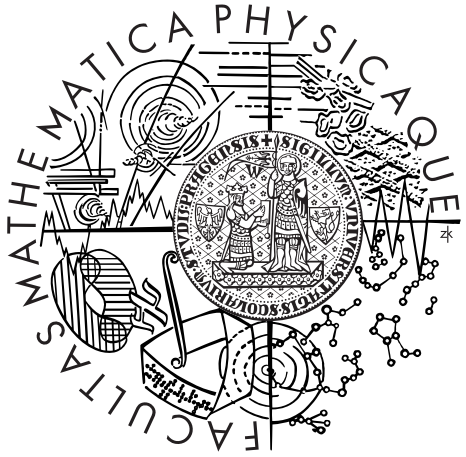
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Vojtěch Vondra

Webová aplikace pro analýzu a vizualizaci legislativního procesu postavená na principech Linked Data

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.

Studijní program: Informatika

Studijní obor: Programování

Praha 2014

I wish to thank my supervisor, Martin Nečaský, Ph.D., for his introduction to the world of Linked Data and for his advice during the development of the ontology and dataset. I appreciate the effort made by the developers of Payola to make Linked Data more accessible to the public.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 21.5.2014

Název práce: Webová aplikace pro analýzu a vizualizaci legislativního procesu postavená na principech Linked Data

Autor: Vojtěch Vondra

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt: Cílem této práce je vytvořit sémantickou databázi návrhů zákonů schvalovaných v českém parlamentu. Pro popis legislativního procesu je vytvořena ontologie popisující jednotlivé kroky schvalování a stavy návrhů zákonů. Databáze i ontologie používá RDF jako nástroj pro sémantizaci dat poskytovaných Poslaneckou sněmovnou. Práce využívá existující ontologie, např. FRBR, pro reprezentaci dat s užším významem. Výsledná databáze bude prezentována pomocí open-source nástroje pro správu Linked Data, Payola, do kterého budou vytvořeny pluginy pro vizualizaci dat pomocí technologií HTML5.

Klíčová slova: Linked Data, sémantický web, RDF, ontologie, legislativa, HTML5

Title: Web application for legislative process analysis and visualization built on top of Linked Data principles

Author: Vojtěch Vondra

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D.

Abstract: The aim of this thesis is to create a new dataset containing bills being passed in the Czech Parliament. It creates an ontology describing the legislative process and the individual stages of passing the bill. Both the dataset and ontology will use RDF to semanticize public exports provided by the Czech Chamber of Deputies. To create the ontology, it is attempted to specialize existing ontologies such as FRBR to describe data with a narrower domain. Resources representing bills are linked to other ontologies to connect the dataset into the Linked Data cloud. For practical presentation of the created dataset, new visualization plugins are programmed into Payola, an open-source Linked Data management tool, using HTML5 technologies.

Keywords: Linked Data, Semantic Web, RDF, Ontology, legislation, HTML5

Contents

Introduction

Laws affect our lives and behavior every day. Their codification defines our rights and our responsibilities. At all times, we must adhere to them and take them into account during our decision making. Laws can also be considered a dataset, easily classified, structured, searchable, changing in time. Creating a dataset and an ontology for bills, which are predecessors of laws, is the goal of this thesis.

In recent years, making government data accessible to the public has become a major topic. Especially in the Czech Republic we see a rise in public participation and an interest in transparency of official institutions. Benefits of the movement are easy to find, more transparency means more accountability and an increase in public participation reduces potential for corruption and brings new topics into discussion. Creating a datasource for the legislative process can benefit both the private and public sector.

In order to make the ontology usable and effective I will try to connect the new data to existing ontologies which are being worked on in the Department of Software Engineering of the Faculty of Mathematics and Physics. To promote usage of the ontology, I will create visualizations of selected metrics and statistics. To create them, I will use Payola, an open-source tool for analyzing and visualizing RDF data. A side-product of this thesis should be visualization plugins usable for any Linked Data dataset.

In the first part of the thesis I would like to present Linked Data which is chosen as the platform for publishing the legislative process. Connecting the data with semantic relationships is crucial and we will describe various tools which are available to create them in Linked Data and RDF, the underlying technology. Technologies used to create the ontology and dataset will be introduced. In the second chapter, the scope of the dataset will be defined and individual entities will be presented. Once the scope has been defined, the third chapter will describe how the ontology was created and how all entities were mapped into Linked Data. The fourth chapter is developer documentation for the converter application used to parse the provided exports from the Chamber of Deputies website into RDF. In the fifth and last chapter created visualizations will be presented.

1. Technologies

In this chapter, the goals and fundamental principles of Linked Data which is chosen as the method for publishing legislative process data will be described. Linked Data is a broader concept and technologies such as RDF on which it relies will be explained on a level which is necessary to understand how the data will be stored and structured.

1.1 Semantic Web

One of the main sources of information consumed and reused on an everyday basis is the World Wide Web accessed through the Internet. The elementary building blocks of the Web are individual Web pages and documents which are connected together with hyperlinks. These links can be used to point to related topics or transclude other documents using a simple reference. There are no limitations how much or how little data individual documents can hold. Together, documents with links make the Web a decentralized platform continuously extended and revised in a worldwide collaborative effort. Web documents and links together form a graph traversable by both human beings using Web browsers and machines which can follow and download the links found in documents.

However, documents and hyperlinks based on the original WWW proposal[4] and used at large today are usually unstructured and omit much of their semantic value which makes automated processing of their relations and meaning difficult. The World Wide Web Consortium (W3C), led by the creator of the World Wide Web, Tim Berners-Lee, has addressed this problem in the Semantic Web activity. From December 2013, Semantic Web is now part of W3C's *Data Activity*[1]. One of the solutions is to extend current languages used for representing Web documents and add structure and semantic information to already published data. The enriched resources and links are called Linked Data, the decentralized graph they form is called the Linked Data cloud[2].

1.1.1 Extending the current web

Documents on the hypertext Web are written in HTML (Hypertext Markup Language) and use untyped hyperlinks represented by the HTML anchor (`<a>`)

[1] `http://www.w3.org/blog/data/2013/12/13/welcome/`
[2] `http://lod-cloud.net/`

element for connections. HTML by itself does not provide a way to describe the type or contents of the document in a standardized way. The same applies for the semantics of hyperlinks. There are many conventions or unstandardized formats which can be used, but usually, they solve a specific use-case, e.g. enriching results from search engines.

One of the multiple attempts to extend HTML, so it can convey this metadata, are *microformats* which target common components of documents, e.g. contact information, event details, product information on e-commerce websites etc. In Example 1.1 a business card type element on a Web page enriched with microformats is shown.

```
<ul class="vcard">
    <li class="fn">John Student</li>
    <li class="org">MFF</li>
    <li class="tel">604-555-1234</li>
</ul>
```

Example 1.1: An example of HTML markup enriched with the hCard microformat

Microformats are not the work of a standards body, but exist thanks to large Web service providers which pledge support for the format (Google, Microsoft, and Yahoo in this case [3]). They are useful for identifying meaningful snippets of content containing structured data, but fail to describe the documents as a whole and address the hyperlink issue.

Similar solutions exist and usually solve a single use-case or issue. HTML documents can contain elements in their header called *meta tags*. Common meta tags contain keywords, descriptions, author information, and similar information. Large media providers such as Google or Facebook create proprietary formats[3,4] to specify metadata which they are able to display. These approaches fail to achieve the necessary universality and granularity to describe arbitrary documents. Linked Data tries to approach this issue.

1.1.2 Linked Data

Web pages on the Internet can be identified and accessed by their Uniform Resource Identifier (URI). Linked Data promotes the idea to give all individual resources unique URIs down to the smallest parts of web page. The selected URIs should be available through HTTP, so they can be accessed by the wide

[3]OpenGraph for social data `https://developers.facebook.com/docs/opengraph/`

[4]Google+ Author profiles `https://support.google.com/webmasters/answer/2539557?hl=en`

public. Availability on the web is still not enough though, an open license allowing usage and modification should be chosen for the data. Now, when the user or machine can access the documents, it is preferable to choose a machine-readable standardized format and link to other documents so the user can learn more related information. Tim Berners-Lee has summarized this in four rules in his article about designing Linked Data[1].

In the aforementioned documents Tim Berners-Lee introduces a quick five-star test, shown in Table 1.1, which gives a quick glance of how well published data conforms to Linked Data principles.

⋆	Available on the web (whatever format) but with an open licence, to be Open Data
⋆⋆	Available as machine-readable structured data (e.g. excel instead of image scan of a table)
⋆⋆⋆	as (2) plus non-proprietary format (e.g. CSV instead of excel)
⋆⋆⋆⋆	All the above plus, Use open standards from W3C (RDF and SPARQL) to identify things, so that people can point at your stuff
⋆⋆⋆⋆⋆	All the above, plus: Link your data to other people's data to provide context

Table 1.1: Five-star test for Linked Data, source: `http://www.w3.org/DesignIssues/LinkedData.html`

Even though the beginning of this chapter uses conventional Web pages written in HTML as examples where semantic data can appear, the only limiting common foundational block should be the Web and HTTP as the transport layer. Bills collected and published as Linked Data in this thesis will exist in the RDF format which will be explained in the next chapter and a HTML version of the source data will be an application of the data for end users.

1.2 RDF

The Resource Description Framework is a W3C technology intended to represent information about Web resources and is the primary technology used to create Linked Data. RDF copies the nature of the Web by creating a graph-based data model. It solves deficiencies mentioned in Chapter 1.1 by adding data types, semantic links between documents and unambiguous URI identifiers of resources. Information stored with RDF is intended to be read by computers.

There are various formats and programming languages in which RDF can be represented. All of the formats follow the abstract model shown in Figure 1.1.

Every object in the graph data model is a RDF resource identified by a URI or a literal value. The triple is often called a *statement* or *arc*.[10]

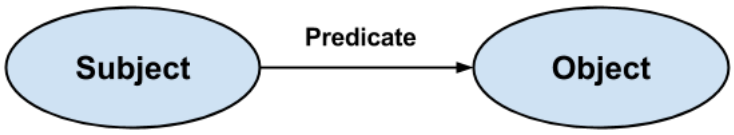


Figure 1.1: RDF triple

- **Subject**, which is the RDF resource being described.
- **Predicate**, which denotes a trait or aspect of the subject and is a RDF resource itself.
- **Object** is another RDF resource or literal[5] which is the value of the Predicate.

RDF can be incorporated into a new technology by specifying a syntax which is translatable into the graph model. The most common formats for storing and representing RDF data will be described in Chapter 1.2.1.

The subject can be part of another triple as the predicate value. A subject will typically have multiple properties as shown in Figure 1.2 and adding the same property multiple times is also possible. Properties in the form of resources will become useful later on during defining of schemas and ontologies. As an example, the `firstName` resource in the figure should be a `FunctionalProperty` (having a unique value) and its `Domain` should be a resource whose type is `Person`. `Person`, `Domain` and `FunctionalProperty` would all be other RDF resources with defined meaning.

In the world of Linked Data, anyone can say anything about a resource. This allows extending existing descriptions of documents and enables Linked Data to work on a global scale[11]. There is no authority which would control what can be published or which would make any editorial changes. The drawbacks of such an approach appear immediately. When designing applications working with RDF data, care must be taken to account for inconsistencies, incorrect or false data, or incomplete information.

[5]for some values, e.g. dates or numerical values, it is more convenient to represent them in their original form instead of referencing them by a URI. Literals can be typed with the ^^operator, e.g. "2014-05-05"^^xsd:date.

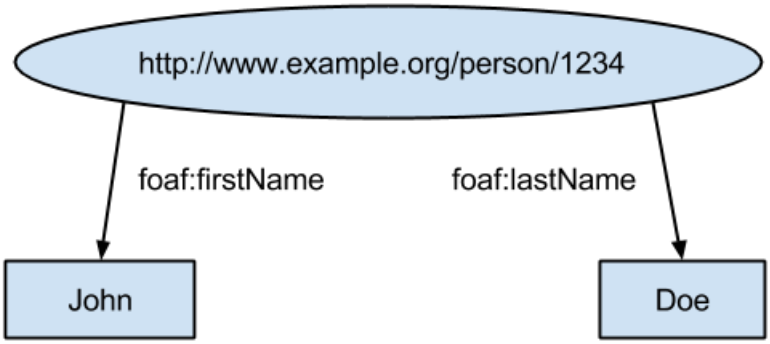


Figure 1.2: Defining multiple properties of a single resources

Namespaces

RDF uses URIs to identify resources. Resources and properties of a common topic typically reside in a *namespace*. In the world of RDF, this means they share the same prefix in their URI. In the following chapters, several RDF vocabularies will be introduced and in each one of them their resources have the same URI prefix. All resources in the core RDF specification are in the following namespace: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, often abbreviated `rdf:`. Namespaces are language constructs of different RDF storage formats, but in the most common formats as RDF/XML or N3, the common URI prefix is simply replaced by an arbitrary identifier and when using it, a colon is inserted between the identifier and the suffix. The URI is then resolved by substituting the identifier with the prefix and appending the suffix as shown in Example 1.2. This follows the same rules as QNames in XML.[12]

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix lb:
   <http://linked.opendata.cz/ontology/legislation/bill#> .

<http://linked.opendata.cz/resource/legislation/cz/bill/2012/758>
        a lb:Bill ;
        dcterms:identifier "2012/758" ;
        lb:introductionDate "2012-07-26" ;
        dcterms:title "Novela z. o pojistovnictvi" .
```

Example 1.2: RDF namespaces in the TTL format

1.2.1 RDF serialization

The general concept of the graph model has been now explained. In order to use RDF data in the real world, a storage format must be chosen. Data then can

be transferred from memory representations into files or streams which can be sent over the Internet. The act of persistently storing the graph model is called *serialization*.

There are two common formats used, RDF/XML and N3. RDF/XML uses XML, a wide spread Web technology with a rich ecosystem of documentation, development tools, editors, and parsers. XML is notorious for its complexity and N3 on the other hand is easier to write by hand and more comprehensible for reading by humans, while still preserving machine readability.

Converters exist between RDF/XML and N3 (and its variants) and tools which hold the graph in memory often provide methods to export to multiple formats. Persistent storage backends with the capability of serializing RDF statements are called *triplestores*.

RDF/XML

XML is a general purpose markup language for storing and transferring data. It is a both human and machine readable text format with Unicode support. Many protocols rely on XML as the format used for transport.[6] Arbitrary data structures can be converted into XML and back. A well-formed XML *document* consists of a tree of XML *elements*. Those start and end with a *tag* and can contain multiple XML elements as children or text content as shown in Example 1.3. Elements can have additional properties such as `author` in the figure called *attributes*.

```
<books>
    <book author="Herman␣Melville">Moby Dick</book>
    <book author="Mark␣Twain">Adventures of Huckleberry
       Finn</book>
</books>
```

Example 1.3: Sample XML document

RDF/XML is a syntax for writing XML which transforms the RDF graph into a tree structured XML document.[13] RDF graphs can be more complex than trees by having circular references, multiple parents, or multiple edges. A graph can be decomposed into a number of paths. Since the RDF graphs are composed of subject-predicate-object triples, the paths always contain an alternating sequence of nodes and properties. In Example 1.4 we see an example of

[6] e.g. the Extensible Messaging and Presence Protocol (XMPP) used for instant messaging or RSS and Atom used for syndicating content on blogs or news websites.

a resource representing a deputy in the Chamber of Deputies which is the subject of three RDF statements: his first and last name, whose values are RDF literals, and membership of a political party. The party is another RDF resource serialized deeper in the document. The party is a subclass of `foaf:Group` which is defined in the FOAF RDF vocabulary described in chapter 1.3.4. Instead of repeating the definition of the `foaf:Group` resource, only a reference by its URI is made.

```
<rdf:Description
   rdf:about="http://ld.opendata.cz/resource/psp.cz/person/237">
    <foaf:firstName>Bohuslav</foaf:firstName>
    <foaf:lastName>Sobotka</foaf:lastName>
    <foaf:member>
        <rdf:Description rdf:about
          ="http://ld.opendata.cz/resource/psp.cz/group/CSSD">
            <foaf:name>Česká strana sociálně
               demokratická</foaf:name>
            <dcterms:identifier>CSSD</dcterms:identifier>
            <rdf:type
               rdf:resource="http://xmlns.com/foaf/0.1/Group"/>
        </rdf:Description>
    </foaf:member>
</rdf:Description>
```

Example 1.4: RDF/XML serialized fragment (without root element with namespaces)

Another possible abbreviation in RDF/XML is listing objects which share the same subject and predicate next to each other. In Figure 1.4, if the Deputy was a member of another group or organ, the organ's resource could be present under the `foaf:member` element. By traversing all non-overlapping paths in the RDF graph we obtain its RDF/XML serialization. The representation in XML is not unambiguous, multiple serialized documents can represent the same graph.

N3 and Turtle

N3 is a non-XML RDF serialization format and a assertion and logic language as well. Its notation is much shorter than RDF/XML and human readability was taken into account during its design.[7] Tim Berners-Lee, one of the major opinion leaders in the Semantic Web and Linked Data movement, works on its development. N3 contains syntactic sugar for common idioms to ease writing and to group related information.[14] N3 goes beyond capabilities of RDF in representing data and Turtle is the RDF-only subset of N3. Any Turtle serialization is

[7] `http://www.w3.org/TeamSubmission/2008/SUBM-n3-20080114/#intro`

valid N3.[15] Turtle is chosen as the format in which the Bill ontology is written for its friendliness when written by hand and expressiveness when being read.

To re-use the example from the description of RDF/XML, the same entity represented in Turtle is shown in Example 1.5. Further examples in subsequent chapters will use Turtle for legibility and consistency.

```
@prefix foaf:     <http://xmlns.com/foaf/0.1/> .
@prefix dcterms:       <http://purl.org/dc/terms/> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://linked.opendata.cz/resource/psp.cz/person/237>
    foaf:firstName "Bohuslav" ;
    foaf:lastName "Sobotka" ;
    foaf:member
       <http://linked.opendata.cz/resource/psp.cz/group/153> .

<http://linked.opendata.cz/resource/psp.cz/group/CSSD>
    a foaf:Group ;
    dcterms:identifier "CSSD" ;
    foaf:name "Ceska strana socialne demokraticka" .
```

Example 1.5: Turtle (N3) serialization

1.2.2 RDF Schema

Until now, RDF was described as a very loose way of representing data where structure is at the discretion of the creator. Structuring and classifying data is useful for obvious reasons, it makes querying and updating of batches easier. It also allows to set expectations about resources and make assertions based on their type.[16]

RDF Schema, often referred to as RDFS or RDF(S), introduces resources which are similar to terms known from the object oriented programming paradigm, examples are shown in Table 1.2. RDF resources become instances of classes which are resources themselves. In RDF statements, predicates are resources called properties which have a range and domain. Both classes and properties can have subclasses and subproperties which narrow their scope.

Many components of RDF Schema were included and used in the more complex Web Ontology Language (OWL) which brings more powerful schematic and semantic features.

Classes used in RDF Schema	
rdfs:Resource	is the universal class, every resource in RDF is of this type
rdfs:Class	is used to create custom classes for other resources
rdfs:Literal	is a literal string or integer value, it can have an additional datatype
rdf:Property	is used to create property classes
Properties used in RDF Schema	
rdfs:subClassOf	is used to organize class hierarchy
rdf:type	is used to denote a resource is an instance of a class
rdfs:domain	is assigned to rdf:Property instances and specifies the subjects which can use it
rdfs:range	is assigned to rdf:Property instances and specifies the allowed values when using the property as a predicate

Table 1.2: Examples of RDF Schema vocabulary, not exhaustive

1.3 Web ontologies

Generally, ontologies in computer science are declarative definitions of knowledge about a certain domain. Ontologies define known subjects (individuals, such as a person or object), both concrete and abstract, list known properties of subjects and possible relationships between them. In addition, ontologies can define constraints on these relations and properties.[20]

In Linked Data, with ontologies we can require RDF resources to have properties (even with a specific data type or belonging to a selected domain) by giving them a class and defining known subject-property-object triples for that class. Using this, we can also require RDF resources to connect to other resources, encouraging extending of existing RDF datasets. On any dataset described by an ontology *reasoners* can be used to infer additional statements not defined explicitly. One of the main goals of this thesis is to create an ontology for bills passed in the Parliament and link it to ontologies describing legal documents including laws.

In the following sections, several established ontologies will be described. Most of them are general purpose and can be used to create more specialized ones with finer constraints and domain and range limitations. An example is the FRBR ontology, which will be used to represent the process of passing a bill as a document.

1.3.1 OWL

The OWL Web Ontology Language is a project endorsed by the W3C which brings vocabulary necessary to implement ontologies for RDF data.

OWL extends RDFS and introduces concepts well-known from object-oriented programming like classes and subclasses. When describing objects with RDF it is useful to define more abstract classes and assign common properties to them, more specialized subclasses will only have properties which are appropriate just for them.

OWL uses vocabulary from RDFS and adds more complex restrictions and semantic value. To give an example, in RDFS we could define two classes, `Cat` and `Dog`, both subclasses of `Animal`. Nothing prevented us from creating a resource, which would be an instance of both as shown in Example 1.6.

OWL has several variants, OWL Lite, OWL DL and OWL Full. Each sublanguage is a valid subset of the next one. OWL Lite provides features for class hierarchy support and simple constraints, OWL DL adds more semantic expressiveness while maintaining decidability[8] and OWL Full gives syntactic freedom which does not guarantee decidability.[20] For example, in OWL Full a class can be also treated as an instance or a collection of instances.[9]

```
@prefix ex: <http://www.example.org> .
<ex:catdog>
    dcterms:title "CatDog" ;
    rdf:type <ex:Cat> ;
    rdf:type <ex:Dog> .
```

Example 1.6: A resource which is an instance of two disjoint classes

With OWL, we can use the `owl:disjointWith` property to denote that something can be either a Cat or a Dog, but not both at once. OWL also provides support for property cardinality description, versioning, annotation or set operations.[10] In this section, those used in the bill ontology will be explained. In the examples below, `owl:` refers to the `http://www.w3.org/2002/07/owl#` namespace.

Classes

First of all, OWL provides classes to establish a new ontology and define its metadata with the `owl:Ontology` class. User-defined classes are instances of

[8] for every statement about resources, it is possible to decide if it holds or not

[9] `http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OwlVarieties`

[10] `http://www.w3.org/TR/2004/REC-owl-features-20040210/#s2`

`owl:Class`, a subclass of `rdfs:Class`. OWL introduces `owl:Thing`, a parent of every userland individual item, and `owl:Nothing`, an empty class.

Since RDFS vocabulary is a part of the OWL specification, we can use all the features mentioned in Table 1.2.

Properties

Different types of properties affect the behavior of predicates in RDF triples, several of them are used in the bill ontology.

- **owl:ObjectProperty** – object properties are relationships between two class instances.
- **owl:DatatypeProperty** – values of datatype properties are RDF literals XML schema datatypes, e.g. strings, integers, or dates.
- **owl:FunctionalProperty** – indicates that the value of the predicate is unique for each subject, e.g. a car can only have one date of manufacture.

Each property can have a defined range and domain narrowing down what subjects can use it and what objects can be the value of the property. Relations can be established on properties with the following property characteristics. These allow *reasoning* on the data model, e.g. deducing new statements based upon existing ones.

- **owl:TransitiveProperty** – if P is a transitive property, x, y and z are resources, and both $P(x, y)$ and $P(y, z)$ are true, then also $P(x, z)$ can be deduced.
- **owl:SymmetricProperty** – let P be a symmetric property and x and y resources. $P(x, y)$ holds true if and only if $P(y, x)$
- **owl:inverseOf** – if a property P is marked as an inverse of property $P2$, then we can deduce $P(x, y)$ from $P2(y, x)$ for resources x and y.

1.3.2 Dublin Core

When representing resources on the web, be it images, documents, videos, or whole pages, it can be observed that most of them have several common properties such as a title, a creator, a date of publishing or subject of the resource.

Additionally when representing an object in RDF, it is useful to tell what format it is stored in, under what license terms it is provided or what language it is written or spoken in. We can refer to this information as the resource's *metadata*.

The Dublin Core Metadata Initiative provides a set of vocabulary terms which describe these common properties. In order to avoid duplication of semantically identical relationships in Linked Data, Dublin Core terms can be used as a shared and interoperable vocabulary. The *Dublin Core Metadata Element Set* contains fifteen base metadata elements.[11][12]

The base metadata elements are available under two common namespaces: the older `http://purl.org/dc/elements/1.1/` and the newer `http://purl.org/dc/terms/`. The first one was created before RDF and lacks definitions of domains and ranges. Later, the second namespace was introduced in order to avoid backward incompatible changes, containing all fifteen original elements and multiple new ones, with a better definition.[13] The bill ontology uses the newer definitions.

Advantages of this approach are obvious, an RDF object is identified primarily by its URI but in the real-world we would like to display a human-friendly name. If the dataset uses Dublin Core, we can look for the triplet which contains the predicate `dcterms:title` and find the appropriate name.

1.3.3 FRBR

The Functional Requirements for Bibliographic Records model was created with the intention to classify bibliographic records but covers a much wider range of intellectual and artistic works. It covers three group of entities, intellectual works themselves, their authors and custodians and subjects of the the intellectual endeavor of the first two.[2] The FRBR RDF vocabulary enables to use this model as an ontology for RDF data. FRBR will be used to organize bills and their individual revisions.

The first group entities defined by FRBR contains four classes:

- Work
- Expression
- Manifestation
- Item

[11] `http://dublincore.org/documents/dces/`
[12] `http://tools.ietf.org/html/rfc5013`
[13] `http://wiki.dublincore.org/index.php/FAQ/DC_and_DCTERMS_Namespaces`

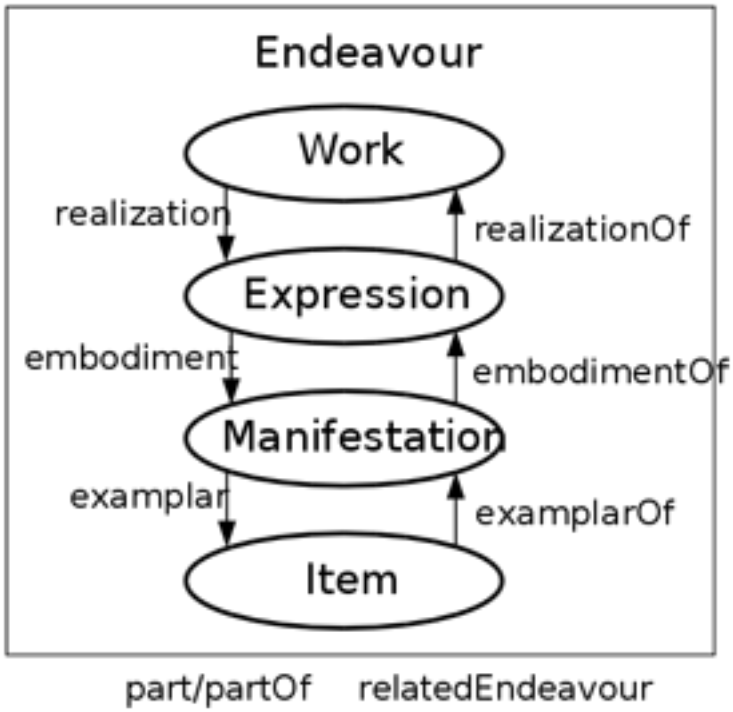


Figure 1.3: Basic FRBR classes and their relations

The `Work` class represents the the artistic work as a whole, be it a book title, a musical piece, or a painting. The `Work` is not connected to a materialized form. Instances of `Expression` are individual realizations of the `Work`. When talking about a book, a first edition and a second revised edition of the same title would be two different Expressions of one Work, the same applies for the original and a translation. Beethoven's Fifth is a Work, its arrangement for an orchestra is an `Expression` but the arrangement played by two different orchestras would create two separate `Manifestations` of it. `Items` are physical copies of expressions, exemplars in your local library or CD recordings of a concert[14].

This is general and flexible enough to describe revisions of a document in time and include the participants present at each step. It isn't necessary to use all of the provided classes. In the case of bills, each bill is considered a `Work`, its individual revisions are `Expressions`, Parliamentary Press containing the revision would be a `Manifestation` and in the printed form an `Item`.

[14] `http://archive.ifla.org/VII/s13/frbr/frbr1.htm#3.2`

1.3.4 FOAF

The Friend of a Friend ontology is used to describe people, information about them, and relationships between them.[17] FOAF defines classes such as `foaf:Agent` and its direct subclasses, `foaf:Person` or `foaf:Group`. Alongside classes representing agents it defines useful properties describing information about them, e.g. `foaf:firstName` and `foaf:surname`, and describing relationships between them, e.g. `foaf:knows` or `foaf:member`.

FOAF was created to present information about authors on the Web and properties in its vocabulary often reflect this goal, but thanks to its attempt to generalize the properties it defines and despite the term *friend* in its title, it can be used in other contexts, e.g. the Parliament or political parties can be considered as groups and their respective members can be classed as people. It can also be used as a base for a more complex organizational structure vocabulary, since frameworks intended for use with RDF might provide specialized handling for the better known ontology, in this case, FOAF.

1.3.5 SKOS

SKOS, short for Simple Knowledge Organization Scheme, is a W3C recommendation for organizing classification systems like taxonomies or thesauri, SKOS refers to them in general as concept schemes. It enables the creator to establish semantic relationships between defined terms, these relationships are not limited to synonyms, broader and narrower terms, associations, or translations.

SKOS will be used to define known states of bills and to enable properties such as who is the bill's sponsor to be typed.

SKOS Core is the component defining the RDF vocabulary used for organizing concept schemes. The fundamental unit is called a *concept*.[18] This can be an idea, object, class or a whole category of them. For example *Emotions* could be a SKOS concept and *PositiveEmotions* and *NegativeEmotions* could be narrowed terms, Example 1.7 shows this example in SKOS written in TTL.

Using SKOS proved to be a good idea for enumerable values. Sometime during the end of 2013, the exported data started to distinguish which region proposed the bill as a sponsor. Data converted before the change used the `http://linked.opendata.cz/resource/legislation/bill-sponsors#RegionalAssembly` property as the value for the bill sponsor. When entries for each region were added, the vocabulary was extended with properties for each region and a `skos:narrower` relationship was added between them.

```
@prefix ex: <http://www.example.com/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

ex:emotion rdf:type skos:Concept;
  skos:prefLabel "Emotion"@en.
ex:positiveEmotion rdf:type skos:Concept;
  skos:prefLabel "Positive Emotion"@en;
  skos:broader ex:emotion.
ex:negativeEmotion rdf:type skos:Concept;
  skos:prefLabel "Negative Emotion"@en;
  skos:broader ex:emotion.
ex:Happy rdf:type skos:Concept;
  skos:prefLabel "Happy"@en;
  skos:definition "feeling pleasure and enjoyment because of
     your life, situation, etc."@en ;
  skos:broader ex:positiveEmotion .
```

Example 1.7: SKOS concept scheme for emotions

1.3.6 Eurovoc

A notable application of SKOS is Eurovoc, a multi-lingual thesaurus managed by the European Union. It covers topics which are subject to European legislation, legislation from the European Parliament in particular.

Equality of every official and working language of the European Union is established in its very first piece of legislation.[15] Finding appropriate translations and usages of terms in other languages than the one in which the document is drafted can be a difficult task. Sometimes direct translations do not produce the correct term and at times, multiple equivalent translation can exist, in which case it is important to stay consistent and use the same one in multiple places. This is one of the goals of Eurovoc is to solve the mentioned problems by using a SKOS concept scheme providing information about the covered subjects in different languages and in the correct context.

Another usage of Eurovoc, directly related to Linked Data, is categorizing resources by subject. A bill can use relation properties from SKOS to tag which SKOS concept it is related to. A simple query to the Linked Data cloud can then return all documents covering a subject, e.g. if a law is related to income tax, the following RDF triple in Example 1.8 can be stated.

[15] http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CONSLEG:1958R0001:20070101:EN:PDF

```
<ld:resource/legislation/cz/bill/2000/1234>
<skos:relatedLabel>
<http://eurovoc.europa.eu/1326>
```

Example 1.8: Adding a Eurovoc thesaurus term to a RDF resource

1.4 Apache Jena

Jena is a framework written in Java providing an API for working with RDF data. Features range from basic model manipulation such as creating and modifying individual statements (RDF triples) to reasoning based on an OWL ontology.[22] Originally developed by HP Labs, it is now a top-level project of the Apache Software Foundation and is under active development.[16]

Core API

The core API is designed to create and read RDF graphs. Operations on created graphs such as unions or intersections are supported and enable the programmer to combine several datasources. Common vocabularies, e.g. FOAF, RDFs, VCARD, are included in the source code.

```
Person p;
com.hp.hpl.jena.rdf.model.Model model;
...
Resource person = model.createResource(p.getRdfUri());
person.addProperty(RDF.type, FOAF.Person);
person.addProperty(RDF.type, VCARD.NAME);
person.addProperty(DC.title, p.getFirstName() + "␣" +
   entity.getLastName());
person.addProperty(FOAF.firstName, entity.getFirstName());
person.addProperty(FOAF.family_name, entity.getSurname());
person.addProperty(FOAF.member,
   model.createResource(parliament.getRdfUri()));
```

Example 1.9: Creating an RDF resource for a Deputy

In the core API, Jena supports simple querying by subject, predicate or object as shown in Example 1.10. For more advanced data querying, SPARQL support is implemented in the Jena ARQ component.

TDB Triple Store

The core API deals with managing the data model in memory. In order to achieve persistence, Jena provides a triple store called TDB. It allows to serialize graphs

[16] `http://jena.apache.org/about_jena/about.html`

```
import com.hp.hpl.jena.rdf.model.*;
Model model;
StmtIterator it;
it = currentModel.listStatements(null, RDF.type, FOAF.Person);
```

Example 1.10: Listing resources of RDF type Person from the FOAF vocabulary

into a custom directory and read them on the next execution of the program. Datasets created by TDB are binary compatible across 32-bit and 64-bit systems.[17]

In addition to storage, TDB provides SPARQL support. SPARQL is query language created for RDF data. It understands the concept of triples and allows querying over the graph supporting filtering, aggregation, negation, set operations or expressions. Other SPARQL specifications support updating of the the dataset, however knowing about features for reading graphs is sufficient for the understanding of data manipulation done in this thesis.[9]

The two basic queries are *SELECT* and *CONSTRUCT*. The first returns a table-like result set, the latter builds a RDF graph. Example 1.11 shows a SELECT query which returns a table with bill URIs in the first column and their names in the second.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?b ?t
WHERE {
 ?b dc:title ?t .
 ?b rdf:type
    <http://linked.opendata.cz/ontology/legislation/bill#Bill>
}
```

Example 1.11: SPARQL SELECT query for bills with their names

Ontology API

In Chapters 1.2.2 and 1.3.1 we introduced two common languages for structuring RDF data, RDF Schema and OWL. Jena's Ontology API tries provides a language agnostic interface for working with languages like RDFS and OWL. The main two components of the API are intended for creating the ontology and making deductions, a process called *reasoning*, based on known properties

Manual writing by hand was chosen for the bill ontology instead of using a programming API. There was no need for multiple language support because

[17] http://jena.apache.org/documentation/tdb/architecture.html

OWL was chosen as the primary language. The ontology, when written in a readable manner, can also serve as documentation for advanced users of the ontology. Automatically generated code usually lacks organization and is harder to read by humans.

2. Source data for the legislative process

In this chapter, the scope of the processed data will be defined. Bills are the primary subject of this work, but in order to create usable applications it is necessary to collect related information. Examples of such information are votes and the representatives who cast them, political parties and their members or organs which make the appropriate decisions for passing a bill.

The primary source of collected data are public database exports of the Czech Chamber of Deputies. The format of collected data will be described and the method of collecting, parsing and updating will be explained.

2.1 Scope of source data

2.1.1 Bills

Before laws are passed an declared, they are called bills and go through the legislative process. This has several stages and a set of rules which define how the bill passes between them.

First, the bill is proposed by one of the following parties: a representative in the Chamber of Deputies, a group of representatives, the government, a regional assembly, or by the Senate. The text of the bill is distributed among all representatives and a first reading is held. The bill sponsors introduces the proposal and after discourse it is either dismissed or advanced to committees for consideration. Two more readings follow, during which amendments can be proposed and accepted by the deputies. At the end of the third reading a final vote is held and in case of success the bill is sent to the Senate. The part of the process taking place in the Senate is outside the scope of this ontology model. After the Senate, the president signs the bill and it is enacted into law. Both the Senate and the president can veto the bill which a majority of all deputies (101) can override.[19]

Figure 2.1: Legislative process in the Czech Republic, source: `http://www.psp.cz/sqw/hp.sqw?k=331`

Information about the bill is passed between stages in the form of Parliamentary Press. Each bill has a unique identifier in the form of *serial number / revision*. This number is unique during one parliamentary term. The revision numbering starts at 0 and is incremented when the bill reaches any new stage.

It is intended to represent all stages of passing a bill, e.g. to analyze how long it stays in certain stages, which stages have the most intermediate changes, or in case a bill is passed to link it with the RDF resource of the law which it becomes.

2.1.2 Amendments

During the first three readings in the Chamber of Deputies changes to bills can be proposed in the form of amendments. Since many crucial changes are made during at this point, it is necessary to include them in the RDF model as well. Amendments can be used to solve flaws of the first proposal which the original proponent has planned for, but they can also be used to modify the bill in favor of a private group or company. The act of convincing a legislator to put together such an amendment is called lobbying and can be illegal if something is provided in exchange.

Proponents of the amendments should be stored, a real-world use case could be listing bills with amendments for each deputy, allowing to identify potential lobbying targets. The actual content of amendments is out of the scope of this thesis, as their format varies strongly and machine comprehension of the changes is a complex task by itself.

2.1.3 Members of Parliament and Political parties

One of the most common bill sponsors are representatives in the Chamber of Deputies or groups of them. Deputies vote in each stage of the legislative process. A RDF resource will be created for each representative and their party and parliamentary membership will be included. FOAF and vCard will be extensively used to provide contact information and mention academic degrees and other achievements.

A single representative often keeps his post after an election and during their lifetime, he or she could be part in several parliaments, a separate resource will be created for each parliament to easily distinguish them. The Parliament exists independently on the Government and Prime Minister, which can change multiple times between elections. In the scope of the collected data, only the terms between elections will be taken into account, individual governments which existed during

them will not be regarded.

2.1.4 Votes

Votes are held in multiple stages of passing a bill and in several different organs. Plenary votes of the whole chamber are held during readings and when overriding Senate and Presidential vetoes. Technical details of voting may vary on different occasions, but the concept can be easily generalized. In the bill ontology, a OWL class intended for representing votes is created and covers the possible outcomes of a vote and includes the individual ballots.

Naturally, every participant can vote in favor or against a proposal. A third common option is to abstain from voting. If a participant is not present at a vote, the absence is marked and the procedure is same as when abstaining. Depending on what type of bill is being proposed, a quorum[1] of votes is necessary. For ordinary laws, the quorum is half of the present deputies. For constitutional laws, three-fifths of the whole chamber must vote for the proposal in order for it to pass.

2.2 Chamber of Deputies exports

The primary source of data for the ontology and model are machine readable exports from the public website of the Czech Chamber of Deputies (`http://www.psp.cz`). The Chamber of Deputies started to publish information in machine readable formats at the end of 2012 with archive files containing bill data and deputies. Over the next year, relations to the collection of laws and proposals of changes in laws were added.

At the time of the writing of this thesis, Eurovoc vocabulary descriptors for individual bills are not included in exported data and in order to include them in the dataset, scraping of the Chamber of Deputies website would have to be used.[2]

Data provided by the Chamber of Deputies is provided free of charge and the only licensing terms are providing a reference to the source of the data and possibly the date of the processing[3]. To comply with the terms, a link to the

[1] the total number of votes necessary to make a decision

[2] I have tried to contact technical administrators of the psp.cz website asking if descriptor IDs could be publicized. The response was negative citing licensing issue. Unfortunately even with explanation and persuasion that referring to a URI does not violate any license terms the descriptors were not added.

[3] `http://www.psp.cz/sqw/hp.sqw?k=1300`

source is included in the header section of the packaged serialized RDF data.

2.2.1 Format of export files

Provided data probably come from a relational database management system. Datasets are normalized tabular data. This means data is organized into multiple tables to reduce redundancy and dependency. Since the data is almost prepared to be imported into a relational database management system, for more complex datasets such as bill stages, some of the data was imported into an temporary database and extracted with SQL into a format more suitable for creating RDF resources. Extraction with SQL is described in Chapter 4.2.3.

Each dataset is a plaintext file encoded in the Windows-1250 character set. Columns are separated with a vertical bar and backslashes are used as escape sequences. An example of one of the files is provided in Figure 2.2.

```
12||Česká republika|Czech republic||1|
18|12|Prezident| ||39|
6|12|Politická strana|Party||14|
11|12|Parlament|Parliament||10|
1|11|Klub|Political Group|1|400|
3|11|Výbor|Committee|3|100|
```

Figure 2.2: The first six lines of the dataset contained in the typ_organu table

Several issues were encountered when parsing the data. Fields are only sometimes trimmed of whitespace. Internally, all string data is converted from Windows-1250 to UTF-8. Some export files contain a wide variety of data. An example is the archive with entries for individual bills which contains bills old as the early 1920s. Only data from the year 1993 and onward is processed. Extending the resource model to older bills could be a subsequent project to this thesis.

Taking the defined scope into account, the expected size of the dataset is several million triples. The largest contributor to this number are individual ballots cast for votes since for every vote, there are circa 200 ballots. Excluding ballots, the dataset contains small hundreds of thousands of statements.

3. Ontology definition

Once the scope of the the source data has been defined, it needs to be converted with the help of RDF and existing and new ontologies into Linked Data. This chapter will describe the ontology whose domain are bills passed by the Czech Parliament. Preferably, the resulting ontology should be general enough to cover similar processes in other countries using an analogical legal system.

Several challenges have to be taken into account such as the changing and evolving nature of bills. A common requirement will be to query documents which are in the process of being passed (or rejected) and new events happen regularly. It is desirable to link resources to laws which are already represented as RDF resources and which they become after the legislative process comes to a successful end. And last but not least directly linking bill resources and legislative process phase resources to the votes which passed or rejected them.

As mentioned before, OWL has been chosen as the platform for writing this legislative ontology. Classes and properties used from OWL were explained in Chapter 1.3.1. The ontology consists of class definitions for new entities, property definitions, concept schemes for state values and semantic restrictions which check exported data follows the legislative process. It is written by hand and not generated by a program.

3.1 Resource URIs

One of the local Czech advocates of Linked Data is the OpenData.cz initiative project which aims to provide datasets about public administration in the Czech Republic and collect applications based upon them.[1] More than a dozen of datasets have already been published, including those about laws.[2] Since it is planned to publish the created dataset alongside them, it is desirable to use a common format of URIs, which are a fundamental property of each RDF resource.

Developers of ontologies and datasets from OpenData.cz have created a set of basic principles to design URIs[5]. To start with the beginning of the URI, two domains are used. `http://linked.opendata.cz` is the common prefix for all finished and clean datasets. The `http://ld.opendata.cz` prefix is used for work in progress datasets and developing ontologies.

Resources are then categorized into three types with different URI prefixes:

[1] `http://opendata.cz/en`
[2] `http://cz.ckan.net/group/opendata-cz`

- Non-information resources or real-world objects – `http://linked.opendata.cz/resource/`
- Ontology resources and vocabulary resources – `http://linked.opendata.cz/ontology/`
- Information resources (documentation) – `http://linked.opendata.cz/page/`

With this in mind, the following URI prefixes were used for the core ontology and associated SKOS schemas:

```
@prefix lb:
<http://linked.opendata.cz/ontology/legislation/bill#> .
@prefix sponsors:
<http://linked.opendata.cz/ontology/legislation/bill-sponsors#> .
@prefix billstages:
<http://linked.opendata.cz/ontology/legislation/
bill-legislative-stages#> .
@prefix decisions:
<http://linked.opendata.cz/ontology/legislation/vote-decisions#> .
```

Formats of resource URLs are described in the next sections which describe each type.

3.2 Class and property definitions

The bill ontology builds upon several existing vocabularies and ontologies which are described in Chapter 1.3. Whenever possible it tries to reuse existing classes and properties either by using them directly or by creating subclasses and subproperties. A class diagram of used classes is shown in Figure 3.1;

For brevity, the following namespace prefix declarations are assumed in all examples in this chapter.

```
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
```

```
@prefix vcard:   <http://www.w3.org/2006/vcard/ns#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix frbr:    <http://purl.org/vocab/frbr/core#> .
@prefix lb:
  <http://linked.opendata.cz/ontology/legislation/bill#> .
@prefix sponsors:
  <http://linked.opendata.cz/ontology/legislation/bill-sponsors#> .
```

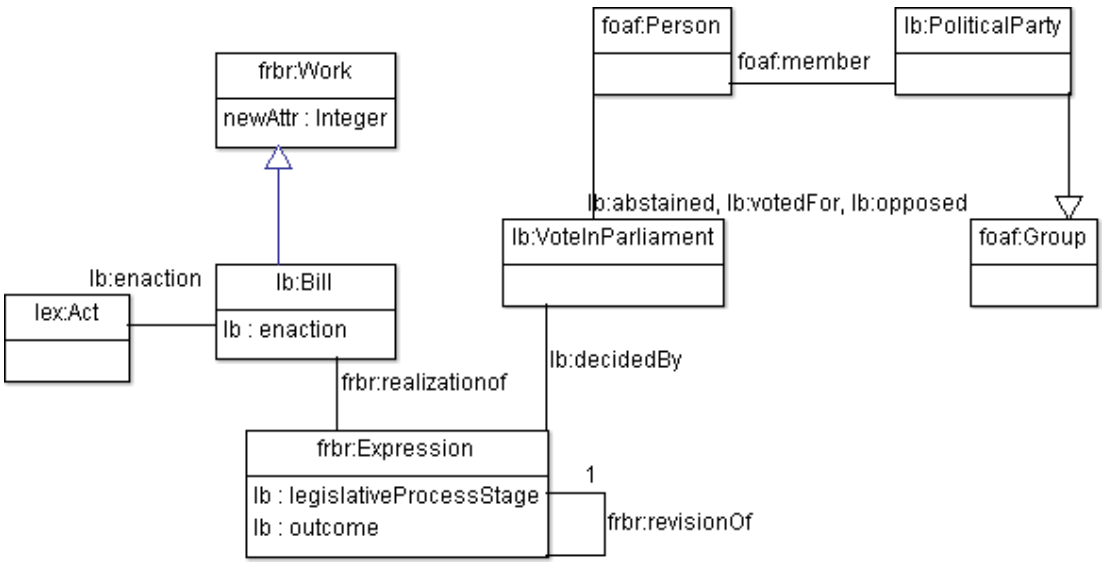


Figure 3.1: Class diagram of RDF classes in the bill ontology

3.2.1 Bills

Bills are the core feature of this ontology model. As previously mentioned in Chapter 1.3.4, the FRBR vocabulary will be used to represent them. This way, the semantic relationship between individual stages of the legislative process and between them and the legislative process can be established in RDF. Another advantage is that frameworks which are capable of working with FRBR data can work with this legislative dataset as well. The reference implementation used was the Lex Ontology[6] described in the next section.

The topmost entity is the legislative proposal which corresponds to the `frbr:Work` class. According to FRBR, a Work is a distinct intellectual creation.[3] In order to add custom properties, `frbr:Work` is sub-classed and the proposals are instances of the `Bill` class.

As the proposal goes through the legislative process, we gain more information about it in each step. The primary goal and intention of the proposal still

[3] `http://archive.ifla.org/VII/s13/frbr/frbr1.htm#3.2`

stays the same though. With this reasoning, the individual steps can be identified as *realizations* of the proposal. In FRBR vocabulary this means individual revisions are instances of `frbr:Expression`. To capture the sequence of revisions in the legislative process the FRBR's `revisionOf` property which points to the previous phase is used. To mark that the expression is a realization of the bill, the `realizationOf` property is used. Since `revisionOf` and `realizationOf` are defined as inverse properties, mentioning them in one of the resources involved suffices.

Whenever properties from other vocabularies with greater support and with an unambiguous meaning are available, e.g. titles and dates from Dublin Core, they are preferred over custom properties defined in the ontologies vocabulary. In case a property from another vocabulary is used, its label, domain and range are specialized for the specific context. Example 3.1 shows an example of reusing the `dcterms:date` property for representing the introduction date of bills. Table 3.2 shows the properties defined on the `Bill` class and properties from other vocabularies which are used.

```
dcterms:date a owl:FunctionalProperty ;
  rdfs:label "Datum předložení návrhu zákona"@cs, "Date of
     proposal of the bill"@en ;
  rdfs:domain lb:Bill ;
  rdfs:range xsd:date ;
  rdfs:isDefinedBy lb:
  .
```

Example 3.1: Reusing a property from a different vocabulary in a specialized context

`dcterms:identifier`	year and serial number of the bill
`lb:billSponsor`	indicates which group or entity proposed the bill to the Parliament
`dcterms:title`	short variant of the proposal's title
`dcterms:description`	full variant of the proposal's title
`dcterms:date`	the date when the bill was proposed
`lb:enaction`	the value of this property is the act as which the bill was enacted into law
`lb:eurovocDescriptor`	values are Eurovoc thesaurus entries which are related to this proposal

Example 3.2: Properties used on instances of Bill

Lex Ontology

The fifth star in Tim Berners-Lee five-star rating system mentioned in Chapter 1.1.2 was for linking your data to other people's data. Bills are predecessor of acts for which an ontology already exists. The Lex Ontology[4] developed by the XRG research group as part of the OpenData.cz initiative contains definitions for legal documents including laws. Example 3.3 shows a RDF resource representing a bill (2012/752 to be specific), the law enacted because of this proposal is already represented by a RDF resource from the Lex Ontology as `http://linked.opendata.cz/resource/legislation/cz/act/2013/47-2013`[6]. Therefore, we provide is as the value of the `lb:enaction` property.

```
<http://linked.opendata.cz/resource/legislation/cz/bill/2012/752>
        a lb:Bill ;
        dcterms:identifier "2012/752" ;
        lb:billSponsor sponsors:RepresentativeGroup ;
        lb:introductionDate "2012-07-13" ;
        dcterms:title "Novela z. o specifických zdravotních
           službách" ;
        dcterms:description "Návrh poslanců Jana Chvojky,
           Jaroslava Krákory, Pavla Antonína a dalších na vydání
           zákona, kterým se mění zákon č. 373/2011 Sb., o
           specifických zdravotních službách, ve znění
           pozdějších předpisů "@cs ;
        rdfs:seeAlso
           "http://www.psp.cz/sqw/historie.sqw?o=6&t=752"
        lb:enaction
          <http://linked.opendata.cz/resource/legislation/
          cz/act/2013/47-2013>
```

Example 3.3: Example of a bill resource

3.2.2 Enumerated types as property values

Some of the values of properties of `lb:Bill` and `frbr:Expression` come from a finite and relatively small domain. To be specific, they are:

- `lb:billSponsor`, a property of `lb:Bill` containing the person who introduced it
- `lb:legislativeProcessStage` used in `frbr:Expression` instances
- `lb:outcome` also used in `frbr:Expression` to store the final decision made in a specific stage

[4] `https://code.google.com/p/lex-ontology/`

To enumerate all possible values SKOS concept schemes are created for each one. The range of the listed properties defined with `rdfs:range` are then instances of concepts from the created schemes.

Defining the range in OWL is not straightforward though. The value of a `rdfs:range` property must be a class. OWL provides six ways to describe a class: using a URI reference, with *property restrictions*, by enumeration and by set operations (intersection, union and complement) on other descriptions. [5] Since each value is an instance of `skos:Concept`, the first method does not provide the necessary specificity because other unrelated concepts could be provided as values. The first attempted approach was to use property restrictions as shown in Example 3.4. This states that the value of `lb:billSponsor` must be a `skos:Concept` and at the same time that concept must have the `skos:inScheme` property with `sponsors:` as the value. Specifying multiple classes as the value of `rdfs:range` effectively means their intersection.

```
sponsors: a skos:ConceptScheme ;
  rdfs:label "Concept scheme for possible types of bill
     sponsors"@en .
  .

sponsors:Representative a skos:Concept ;
  skos:inScheme sponsors: ;
  skos:prefLabel "Member of Parliament"@en ;
  skos:topConceptOf sponsors:
  .

 lb:billSponsor a owl:FunctionalProperty ;
  rdfs:range  skos:Concept, [
    a owl:Restriction ;
    owl:onProperty skos:inScheme ;
    owl:hasValue sponsors:
  ]
  .
```

Example 3.4: Use OWL Property Restrictions to define a property range

This approach is verbose and introduces anonymous classes which are better to avoid as one cannot refer to them after they are defined. The second approach which solved these issues involves subclassing `skos:Concept` and typing all concepts as instances of this new class as shown in Example 3.5.

Enumeration with `owl:oneOf` could be used, but it is unnecessarily verbose and changes in the concept scheme would propagate further than necessary into property definitions. SKOS is used not only for enumerating values, but also to organize them. For example, in the concept scheme containing stages of the

[5] `http://www.w3.org/TR/owl-ref/#ClassDescription`

```
sponsors:Sponsor a owl:Class ;
  rdfs:subClassOf skos:Concept .

sponsors: a skos:ConceptScheme ;
  rdfs:label "Concept scheme for possible types of bill
     sponsors"@en
  .

sponsors:Representative a sponsors:Sponsor, skos:Concept ;
  skos:inScheme sponsors: ;
  skos:prefLabel "Member of Parliament"@en ;
  skos:topConceptOf sponsors:
  .

 lb:billSponsor a owl:FunctionalProperty ;
  rdfs:range sponsors:Sponsor
  .
```

Example 3.5: Creating a subclass to define a property range

legislative process, there are concepts for an override of a veto from the Senate and for an override of a presidential veto. Both are subconcepts of a broader term indicating overriding any veto in general. This could be useful to simplify queries looking for types of bills being often returned to the Chamber of Deputies.

3.2.3 Parliament members, parties and parliament

FOAF and vCard are the most popular vocabularies used for describing people and groups[6] and both are used in the bill ontology. Every deputy is an instance of `foaf:Person`. Deputy resources are also `vcard:Name` instances which provides support for formal honorifics and a property for the date of birth. Both are shown in Example 3.6

```
<http://linked.opendata.cz/resource/psp.cz/person/5917> a
   foaf:Person, vcard:name ;
  dcterms:title "Leoš Heger" ;
  foaf:name "Leoš Heger" ;
  foaf:lastName "Heger" ;
  foaf:homepage "http://www.leosheger.cz/" ;
  vcard:honorific-prefix "doc. MUDr." ;
  vcard:honorific-suffix "CSc." ;
  foaf:mbox <mailto:hegerl@psp.cz> ;
  foaf:member
     <http://linked.opendata.cz/resource/psp.cz/group/TOP09>
  .
```

Example 3.6: A complete resource representing a Czech deputy

[6] `http://prefix.cc/popular/all`

`foaf:Group` is used for two types of entities, political parties and for each parliamentary session. The reason behind the decision to create resources for each session instead of the parliament as an organ is that laws being passed are strongly related to the current majority and real-world applications might want to take this into account. However, every resource describing a parliamentary session is linked to `http://linked.opendata.cz/resource/cz/authority/parliament` which represents the parliament as an organ of authority which is defined in the *Metadata about law documents published by the Parliament of the Czech Republic*[7] from the OpenData.cz data catalog. The first parliamentary session in the Czech Republic in RDF is shown in Example 3.7.

```
<http://linked.opendata.cz/resource/psp.cz/group/165>
  a foaf:Group ;
  foaf:name "Poslanecká sněmovna 06.06.1992 - 06.06.1996" ;
  owl:sameAs
    <http://linked.opendata.cz/resource/cz/authority/parliament>
      .
```

Example 3.7: The 1992 - 1996 parliamentary session in RDF

3.2.4 Votes

All votes, meaning the event when voting was held, are instances of `lb:VoteInParliament`. No other suitable vocabulary was found to represent them, so all properties and the class are defined in the ontology. The following properties are introduced and their ranges are instances of `foaf:Person` which, not coincidentally, we use to represent deputies.

- `lb:hasSupporter` and its inverse `lb:votedFor`
- `lb:hasOpponent` and its inverse `lb:votedAgainst`
- `lb:hasAbstainee` and its inverse `lb:abstained`
- `lb:hasAbsentee` and its inverse `lb:absentAt`

Inverse properties are defined using the `owl:inverseOf` property. Both are present in the ontology, because it seemed more natural to define the properties on a class that is introduced in the ontology (a vote has a supporter), but when directly looking at the dataset it is more convenient to list the individual voters as values of the inverse properties defined in statements on the vote resource.

[7] `http://cz.ckan.net/dataset/metadata-k-z-kon-m-z-psp-cz`

4. Linked Data converter application

In this chapter, the methods and infrastructure used for converting raw data exports of bills and related information into RDF will be described. An application written in Java with capabilities of creating and updating the model was created for this task. Its architecture and key transformation methods will be shown. At the end of the chapter, it will be shown how the generated model is published in the Linked Data cloud.

4.1 RDF converter

The converter is a standalone Java application which in one pass collects all the necessary data from remote sources and dumps them in RDF. The application can be separated into two main components. The first handles fetching exported data and an API for reading them, the second uses this API to convert data into RDF. All of the code is in the `cz.vojtechvondra.ldbill` package and its subpackages.

Requirements and dependencies

The required language version is Java 7 SE (Standard Edition). The main reason for choosing Java as the programming language of the converter is the existence of the Apache Jena library. Similar implementations exist in different languages[1], but Apache Jena enjoys the largest support. Choosing Java also means the application has multi-platform support.[22]

Dependencies, packaging, and building are managed by Maven. The application has the following dependencies with minimum versions:

- Apache Jena 2.11.0 – provides a RDF API and triplestore
- zip4j 1.3 – a library for handling ZIP compressed archives
- h2 1.3 – an embedded relational database system
- MySQL JDBC connector 5.1.29
- log4j 1.2 – a library for runtime logging

[1] e.g. RAP RDF API for PHP or RDFLib for Python

- jUnit 4 – unit testing framework

The application can be built with Maven by running `mvn clean install` in the project root. Maven should download all dependencies, generate necessary sources and package the application.

A JAR file containing the converter is produced and copied to the *target/* directory with other JAR dependencies.

4.2 Datasource API

Classes from the datasource API are located in the `cz.vojtechvondra.ldbill.psp` package. They are meant for downloading and accessing raw data from the Chamber of Deputies archives.

4.2.1 PSPDownloader

On the website of the Czech Chamber of Deputies several ZIP files are provided, each containing a number of datasets. The `PSPDownloader` service provides a two public methods, `getDataset` which returns a reference to a local file containing the specified dataset and `getKnownDatasetNames` which returns a list of possible parameters for `getDataset`. Internally, it contains a hashmap which maps datasets to the containing archives.

ZIP archives are requested through HTTP and extracted with the zip4j library. To save bandwidth, ZIP files are cached in the temporary directory of the user under which the program is executed. This way, if multiple datasets from the same archive are requested, the archive is downloaded only once. Execution time is also faster when running the program repeatedly. Example 4.1 shows how to download a dataset file.

```
PSPDownloader downloader = new PSPDownloader();
try {
    File exportFile = downloader.getDataset("osoby");
    // Prints "/tmp/ldbill/osoby.unl"
    System.out.println(exportFile.getAbsoluteFile());
} catch (IOException e) { }
```

Example 4.1: Downloading a dataset

4.2.2 PSPExport

The `PSPExport` service is a wrapper on top of `PSPDownloader`. It is able to parse the individual lines from the export file and sanitize them. It handles loading the downloaded file with the correct encoding. Example 4.2 shows how to load an export file line by line.

```
PSPExport export = new PSPExport(dataDownloader, "osoby");
while ((data = export.getLine()) != null) {
    Deputy d;
    try {
        d = Deputy.createFromLine(data);
    } catch (IllegalArgumentException ignored) {
        /* Not a valid entity in the export file, log and
           continue */
        continue;
    }
    addEntityToModel(currentModel, entity);
}

try {
    export.close();
} catch (IOException e) {
    /* Log error */
}
```

Example 4.2: Importing deputies using the PSPExport service

4.2.3 ExportDatabaseLoader

As mentioned in Chapter 2.2.1 about the format of the export files, the provided archives are raw dumps from a relational database management system. In simple cases like deputy members and political parties the data can be extracted directly as no related data is stored in other tables.

In the case of bills and the stages of the legislative process, it is much simpler and safer in terms of data consistency to re-import the data into a relational database, query it with SQL and convert the result set into RDF.

For votes and ballots, the reason to re-import the plaintext dumps into a relational database is performance. Ballots reference votes by ID and a non-indexed searches for corresponding resources for all ballots would take several hours.

Choice of RDBMS

In order not to introduce dependencies on other running services, H2, a small embedded solution was chosen. It has a very small footprint (.jar dependency is about 1MB). It can store a database in-memory which is ideal for this use-case because no data is persisted, only the querying features of the database engine are needed.

Unfortunately, this choice underestimated the size of the dataset and the memory necessary for a single run of the converter reached several GB and the JVM often crashed on a `OutOfMemoryError` exception because the garbage collecting overhead was too high. This was caused by not having enough space on the heap and the necessity of aggressive garbage collecting.

It was necessary to persist the data on disk to overcome the memory issues. H2 performance during insertion seemed slow, so another RDBMS MySQL was chosen. Setup of the development environment was more complicated as MySQL cannot be embedded into the application but it can handle the ten million records the dataset consists of.

Since both adapters use JDBC, a Java SE data access interface which isolates the storage implementation with an abstraction, the process of switching the storage backend was very simple. Support for H2 was kept in the sources and can be enabled by calling the `ConnectionFactory` appropriately.

`ExportDatabaseLoader` provides two public methods. `importData` imports the provided `PSPExport` into the database. A static helper, `importAll` uses `PSPDownloader::getKnownDatasetNames` to run an import on all known datasets. An example of importing all known datasets is shown in Example 4.3 The importer API then can use the relational data if queries the same `java.sql.Connection`.

```
try (Connection con = ConnectionFactory.create(
        ConnectionFactory.JdbcDrivers.MySQL
    )
) {
    ExportDatabaseLoader.importAll(con, dataDownloader);
}
```

Example 4.3: Importing all known datasets into a temporary database

Importing batches of data

As mentioned in the previous section about choosing the right database system, performance issues when inserting millions of rows were encountered. Optimiza-

tions for importing multiple rows at once were used.

A safe and efficient way to execute queries is to use Prepared Statements which can often be sent to the database server beforehand and compiled. Prepared Statements are parametrized and for each row, where only the values which are inserted differ, the parameters will be those values.

In a loop, the statement will be filled with parameters and added to a batch with `PreparedStatement.addBatch()`. Every n-th loop iteration, the batch will be executed with `PreparedStatement.executeBatch()` and a commit command will be called on the connection so everything is sent to the database server. In the data loader, the batch is executed every 100th iteration.

If explicit transactions are not used, the database server can commit data after each statement which costs time. In order to avoid this, *auto commit* is set to false before the batch insert and restored after it finishes in a *finally* block.

A complete example can be see in Example 4.4.

Generating database tables

Documentation for table schemas, which include column names, data types, and table names, is provided for all the table exports from the Chamber of Deputies[2]. To generate the tables which will contain the imported data a set of classes under the `cz.vojtechvondra.ldbill.psp.tables` package is generated. All are subclasses of the abstract `TableDefinition` which provides convenience methods for formatting SQL queries for creating tables and getters for column names. To generate these classes, a small script written in *PHP*[3] is used and is stored in the `scripts/` directory of the main application repository. It can be run with the following command:

```
cd scripts/
php generateTableSchemas.php
```

This creates a class for each dataset in a temporary directory which can be copied to the package folder in the Java source code. The application then uses Reflection to access the table definition during runtime when importing the dataset.

[2] `http://www.psp.cz/sqw/hp.sqw?k=1300`

[3] PHP is a server-side scripting language (similar to Python or Perl) primarily used for web development but it can be used as a programming language for general purpose scripting, especially its I/O and string handling APIs are very simple to use though not robust as for example Java

```
PreparedStatement stmt;
PSPExport export = ...;
String insertSql = "INSERT␣INTO␣table␣VALUES(?,␣?)";
try {
    conn.setAutoCommit(false);
    stmt = conn.prepareStatement(insertSql);
} catch (SQLException e) {
    logger.error("Could␣not␣create␣prepared␣statement", e);
    return;
}
try {
    while ((data = export.getLine()) != null) {
        stmt.setInt(1, data.key);
        stmt.setString(2, data.value);
        stmt.addBatch();
        if ((batchCount + 1) % 100 == 0) {
            stmt.executeBatch();
            conn.commit();
            stmt.close();
            stmt = conn.prepareStatement(insertSql);
            batchCount = 0;
        }
        batchCount++;
    }
    stmt.executeBatch();
    conn.commit();
} catch (SQLException e) {
    logger.error("SQL␣error␣when␣loading␣data␣into␣table.", e);
} finally {
    try {
        stmt.close();
        conn.setAutoCommit(true);
    } catch (SQLException e) {
        logger.error("Could␣not␣close␣prepared␣statement.", e);
    }
}
```

Example 4.4: Executing prepared statements in batches

Calling of the generating script is included in Maven's configuration by using the `exec` Mojo[4] of the `exec-maven-plugin`. It is configured to run the PHP script during the `generate-sources` phase of the build lifecycle. The target directory for the generated sources is set to the correct location inside the source directory, so no additional copying is needed.

Generated sources should not be commited in version control systems, the contents of the package directory containing generated table definitions is ignored in Git. The `maven-clean-plugin` is also configured to delete the contents of the directory during the `clean` phase of the build.

4.3 Importer API

The importer is a collection of independent classes which incrementally extend the graph model. All import classes implement the `ImportStep` interface which contains a single method `extendModel`. The goal of each implementation is to take a single part of the dataset and extend the RDF model with new data.

Two abstract classes implementing `ImportStep` exist. `FileImportStep` provides basic functionality for import steps which directly insert data from the plain-text files. `JdbcImportStep` is the common ancestor for classes which access the database to insert data into the model.

The design of the individual import steps tries to reflect the nature of Linked Data. Resources can link to incomplete information and knowledge about a resource is gathered gradually. Together, resources reference each other by URIs and a resources' description can always be completed later.

FileImportStep implementation

Each class which extends `FileImportStep` must implement two abstract methods:

- `createEntity` – based on a line from the raw database dump, this methods returns a POJO[5] which validates and sanitizes the string data and converts them into a reasonable Java representation (e.g. converts string dates to `java.util.Date`). The class type of the entity is the generic parameter of `FileImportStep` which is specified in the extending class.

[4] Maven term for implementation of a single plugin goal

[5] Plain Old Java Object – a class which does not inherit from framework or Java object model classes

- `addModelToEntity` – this method accepts the created entity and adds it to the RDF model extending the graph. In the application, this is an instance of Jena's `com.hp.hpl.jena.rdf.model.Model` which is passed around the application.

Each import step is passed an instance of `PSPExport` and the current RDF model in the constructor. It then uses the datasource API to go through the file line-by-line and tries to call `createEntity` on each one. If the implementation does not support the contained data[6], it returns a runtime `IllegalArgument Exception` which is logged and the import continues with the next line.

Ontology import

The ontology itself is an RDF graph an needs to be imported as well. Since the ontology was written by hand in TTL format, it seemed most straightforward to use Jena's API for reading serialized RDF files (specifically `RDFReader`). The `RdfImportStep` class was created for this task and accepts a `File` or `InputStream` in its constructor from which it loads the data. `RDFReader` loads the input stream into an existing model, so conventiently the one being extended is used directly.

Since the ontology is imported every time and is part of the application's version, it is packaged as a resource in the resulting jar file. When a Java application is packaged as a jar, one must use `Class.getResourceAsStream()` instead of `Class.getResource()`. This is because the jar is a ZIP file in which the resource is packaged and a file URL does not make sense. However, Java can provide a Stream leading to the file in the archive. If an attempt to access a URI returned by `getResource` is made from an executed jar, Java throws a bit misleading exception: *URI is not hierarchical*.

Logging

Each import step logs in detail action it performs and reports any unexpected circumstances. Log4j is used as the logging library and with the default configuration, logs are printed to standard output. The most detailed output can be obtained by observing the *Debug* logging level. Possible errors in parsing the data are logged on the *Warning* level. During subsequent runs of the converter during testing, most of them were eliminated by fixing bugs in the conversion process. An example are undocumented legislative process stages recognized by the `BillRevision` class or missing dates in rows containing data for the revisions.

[6] this can happen when the dataset contains multiple entities, e.g. the `organy` dataset contains both parliamentary sessions and political parties

4.4 Building and continuous deployment

The application is built and published using a Jenkins[7] instance, an open-source continuous integration server. It takes care of some common tasks associated with development and allows multiple developers to work synchronously on the project. Jenkins is notified about any changes in git from the Github project (when any change is pushed) and runs a full build with Maven including unit tests. If a build was successful, it publishes the compiled jar and library dependencies as artifacts and plots data about testing.

Unit tests currently cover only part of the application, but code which was created at the end of development should be covered more. During development test-driven development was attempted, but the main problem was complicated mocking of the resource model. Some unit tests were added as regression tests, to verify a bug encountered during usage was fixed.

Building and artifact archivation is the task of the first job: *ld-bill*. A second downstream job exists, called *ld-bill-convert* which uses the converter from the first one and runs the import. If the import succeeds, it promotes the linked data to the public server which serves it (mentioned in the next section). It also archives the serialized data as a build artifact.

The server is public and is located at `http://ci.vojtechvondra.cz`. Anonymous users are given read-only access to build jobs related to the bill ontology and can download artifacts produced for each changeset.

4.5 Serving Linked Data over SPARQL

As mentioned in the previous section, Jenkins promotes the converted data to SPARQL server called Fuseki. It is open-source as well and maintained by the Apache Foundation as part of the Jena project. It is a HTTP server providing support for the SPARQL 1.1 Query protocol[8] and allows anyone to make queries on the dataset.

In the attached materials (as described in Appendix A) a configuration file for Fuseki is provided. It is also written in TTL which is a suitable language for the declarative nature of configuration files.[9] It loads the dataset from the converted TTL file and makes a queryable SPARQL endpoint available. The server can be

[7] `http://jenkins-ci.org/`

[8] it provides support for writing as well, but the instance used for the bill dataset does not enable them

[9] More information about syntax of configuration can be found in the Fuseki documentation: `http://jena.apache.org/documentation/serving_data/#fuseki-configuration-file`

run with the following command:

```
fuseki-server --config=config.ttl
```

4.5.1 Updating the dataset

When approaching the task of finding a way to keep the dataset up-to-date with the latest changes in legislature, the first proposed solution was to find a way to incrementally add changes since the last generation of the dataset. There is no simple way to detect the individual changes however, as from experience during development, amendments to the tables holding the data were made propagating back into history[10]. This approach also seemed generally error-prone and in case of a software bug, a complete re-conversion would have to be made to ensure consistency of data.

During testing, the average duration of a complete conversion process with an initial import into a temporary relational database was around two hours on a server with a single CPU and 2GB of RAM. This seemed as an acceptable task running time for it to be run regularly. As mentioned in Chapter 4.4 about continuous deployment an integration server with a scheduled job running the converter was set up. The *ld-bill-convert*[11] job runs nightly and produces the whole dataset. Using the Jenkins Promoted Builds plugin[12] it automatically, right after the build finishes, copies the resulting dataset into the Fuseki directory and reloads the server.

In case of failure, a number of previous builds is kept to have copies of correctly coverted Linked Data datasets available.

[10] e.g. adding of voting data from 26th June 2013 as noted in the changelog `http://www.psp.cz/sqw/hp.sqw?k=1340`

[11] `http://ci.vojtechvondra.cz/job/ld-bill-convert/`

[12] `https://wiki.jenkins-ci.org/display/JENKINS/Promoted+Builds+Plugin`

5. Visualizations

This chapter will show the practical applications of the generated data model. To demonstrate possible usages of the dataset created in this thesis, several visualizations usable and comprehensible for average users have been created. Some re-use rendering capabilities of existing systems and some use a newly created timeline viewer.

5.1 Payola

All visualizations are created and managed through an application called Payola. A student project from the Charles University, it aims to be an application for displaying and analyzing RDF data. One of the main goals is to make it possible for users without knowledge of SPARQL to extract and analyze data. It then offers ways to share analyses with other users[1].

In Payola, the user can define datasources and run *analyses* on them. Results of existing analyses can be used as datasources as well. Analyses produce a graph or SPARQL query row set which can subsequently be rendered using a number of plugins (e.g. force directed graph, grouped triples or map visualization)[8]. Joining of multiple datasources is supported and a mouse-controlled visual interface for creating analyses is available. One of the goals of this thesis was to enhance Payola by creating a new way to visualize analyses and to promote it by creating a new Open Data dataset.

Payola is written primarily in Scala and Javascript. What more, the client side accessed by a web browser is written in Scala but compiled to client Javascript by a custom compiler called *s2js*. Objects such as HTML DOM elements have their counterparts in Scala and the architecture is smart enough to convert memory objects from Scala to the appropriate Javascript representation including support for pre-defined functions. The system will be explained in more detail in the next section.

5.2 Statistical visualization

The first statistical visualization results in a column graph showing the number of proposed bills by year. The existing column graph plugin in Payola accepts a

[1] A comprehensive list of features can be found in the User Guide: `https://github.com/payola/Payola/blob/master/docs/user_guide.md`

specially structured graph with several nodes that have three outgoing edges: a connection to a common root node, the value and the label. Recognition of which value is which is done only by type detection. The SPARQL SELECT query used to construct the graph is shown in Figure 5.1 and uses many SPARQL language constructs. The core is an aggregation SELECT query which groups bills by the date part of their proposal date and is wrapped by a CONSTRUCT query which creates the graph described.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
        ?uuid rdf:type <node://columnroot>.
        ?uuid rdf:type ?count .
        ?uuid rdf:type ?sy
} WHERE {
        SELECT
                ?y (UUID() as ?uuid)
                (STR(?y) as ?sy)
                (COUNT(?s) as ?count)
        {
                ?s rdf:type <http://linked.opendata.cz
                   /ontology/legislation/bill#Bill> .
                ?s <http://purl.org/dc/terms/> ?d
                FILTER (
                        STRDT(?d, xsd:date) >=
                           "1993-01-01"^^xsd:date
                )
        } GROUP BY (YEAR(STRDT(?d, xsd:date)) AS ?y) ORDER BY ?y
}
```

Example 5.1: SPARQL query for Payola column graph displaying proposed bills by year

Originally, the converter application did not use typed literals to represent the dates so the `STRDT` function which produces a typed literal is used to make sure date functions such as `YEAR` will understand the value. A FILTER clause is used to filter only modern day bills. Since the construct query requires an IRI in the first place of the statement, `UUID` is used to generate a random IRI for each aggregated count.

The resulting graph is shown in Figure 5.1. The values can be interpreted in terms of political events. An example is the lower amount of proposed laws in the years 1998 and 2006. In 1998 a caretaker government with Josef Tošovský as Prime minister was appointed and in 2006 Mirek Topolánek's first government did not pass a vote of confidence after the summer elections. The 1998 number however does not correspond to the number in 2009 where 200 bills were proposed

during the Jan Fischer caretaker government. A possible expansion of the dataset which could give an explanation would be to assign bills with the parliamentary session and/or government during they which they were presented. This could be part of a future public administration or political system ontology.

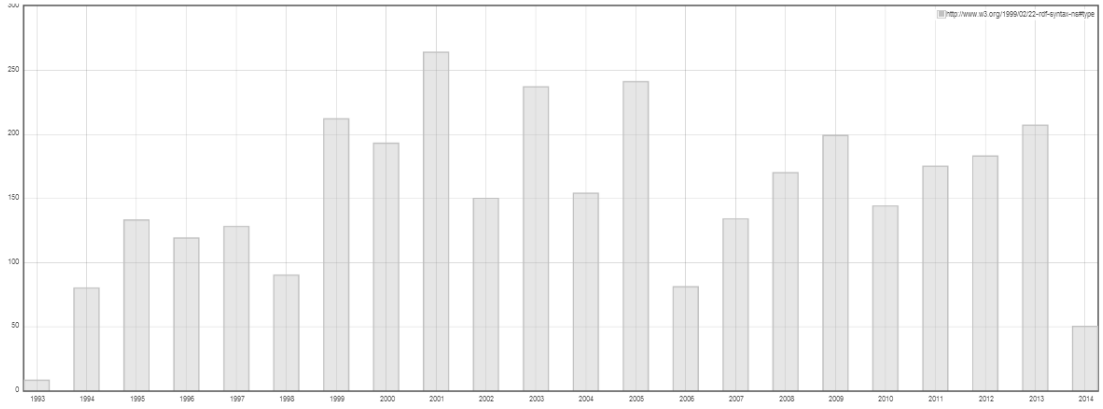


Figure 5.1: Bill proposals by year

5.3 Identifying non-standard legislative procedure

Using SPARQL, it is possible to analyze the data and identify possible occurrences of non-standard legislative procedure. One such case are bills which were passed hastily and the time passed between the second and third reading (during which amendments can be proposed) is shorten than usual[7, p. 14]. Interpretation and result filtering is not the goal of this thesis but this analysis is a shown concrete real-world example and will be shared on the live Payola site as a base for others.

The query in Example 5.2 returns bills proposed in 2013 ordered ascending by the length of the interval between the two readings.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX lb: <http://linked.opendata.cz/ontology/legislation/bill#>
PREFIX frbr:  <http://purl.org/vocab/frbr/core#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX billstages: <http://linked.opendata.cz/ontology/
legislation/bill-legislative-stages#>
SELECT DISTINCT  ?name ?ident ?datesr ?datetr ((?datetr -
   ?datesr) AS ?interval)
WHERE
{
    ?bill a lb:Bill .
    ?bill dcterms:date ?billdate .
    ?bill dcterms:title ?name .
    ?bill dcterms:identifier ?ident .
    ?sr frbr:realizationOf ?bill .
    ?tr frbr:realizationOf ?bill .
    ?sr lb:legislativeProcessStage  billstages:SecondReading .
    ?tr lb:legislativeProcessStage  billstages:ThirdReading .
    ?sr dcterms:date ?datesr .
    ?tr dcterms:date ?datetr .
    FILTER (YEAR(?billdate) = 2013)
} ORDER BY ?interval
```

Example 5.2: SPARQL query returning list of bills from 2013 ordered by interval between second and third reading

The interval field is returned by the SPARQL query engine in xs:duration[2] format. This is due to RDF being closely related to XML and re-use of XML datatypes which are defined using XML Schema (XSD). For example, the `YEAR` function from Example 5.2 is the equivalent of XPath's `year-from-dateTime`. Unfortunately, the SPARQL equivalent of XPath `days-from-duration`[3] does not exist and string manipulation must be used.

Name	**Second**	**Third**	**interval**
2013/1058	2013-08-13	2013-08-16	P3DT0H0M0.000S
2013/1121	2013-08-13	2013-08-16	P3DT0H0M0.000S
2013/13	2014-02-11	2014-02-14	P3DT0H0M0.000S
2013/53	2014-02-11	2014-02-14	P3DT0H0M0.000S
2013/887	2013-05-10	2013-05-15	P5DT0H0M0.000S
2013/888	2013-05-10	2013-05-15	P5DT0H0M0.000S
2013/891	2013-05-10	2013-05-15	P5DT0H0M0.000S
2013/896	2013-05-10	2013-05-15	P5DT0H0M0.000S

Table 5.1: Result of query from Example 5.2 (only selected columns)

[2] `http://www.w3.org/TR/xmlschema-2/#duration`

[3] `http://www.w3.org/TR/xpath-functions/#func-days-from-duration`

5.4 Timeline visualization plugin

The `TimelinePluginView` class encapsulates a new visualization plugin rendering resources which have a date description in chronological order with distance between them proportional to the time elapsed between them. An existing Javascript library was chosen and the task of the plugin is to convert the Graph stored in Payola's memory into a representation comprehensible for the library.

When choosing the library several aspects were taken into account. The first requirement was a HTML5 and Javascript implementation (no Flash or other technologies requiring additional runtime dependencies). The second requirement was for the project to be open-source and usable under a Open Source license[4]. Functional requirements included: the ability to display both points and ranges, to display auxiliary info in a tooltip or in the timeline item's description. As for design, it should be possible to seamlessly integrate it into Payola's interface. Since Payola is not designed responsively, responsiveness of the timeline's markup was only an optional feature.[5]

The first candidate was the Timeline visualization from Google Charts (`https://developers.google.com/chart/interactive/docs/gallery/timeline`). Rendering charts with HTML5 and SVG, it met the technology requirements. It is however primarily intended to draw Gantt charts and single timeline points (with no duration) couldn't be drawn. Additionally, Google Charts is only an API and the Terms of Service do not allow the library to be downloaded and used offline[6]. This disqualified Google Charts from the selection.

The second candidate was the SIMILE Timeline Web Widget (`http://www.simile-widgets.org/timeline/`). It meets the technology requirements as well, is released under the very liberal BSD license and can be used offline. It distinguishes timeline points and ranges and would be acceptable solution. The third candidate however, Timeline JS, matched the features of the previous widget and seemed to have more traction in development and community contributions. Visually appealing, it is simpler to configure than the previous widget while maintaining the same feature set[21].

[4] one of OSI approved licenses: `http://opensource.org/licenses`

[5] the recommended screen width is at least 1440px `https://github.com/siroky/Payola/blob/develop/docs/installation_guide.md#system-requirements`

[6] `https://developers.google.com/chart/interactive/faq#offline`

5.4.1 Implementation of the Timeline plugin

The plugin is a subclass of the `PluginView` class and the second chart type plugin (next to the Column Chart[7]). Similarly to the Chart Plugin, the Timeline Plugin expects the graph to have a certain structure. Effort was made to make the necessary structure prerequisites less rigid than for the Column Chart which for example requires all edges to be instances of `rdf:type` although it defies its semantic meaning.

A graph node is called an *event node* if it has one *label edge* and one *date edge*. Possible property types are listed below. It can also contain any number of other properties different from the listed. The edge URIs are enumerated in the `cz.payola.common.rdf.Edge` class.

- Label edge
  - http://www.w3.org/2000/01/rdf-schema#label
  - http://purl.org/dc/terms/title
  - http://purl.org/dc/elements/1.1/title
- Date edge
  - http://purl.org/dc/terms/date
  - http://purl.org/dc/elements/1.1/date

The value of the node at the origin of the *date edge* should be a valid date as specified by `xsd:date`. The plugin expects to find one node which has multiple incoming edges where the origin is an *event node* and it is required that it has a label. The type of the incoming edges is irrelevant. It can have additional outgoing edges as well, containing auxilliary data. An example of such a graph is shown in Figure 5.2.

When rendering the timeline, the value of the label node is used as the headline for the event and the value of the date node is used to place the event onto the timeline. Any statements of a type different than label or date are rendered as a HTML list under the headline. The resulting visualization is shown on Figure 5.3. `rdf:type`, `lb:legislativeProcessStage`, `frbr:realizationOf` and `lb:outcome` are additional properties of the event which where selected in the analysis.

[7] `https://github.com/siroky/Payola/blob/develop/docs/developer_guide.md#chart-plugins`

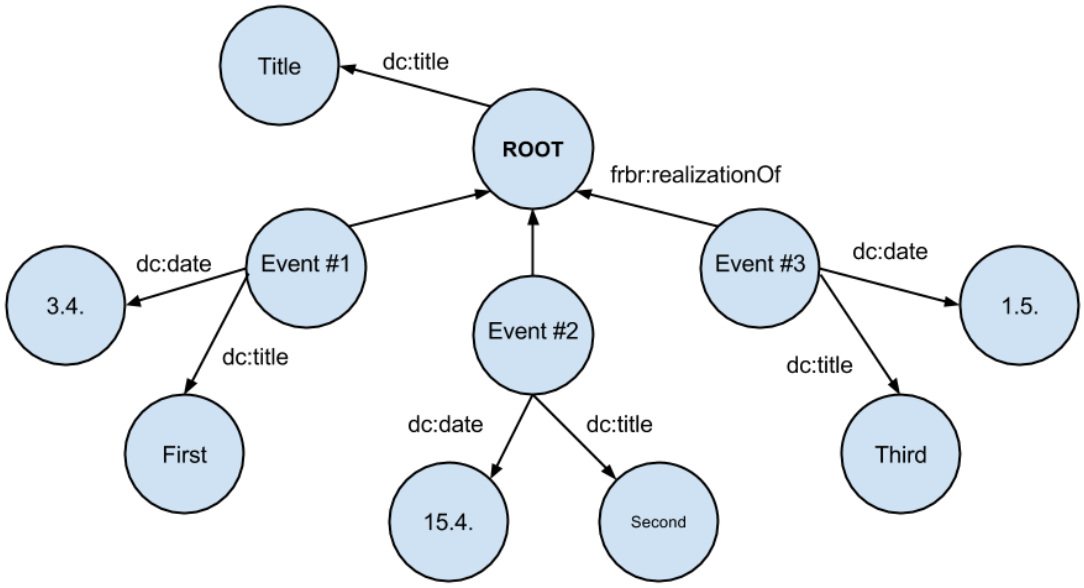


Figure 5.2: Structure of the analysis graph for the Timeline plugin

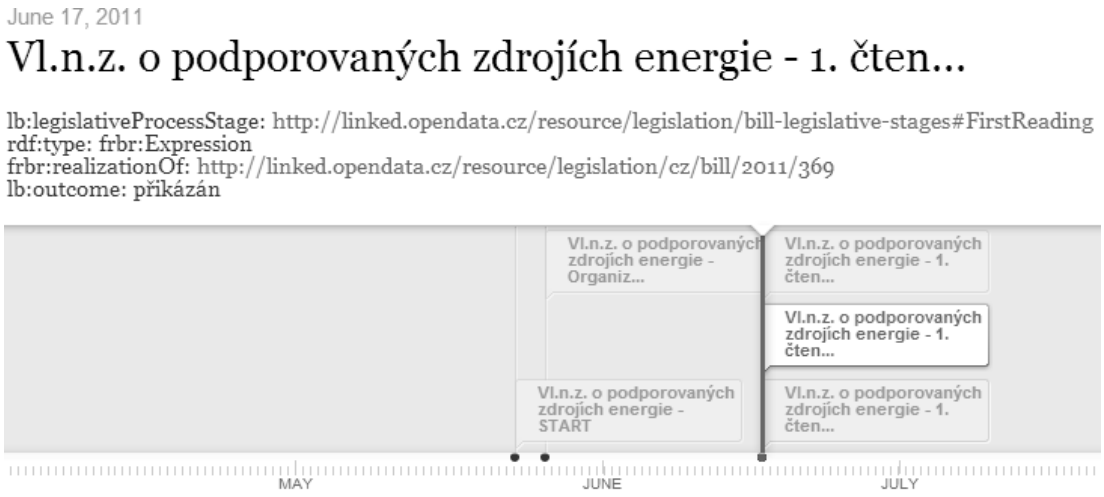


Figure 5.3: Legislative process visualization with Timeline JS

Making the root node optional

Received feedback from Payola developers included that the root node should not be mandatory. The plugin was generalized so it supports receiving a non-connected graph with multiple event nodes. First all vertices are scanned and event nodes are filtered, then, in a second pass, potential root nodes for the legend are found and the one with the most outgoing edges is selected.

The ASK SPARQL query in Example 5.3 will answer *yes* or *no* depending on if the supplied graph is in a format supported by the generalized plugin.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
ASK {
 {
  ?s ?labelprop ?l .
  ?s ?dateprop ?d .
  FILTER(
    (?labelprop = dc:title || ?labelprop = dcterms:title ||
       ?labelprop = rdfs:label)
    &&
    (?dateprop = dc:date || ?dateprop = dcterms:date)
  )
  OPTIONAL {
    ?legend ?labelprop ?ll .
    ?s ?anyprop ?legend
  }
 }
}
```

Example 5.3: SPARQL ASK query determining Timeline visualization support

5.4.2 Using the Timeline plugin to display the legislative process

For the general public, it is more convenient to find laws by their number assigned to them when they are enacted. To illustrate on a concrete example, let's assume we want to see the legislative process for the highly controversial law about Alternative energy sources support (Zákon o podporovaných zdrojích energie a o změně některých zákonů in Czech) enacted as 165/2012 Coll. The RDF resource representing the Act was created in the Lex Ontology project from the Czech OpenData initiative and can be found online at `http://linked.opendata.cz/resource/legislation/cz/act/2012/165-2012`. This will serve as the input parameter for the analysis.

The first branch of the analysis will select the Bill resource which lead to this Act by using the *Filter* plugin and the `lb:enaction` property. Then, the *Property selection* plugin will be used to add the title and bill sponsor to the result by selecting the `lb:billSponsor` and `dc:title` properties. This does not give the process yet, these are added using a second branch. With the *Typed* plugin, resources of type `frbr:Expression` are selected to create the root of the second branch. Their title, date and legislative process stage are added with the *Property selection* plugin. Finally, the two branches are merged using an inner *Join* plugin. The join is made using the `frbr:realizationOf` property. This gives us a graph which matches the requirements of the Timeline plugin as it has one root node (the Bill resource) with a label connected to several *event nodes*.

5.5 Public analyses

The timeline plugin with the legislative process displayed and statistical queries can be found on the Payola demo website. My public account profile at `http://live.payola.cz/profile/vojtavondra@gmail.com` lists created visualizations related to the Bill Ontology and a public datasource where the dataset is available through a SPARQL query endpoint.[8]

[8]Direct links are not guaranteed to work, but are generally stable. The following link leads directly to a rendering of the timeline plugin: `http://live.payola.cz/analysis/06bc11da-b34a-440f-a53e-148449702920#viewPlugin=cz_payola_web_client_views_graph_visual_TimelinePluginView`

Bibliography

[1] Berners-Lee, Tim. Linked Data. In: *W3 Design Issues* [online]. 2006, 2009-06-18 [cit. 2013-11-15]. Available from: http://www.w3.org/DesignIssues/LinkedData.html

[2] IFLA Study Group on the Functional Requirements for Bibliographic Records. Functional Requirements for Bibliographic Records. Mnichov: K.G. Saur Verlag, 1998. UBCIM publications ; new series, 19. ISBN 978-3-598-11382-6.

[3] Goel, Kavi and Pravir Gupta. Introducing schema.org: Search engines come together for a richer web. In: *Official Google Webmaster Central Blog* [online]. 2011, 2011-06-02 [cit. 2013-12-02]. Available from: http://googlewebmastercentral.blogspot.cz/2011/06/introducing-schemaorg-search-engines.html

[4] Berners-Lee, Tim and Robert Cailliau. WorldWideWeb: Proposal for a HyperText Project. World Wide Web Consortium (W3C) [online]. 1990 [cit. 2013-12-02]. Available from: http://www.w3.org/Proposal.html

[5] Knap, Tomáš and Jakub Klímek. Zasády pro tvorbu datových URI Prague, 2014. Available from: `https://docs.google.com/document/d/187nVDaKn_e24goqnwExCyiDZowDfnEoqauRWyu55G7M/`

[6] The LEX Core Ontology Cookbook. lex-ontology. [online]. 3.1.2014 [cit. 2014-05-08]. Available from: `https://code.google.com/p/lex-ontology/wiki/LEXCoreOntologyCookbook`

[7] Zelený kruh s.s. Poslanecké pozměňovací návrhy: Analýza Zeleného kruhu. 2014. Available from: `http://www.rekonstrukcestatu.cz/publikace/analyza-pozmenovaci-navrhy---zeleny-kruh.pdf`

[8] Helmich, Jiří, Jakub Klímek a Martin Nečaský. http://payola.github.io/Payola/ [online]. 2014 [cit. 2014-05-09]. Available from: `https://github.com/payola/Payola`

[9] SPARQL 1.1 Query Language. W3C. W3C [online]. 2013, 2013-03-21 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/sparql11-query/

[10] RDF 1.1 Primer. W3C. W3C [online]. 2014, 2014-02-25 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/rdf11-primer/

[11] Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C. W3C [online]. 2004, 2004-02-10 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[12] Namespaces in XML 1.0 (Third Edition). W3C. W3C [online]. 2009, 2009-12-08 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/REC-xml-names

[13] RDF/XML Syntax Specification (Revised) [online]. 2004, 2004-02-10 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/REC-rdf-syntax/

[14] Berners-Lee, Tim. Notation 3 Logic. In: *W3 Design Issues* [online]. 2005, 2005-09-27 [cit. 2014-05-09]. Available from: http://www.w3.org/DesignIssues/Notation3.html

[15] Berners-Lee, Tim. Turtle - Terse RDF Triple Language [online]. 2011, 2011-03-28 [cit. 2014-05-09]. Available from: http://www.w3.org/TeamSubmission/turtle/

[16] RDF Schema 1.1 [online]. 2014, 2014-02-25 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/rdf-schema/

[17] Brickley, Dan; Miller, Libby. FOAF vocabulary specification 0.98. Namespace Document, 2010, 9. [cit. 2014-05-09]. Available from: http://ontogenealogy.com/documents/2012/08/foaf-vocabulary-specification-0-98-20100809.pdf

[18] SKOS Simple Knowledge Organization System Primer [online]. 2009, 2009-08-18 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/skos-primer/

[19] Legislativní proces projednávání návrhů zákonů. Poslanecká sněmovna: Parlament České republiky [online]. 2013 [cit. 2014-05-09]. Available from: http://www.psp.cz/sqw/hp.sqw?k=331

[20] OWL 2 Web Ontology Language Document Overview (Second Edition) [online]. 2012, 2012-12-11 [cit. 2014-05-09]. Available from: http://www.w3.org/TR/owl2-overview/

[21] NUKnightLab/TimelineJS. Northwestern University Knight Lab . [online]. 2013 [cit. 2014-05-09]. Available from: https://github.com/NUKnightLab/TimelineJS

[22] An Introduction to RDF and the Jena RDF API. Apache Jena [online]. 2011 [cit. 2014-05-09]. Available from: http://jena.apache.org/tutorials/rdf_api.html

List of Abbreviations

API Application Programming Interface, specifies how programs communicate with each other

DC Dublin Core, a RDF vocabulary to describe basic metadata

FOAF Friend of a Friend, a RDF vocabulary for describing people

FRBR Functional Requirements for Bibliographic Records, a RDF vocabulary to describe bibliographical records and artistic works

HTML Hypertext Markup Language used to create websites

HTTP Hypertext Transfer Protocol, a network protocol for transmitting data over the Internet

IRI Internationalized resource identifier, the same as URI with Unicode support

OWL Web Ontology Language, a RDF standard to create ontologies

PSP The Czech Chamber of Deputies

RDBMS Relational database management system

RDF Resource Description Framework, a language to describe Semantic Data

RDFS RDF Schema, a language to validate and structure RDF data

SKOS Simple Knowledge Organization System is a set of standards for describing taxonomies in RDF

SQL Structured Query Language, which is used to read and write data from relational database management system

URI Uniform Resource Identifier, which is used to identify resources on the web

W3C World Wide Web Consortium, an international standards institution for the Web

XML Extensible Markup Language, a markup language used for encoding documents

A Attached media

The attached electronic version contains the following files:

thesis.pdf
: An electronic version of this document.

doc/
: Project documentation and generated Javadoc API documentation. The documentation index can be displayed by opening index.html in a web browser.

ontology/
: The ontology in TTL format and a generated dataset.

src/
: Source files for the converter application.

dist/
: Built JAR from the source files.

conf/
: In the `fuseki/` directory it contains the configuration file for Apache Jena Fuseki. In the `jenkins/` directory, configuration files for the build and converter jobs are stored.

payola/
: Contains submitted patches to Payola accepted by the developers. Patch #55 is a fix for the column chart plugin and #56 contains the timeline visualization.[9]. Github provides a easy-to-read diff view at `https://github.com/payola/Payola/pull/55/` and `https://github.com/payola/Payola/pull/56/`.

[9] the Timeline JS patch contains a copy of the TimelineJS plugin, original licenses and attribution were retained

B Running the converter

The converter can be executed by running the following command:

```
java -jar ld-bill.jar -i -n "dbhost/dbname" -u "dbuser" -p "dbpass"
-o "data.ttl"
```

Parameters

-n

MySQL database hostname and database name. The format is *hostname/dbname*. The database must be in utf-8.

-u

Database user

-p

Database password

-o

Target output file for dataset

-i

Import data into MySQL for conversion. This parameter can be ommited in subsequent runs if only output formatting is changed.

Building from source

Make sure Maven and Java SE 7 JDK is installed and run the following command in the `src/` directory:

```
mvn clean install
```

All dependencies should be downloaded automatically and a JAR file will be produced in the `src/target` directory.