcterms:isPartOf>http://beakit.com/datasets/media_items</dcterms:isPartOf>
    <dcterms:publisher
resource="http://www.prototyp.se/company.rdf#prototyp"/>
    <dcterms:license resource="http://creativecommons.org/licenses/by/3.0/"/>
    <rdf:type resource="beakit:Item"/>
    <beakit:name
datatype="http://www.w3.org/2001/XMLSchema#string">Snakes on a Plane</beakit:name>
    <owl:sameAs
resource="http://rdf.freebase.com/ns/en.snakes_on_a_plane"/>
  </rdf:Description>
</rdf:RDF>


</rdf:Description>

</rdf:RDF>
10.5 Appendix 5
modified show method in media_item_controller.rb

```ruby
if params[:format]

coder = HTMLEntities.new

@item = MediaItem.find_by_id(params[:id])

respond_to do |format|

  format.json { render :json => mergeTables(@item,coder) }

  format.jsonld { render :json => mergeTables(@items,coder) }

  format.rdf { render "show" }

end

else

respond_to do |format|

  format.html { redirect_to :status=>303, :controller=>'media_item', :action=>'show', :id=>params[:id], :format=>'json' }

  format.rdf  { redirect_to :status=>303, :controller=>'media_item', :action=>'show', :id=>params[:id], :format=>'rdf' }

  format.json  { redirect_to :status=>303, :controller=>'media_item', :action=>'show', :id=>params[:id], :format=>'json' }

  format.jsonld  { redirect_to :status=>303, :controller=>'media_item', :action=>'show', :id=>params[:id], :format=>'jsonld' }

end

end
```
10.6 Appendix 6

http://www.beakit.com/media_item/66.json or
http://www.beakit.com/media_item/66.jsonld

sample JSON-LD output for media item with internal ID of 66

{
  @context: {
    name: "http://beakit.com/vocab#name",
    freebase: "http://www.w3.org/2002/07/owl#sameAs",
    created: "http://purl.org/dc/terms/created",
    updated: "http://purl.org/dc/terms/modified",
    set: "http://purl.org/dc/terms/isPartOf",
    publisher: "http://purl.org/dc/terms/publisher",
    license: "http://purl.org/dc/terms/license",
    label: "http://www.w3.org/2000/01/rdf-schema#label",
    onAmountofLists: "http://beakit.com/vocab#onAmountofLists",
    amazonAffiliationLink: "http://beakit.com/vocab#amazonAffiliationLink",
    itunesAffiliationLink: "http://beakit.com/vocab#itunesAffiliationLink"
  },
  @id: "http://beakit.com/media_item/66",
  @type: [
    "http://beakit.com/vocab#Item",
    "http://rdf.freebase.com/ns/film.film"
  ],
  label: "Description of the beakit item 'Snakes on a Plane'",
  name: "Snakes on a Plane",
  created: "2012-03-01T13:09:03Z",
}
An FBI agent takes on a plane full of deadly and poisonous snakes, deliberately released to kill a witness being flown from Honolulu to Los Angeles to testify against a mob boss.
directors: [
  "David R. Ellis"
],

producers: [
  "Craig Berenson",
  "Don Granger",
  "Gary Levinsohn"
],

writers: [
  "John Heffernan",
  "Sebastian Gutierrez"
],

genres: [
  "Thriller",
  "Action",
  "Comedy",
  "Airplanes and airports",
  "Horror",
  "Natural horror films",
  "Action/Adventure",
  "Action Thrillers",
  "Horror Comedy"
],

production_companies: [
  "New Line Cinema",
  "Mutual Film"
],
languages: [}
"English Language",
],
locations: [
  "Hawaii"
],
origins: [
  "United States of America",
  "Germany",
  "Canada"
]
Appendix 7

http://www.beakit.com/sitemap.xml

sitemap.xml, served at /sitemap.xml

<?xml version="1.0" encoding="UTF-8"?>

<urlset
    xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
>
    <sc:dataset>
        <sc:datasetLabel>Beakit Media Items Data Set</sc:datasetLabel>
        <sc:datasetURI>http://<%= ENV['domain']%>/datasets/media_items</sc:datasetURI>
        <sc:linkedDataPrefix>http://<%= ENV['domain']%>/media_item/</sc:linkedDataPrefix>
        <sc:sampleURI>http://<%= ENV['domain']%>/media_item/120</sc:sampleURI>
        <sc:dataDumpLocation>http://<%= ENV['domain']%>/dumps/media_items.rdf</sc:dataDumpLocation>
        <sc:dataDumpLocation>http://<%= ENV['domain']%>/dumps/media_items.jsonld</sc:dataDumpLocation>
        <changefreq>weekly</changefreq>
    </sc:dataset>
</urlset>