Semantic Web: Project documentation

Marc Kirchmann and Sebastian Leitz

Cooperative State University Baden-Wuerttemberg
Master Computer Science
minf010@cas.dhbw.de, minf11@cas.dhbw.de

Abstract. We created a web interface that enables users to list and find data from a set of films and associated information. This data is stored within a server that the interface communicates with.
One part of the project was to find suitable data from public sources and store this in a format that can be understood by semantic frameworks such as Jena. We configured this software in an appropriate way so that it is able to read the stored data and provide this to other systems.
In a second step we designed and created a small web application that uses an API of the software from step one to fetch data from the server. The user can conduct simple search operations in that application which in turn communicates with the server API in order to find matching data.

1 Introduction

This document describes the authors' efforts to fulfill the tasks given in the assignment on the "Semantic Web" lecture. The assignment requests to create any kind of ontology using RDF and make this available for reading and querying to users via a website.

2 Structure and Usage of the Movie Ontology

Our semantic search website should enable a user to search for movies or other associated artifacts, e.g. a particular genre or a film's most famous actors. Thus an ontology is required which puts all of those things in relation.
We decided to adopt an existing ontology, called "MO - the Movie Ontology" [1]. It makes use of OWL [2] to map movie entities to classes and defines class hierarchies as well as predicates that show their relationship. To describe its elements, it uses other well-known ontologies, e.g. from "DBpedia" [3]. Although the ontology provides definitions and facts for a lot of entities, only a subset of them is used for the website.
The following sections introduce the entities that can be searched and some of their associated predicates. For the sake of readability, the prefix of a complete URI is omitted in this documentation, although it is used in the ontology.

2.1 Movie

A movie is represented as an OWL class. It is the central element of the ontology. Listing 1.1 shows how it is defined in an RDF graph that uses RDF/XML notation.

Listing 1.1. OWL Movie Class in RDF/XML notation

```
<owl:Class rdf:about="&www;Movie" />.
```

Most of the other parts of the ontology have a direct or indirect connection to it, i.e. a predicate describing their relationship. An example is given in listing 1.2. As a movie usually belongs to one or more genres, this is represented by the predicate belongsToGenre. The range and domain entries define the OWL classes whose instances are used as subject (domain → a movie) or object (range → a genre) in a statement that uses belongsToGenre as its predicate.

Listing 1.2. Exemplary Movie predicate in RDF/XML notation

```
<owl:ObjectProperty rdf:about="&movieontology;belongsToGenre">
  <rdfs:range rdf:resource="&movieontology;Genre" />
  <rdfs:domain rdf:resource="&www;Movie" />
</owl:ObjectProperty>
```

2.2 Actor

Another important part of the ontology is the OWL class for actors, shown in listing 1.3. It is a subclass of Person, which itself is part of DBpedia's ontology [4]. That means every Actor implicitly is a Person and thus inherits the properties of a Person.

Listing 1.3. OWL Actor Class in RDF/XML notation

```
<owl:Class rdf:about="&ontology;Actor">
  <rdfs:subClassOf rdf:resource="&ontology;Person" />
</owl:Class>
```

An actor or actress plays in a movie. This is what the predicate in listing 1.4 is about. It defines the relationship hasActor between a Movie and an Actor and its reverse definition:

- Movie hasActor Actor.
- Actor isActorIn Movie.

Listing 1.4. Exemplary Actor predicate in RDF/XML notation

```
<owl:ObjectProperty rdf:about="&movieontology;hasActor">
  <rdfs:range rdf:resource="&ontology;Actor" />
  <owl:inverseOf rdf:resource="&movieontology;isActorIn" />
  <rdfs:domain rdf:resource="&www;Movie" />
</owl:ObjectProperty>
```

2.3 Usage of the Ontology

Figure 1 provides an overview of how we utilize the movie ontology in the context of our movie search. Basically we use some of the relationships between the Movie and Actor or Genre entities and some individual properties like the birthDate of an Actor or the title of a Movie.

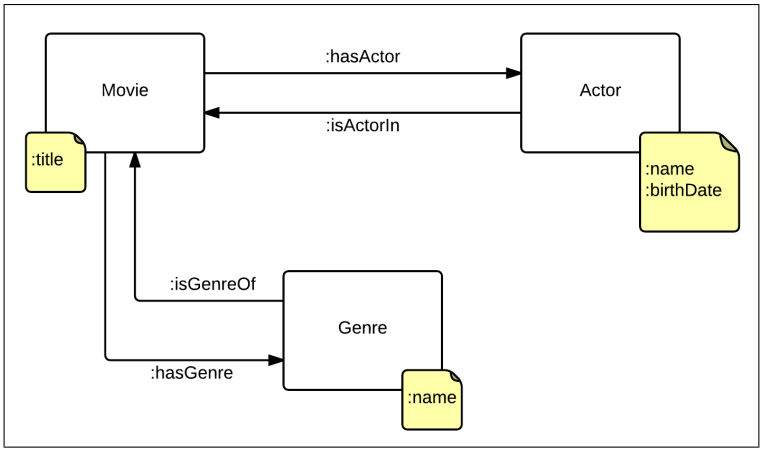

Fig. 1. Movie Ontology Overview

3 Project Setup

Our semantic search application is based on the Apache Jena framework, which is a framework for building applications for the semantic web. It provides the opportunity to store [5] and query [6] data from RDF graphs and perform reasoning based on RDFS or OWL to derive additional knowledge from existing instance data and class descriptions [7, 8]. Fuseki, which is one of the components of the Jena framework, offers a REST-based SPARQL endpoint [9]. Our application can send SPARQL queries via HTTP to perform semantic searches on the server.

We use Fuseki as data storage backend for our application. It ships with scripts to add RDF graphs to the knowledge base. After downloading the graph file *movieontology.owl* from [1], which represents the movie ontology as introduced in section 2, it could be added to Fuseki using a script file from the Fuseki root folder.

```
s-post http://localhost:3030/ds/data default movieontology.owl
```

To demonstrate the searching capabilities of our application, some exemplary instance data in RDF/XML notation was added, too.

```
s-post http://localhost:3030/ds/data default films.owl
```

Fuseki is set up to allow for reasoning on this ontology. [10] A user can query the semantic database by using a web interface that generates SPARQL queries, which is introduced in the next section.

4 Semantic Movie Search Website

Part of the given task was to make all the data described in the two previous sections available for user interaction via a website.

We started by defining the relevant functionality of the website in order to build an HTML prototype:

- Search field for user input
- Button to send the query to the server
- A subsection to display a list of results
- If required, a view giving details on a single result

The website involves a certain amount of dynamic information, so we decided to use a small server-sided PHP script to take care of any logical decisions. We also integrated a library for SPARQL queries in order to simplify our script. The library acts as an abstraction layer and takes care of many small items such as connecting to the server, checking for errors as well as evaluating and transforming the response from the server. [11]

4.1 The Search Interface

We constructed a simple markup which includes all items from above listing with HTML 5 and some basic CSS styles. HTML 5 takes care of a placeholder text within the search field to guide the user and also makes sure no empty queries are sent (required form field).

Each time the user makes a request to the website, a PHP script is called to action by the webserver. The script evaluates the current URI and the determines whether a query has been submitted by the user. This is decided based on the presence of the *query* variable.

If the query variable is not present, the script outputs the static HTML content of the search interface with an appropriate HTTP Content-type header.

Otherwise, the value of the query variable is extracted from the URI and used to fetch data from the server. In a first step, the user input is transformed into a SPARQL query, which is then evaluated by the Fuseki server.

4.2 Query Construction

The SPARQL query is constructed based on the input of the user. A single query is sent to Fuseki which returns all interesting datasets based on the ontology described in figure 1. A search does not necessarily result in a homogenous result set, but can contain URIs for various entities, e.g. movies and actors.
Our query approach includes the following steps:

Extract information Based on the entity, the query finds all interesting properties and relations to other entities. The query checks the type of a statement's subject and concludes, based on the movie ontology and stored instance data, that it belongs e.g. to the Movie class. Then it tries to fetch other mandatory or optional relations, e.g. the actors that play in this movie (i.e. Actor class instances that match the predicate introduced in 1.4).
Those relations depend on the subject the query found, i.e. Movie class instances have other dependent relations than Actor instances. The query therefore aggregates all intended class alternatives (Movie, Actor or Genre) and their respective predicates using the UNION statement, which provides the ability to query graph pattern alternatives [12].

Filter data All found instances at least need to provide a type and a description. Those fields are used to filter the search results of the query. Therefore, the user's input is interpreted as being a regular expression to filter the description field, which can be a movie's title, an actor's name or the name of a particular genre.

In addition to its URI, connected information is returned based on the entity which was found.
A movie also shows:

- the movie's title
- the URI of each genre it belongs to
- the URI of each actor

An actor also shows:

- the actor's name
- the URI of each movie he or she is actor in
- the actor's birth date (optional)

A genre also shows:

- the genre's name or description
- the URI of each movie that belongs to this genre

4.3 Response Evaluation

The query constructed by the above step is sent to the Fuseki server using the SPARQL abstraction layer as well.

The server returns a SPARQL result within an XML document. The abstraction layer parses this XML structure and inserts an XSL file reference. Our PHP script then sends the resulting new XML structure to the user's browser with an HTTP Content-type of *application/xml*.

The stylesheet is an extensive document which transforms the SPARQL XML into an entirely different structure for the website. One of the core challenges posed to this stylesheet was a mapping of multiple rows of the SPARQL result into one actual result item. See Listing 1.5 for the initial XML.

Listing 1.5. XML extract returned by the server when searching for a movie

```
<result>
 <binding name="uri">
  <uri>https://github.com/seaneble/dhbw_semanticweb#Titanic</uri>
 </binding>
 <binding name="desc">
  <literal>Titanic</literal>
 </binding>
 <binding name="actor">
  <literal>Leonardo DiCaprio</literal>
 </binding>
  [...]
</result>
<result>
 <binding name="uri">
  <uri>https://github.com/seaneble/dhbw_semanticweb#Titanic</uri>
 </binding>
 <binding name="desc">
  <literal>Titanic</literal>
 </binding>
 <binding name="actor">
  <literal>Kate Winslet</literal>
 </binding>
  [...]
</result>
```

There are two rows in the result for the same movie, because two different actors are starring in it. We wanted the HTML result to look like one movie entry with a list of actors.

The XSLT loops through all the result and evaluates each result's context to see whether there are previous or following elements of the same type. They are grouped together to form an HTML structure like in Listing 1.6.

Listing 1.6. HTML structure for one movie result

```
<div class="result_movie">
 <p><strong><a href="?query=Titanic">Titanic</a></strong> is a
 movie of the genre <em><a href="?query=Romance">Romance</a>
 </em>.</p>
 <h3>Cast</h3>
 <ul>
  <li>
   <a href="?query=Leonardo+DiCaprio">Leonardo DiCaprio</a>
   <a href="?query=Kate+Winslet">Kate Winslet</a>
  </li>
 </ul>
</div>
```

Our stylesheet also adds hyperlinks to all kind of data types. The user can navigate e.g. from a genre to a movie and further on to an actor using those hyperlinks. Each hyperlink performs a new search.

5 Conclusion

We were able to create a semantic search website through which users can navigate data and make queries at their leisure.

The data is stored in a Fuseki server instance and queried by a PHP script via the RESTful API of Fuseki.

All data is structured in accordance with the publicly available Movie Ontology. A set of sample data was loaded into Fuseki for example queries, yet storing all existing movies, genres, actors and their relationships would be possible, though complicated.

References

1. Bouza, A.: Mo the movie ontology (2010) [Online; 26. Jan. 2010].
2. n.A.: Tutorial on the basics of OWL and RDF. `http://www.linkeddatatools.com/introducing-rdfs-owl` (n.Y.)
3. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal (2014)
4. n.A.: About: persona. `http://dbpedia.org/ontology/Person` (n.Y.)
5. n.A.: Apache Jena - TDB. `https://jena.apache.org/documentation/tdb/index.html` (n.Y.)
6. n.A.: Apache Jena - ARQ - A SPARQL Processor for Jena. `https://jena.apache.org/documentation/query/index.html` (n.Y.)
7. n.A.: Apache Jena - Reasoners and rule engines: Jena inference support. `http://jena.apache.org/documentation/inference/` (n.Y.)
8. n.A.: Apache Jena - Jena Ontology API. `https://jena.apache.org/documentation/ontology/` (n.Y.)
9. n.A.: Apache Jena - Fuseki: serving RDF data over HTTP. `https://jena.apache.org/documentation/serving_data/` (n.Y.)
10. Baach, J.: Using Inference and Reasoning in Fuseki. `https://baach.de/Members/jhb/getting-started-with-jena-fuseki-and-owl-reasoning` (n.Y.)
11. Gutteridge, C.: The PHP SPARQL library. `http://graphite.ecs.soton.ac.uk/sparqllib/` (2012)
12. Harris, S., Seaborne, A., Prud'hommeaux, E.: Sparql 1.1 query language. `http://www.w3.org/TR/sparql11-query/` (2013)