Kim Hammar
12 January, 2018
ID2208 Programming Web Services

Programming the Semantic Web
A Tutorial

This tutorial will demonstrate how to build a basic semantic web application in Java, using the semantic web stack of RDF, OWL, SPARQL, XML and RDFS. The accompanying code is available here. Just as regular web services, semantic web applications are loosely coupled services, based on the HTTP transport protocol, using URIs as identifiers, and communicating through textual serialization formats. The characteristic parts of programming the semantic web is that there is an emphasis on *linking* between entities, and providing descriptions about the web content in formats designed for machines rather than humans. This, in combination with web standards, makes the web more accessible for machines, or *agents* if you will, to use the web. In the semantic web, a link is not just a pointer to some document, rather it describes a relationship between two concepts on the web, where concepts are finer than web documents. Equivalently, you can think of a link in the semantic web data model as a typed link in a larger web graph.

The Example Application

Lets assume that the domain for this semantic web application consists of two web services that represents courses at two different universities. The courses are linked by means of being part of a joint degree program (also a separate web service), consisting of only these two courses. We will make use of semantic web technologies to build the web services; publishing the data of the web services in both human-readable and machine-readable formats, linking the services with the RDF data model, and defining a shared vocabulary of semantics and concepts for the two services with an ontology in OWL. Finally, we'll manifest how a software agent can use the services as a shared database using RDF query languages like SPARQL.

Defining the Vocabulary

In its simplest form a vocabulary for a semantic web application consists of classes/concepts and different predicates between the classes. The purpose of the

vocabulary is to enable software agents to verify if two resources are equal or different, to make inferences based on relationship rules between entities, and to look-up the meaning of a relation between two entities. OWL and RDFS can be considered as semantic markup languages for defining such vocabularies, where RDFS is a subset of OWL. In essence, OWL and RDFS are ways of capturing knowledge about a data domain. As we'll see later in the tutorial, when we create the RDF documents, the linked data model of the semantic web is based on triplets of $\langle subject, predicate, object \rangle$, where all three components of the triple are identified with unique identifiers. As the components are identified in the form of unique URIs, it enables to link the triplets to vocabularies, or ontologies, defining the semantics of the subjects, predicates, and objects.

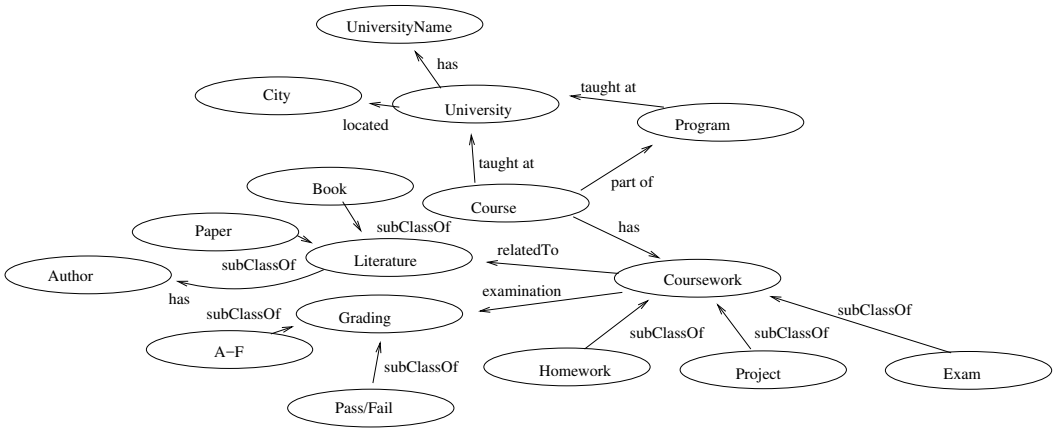


Figure 1: Ontology for the domain of our semantic web application.

OWL, RDF, RDFS etc are conceptual languages which can be serialized in different ways to make them accessible for machines and interchangeable over the web. The most standard serialization format is RDF/XML so that is what we'll use in this tutorial. Typically you define an ontology at a conceptual level and then let software tooling serialize it into RDF/XML format. I recommend using Protege[1] for defining the ontology.

The first thing to do when creating a new ontology in Protege is to define the identifier of the ontology. One of the core principles of the semantic web is to use URIs as identifiers (think of it as names rather than addresses of websites). Another

[1] https://protege.stanford.edu/

best practice is to provide information about a resource at its URI, since I plan to publish the ontology at an endpoint located at `http://localhost:8080/`, I use the URI: `http://localhost:8080/id2208/tutorial/ontologies/semwebprogram` as the identifier of the ontology.

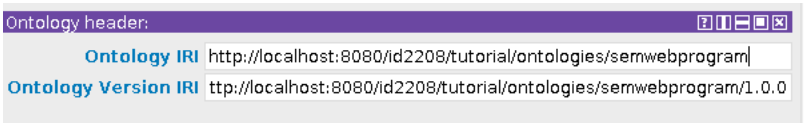
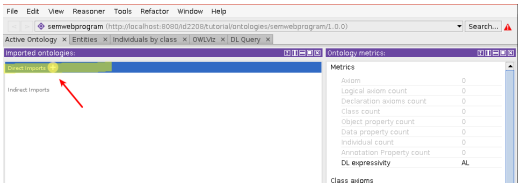
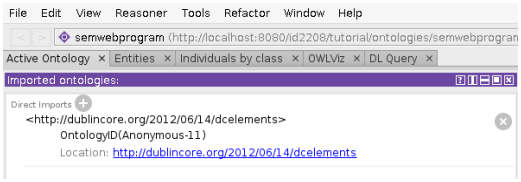


Figure 2: Defining the unique identifier of the ontology. One of the principles of the semantic web is to publish information about the resources at the same address as their URI.

Two other principles of the Semantic web are, (1) reuse existing vocabularies when possible (I relax this principle a bit for the sake of the tutorial); (2) and to declare the provenance of published resources. Without provenance of published resources, the chance that other people will trust to re-use or link to your resource is reduced. The Dublin Core (DC) [2] ontology is a vocabulary for describing generic meta-data and is well suited for declaring the provenance of our ontology. Lets import the DC ontology in Protege and define the provenance.



(a) Press the "+" sign, select *"import an ontology contained in a document located on the web"*, and use the URL `http://dublincore.org/2012/06/14/dcelements`. You should now have the DC ontology imported, as in figure 3b.

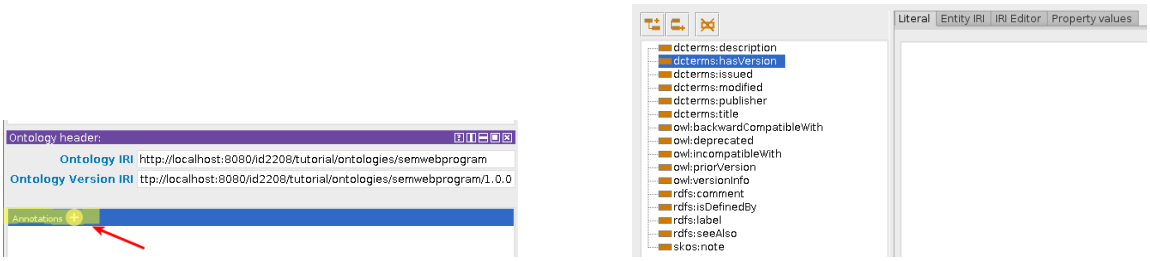


(b) The resulting view after importing the Dublin Core ontology.

Figure 3: Import the Dublin Core Ontology to reuse its properties for defining meta-data.

[2] `http://dublincore.org/2012/06/14/dcelements`

The next step is to add some meta-data to the ontology, as demonstrated in figure 4.



(a) Press the "+" sign to add annotation properties.

(b) Select the annotation properties to add and enter the values.

Figure 4: Add some meta-data to your ontology by defining annotation properties defined in the Dublin Core ontology.

After adding the meta-data to your ontology by using annotation properties, in Protege you should see a similar view as in figure 5.

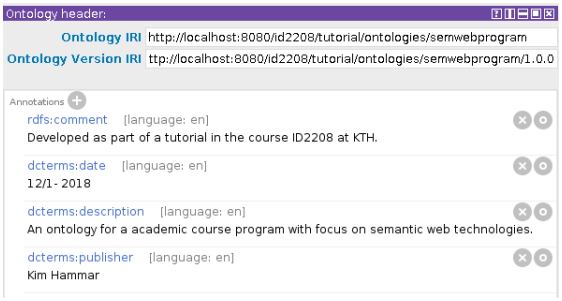
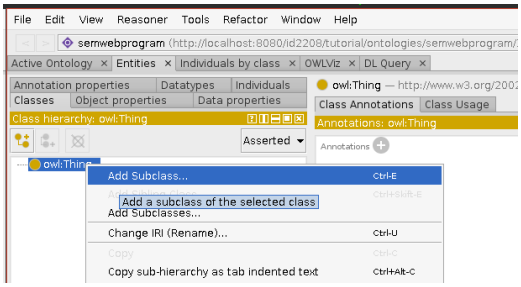


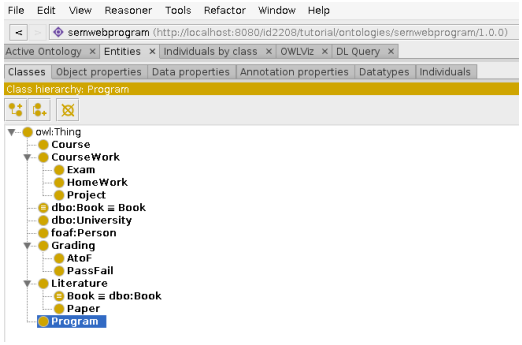
Figure 5: Example meta-data for the ontology, defined with annotation properties from the Dublin Core ontology.

The next step is to fill in the actual ontology, which is done in a similar fashion through the Protege UI. Start with defining the classes of the ontology. OWL classes describes sets of individuals, allowing you to define taxonomies of concepts in a similar fashion to how you define taxonomies of classes in object oriented program design. `owl:Thing` is the root class in OWL, to add classes to the ontology you add sub-classes to `owl:Thing`. A common practice is to use capitalized naming for classes. For proof of concept I will reuse two classes from open source ontologies. I use the

"University" class from the open source ontology at dbpedia[3], and to represent an "author" of the literature class in my ontology I will reuse the "Person" class from the FOAF ontology[4]. Consequently I end up with the class hierarchy in figure 6b.



(a) To add new concepts to the ontology in Protege, press the button to add sub-classes to `owl:Thing`.



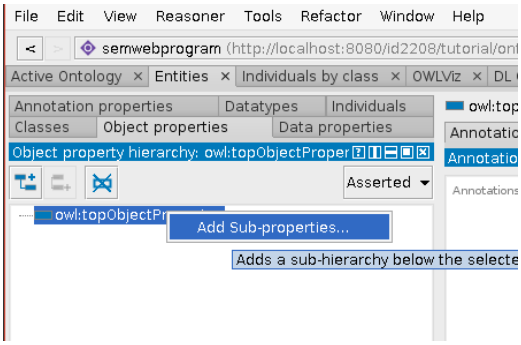
(b) Taxonomy of concepts for the domain of our semantic web application.

Figure 6: Adding concepts to the ontology.

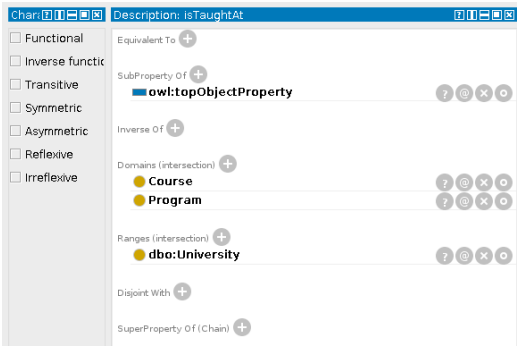
To reuse a class from an existing ontology, simply create a class as usual, but instead of specifying a short-name, specify the full URI, e.g `http://dbpedia.org/ontology/University`.

The next step to finalize the ontology is to add object properties and data properties to the created classes. Individuals in OWL are instances of classes, and properties are binary relations on individuals. Properties link two individuals together by relations. Object properties are properties that relates individuals of two classes to each other, while data properties are properties that relate an individual of a class to some literal value. A common naming practice for properties in OWL is to use names beginning with lower case letters and starting with "has" or "is". Just as classes, properties can be organized into hierarchies of sub- and super- properties. To create object properties in Protege, do as demonstrated in figure 7.

[3] `http://dbpedia.org/ontology/University`
[4] `http://xmlns.com/foaf/0.1/Person`



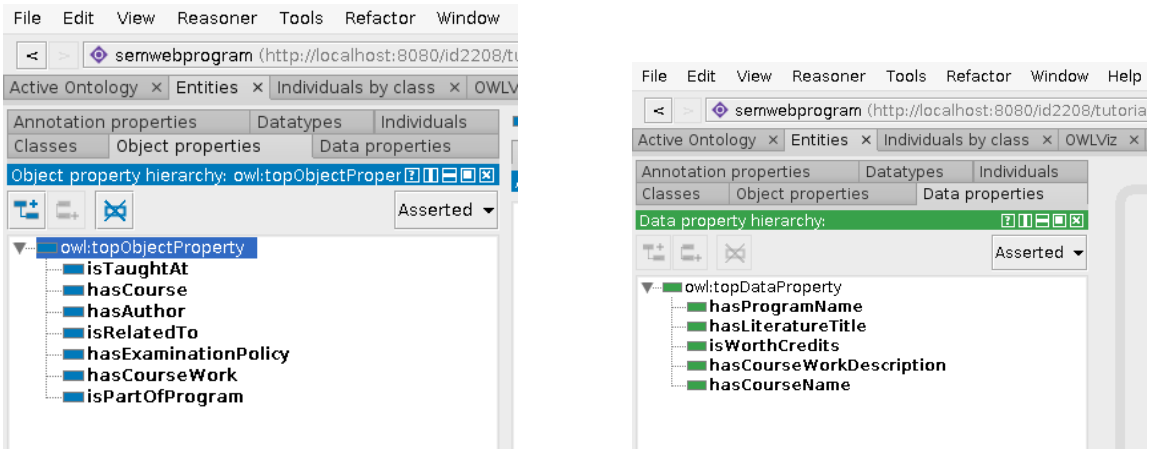
(a) Protege UI for creating object properties.



(b) Defining the domain and range of an object property in Protege.

Figure 7: Adding object properties to the Ontology in Protege.

The domain/range of an object property is equivalent to the corresponding properties of mathematical functions. The domain of an object property is the possible classes that can have the property, and the range of an object property is the possible classes that can be the value of the property. If the domain or range is not specified, all classes are possible. It depends on your application whether this type of detailed specification in the ontology is necessary. To create data properties in Protege, do the same procedure but now define the range of the properties as being literal values such as "string" or "integer". When finished, you should have the object and data properties listed in figure 8.



(a) Object properties. **(b)** Data properties.

Figure 8: Adding object properties to the Ontology in Protege.

Notice that we have not added any property to the classes from the external ontologies DBPedia and FOAF. This is because these ontologies already include a bunch of properties that will be re-used and we don't have to define them in our ontology.

That's it! You've built an ontology, now press "saveAs"→ RDF/XML to save and serialize the ontology. Figure 9 shown a visualization of the ontology.

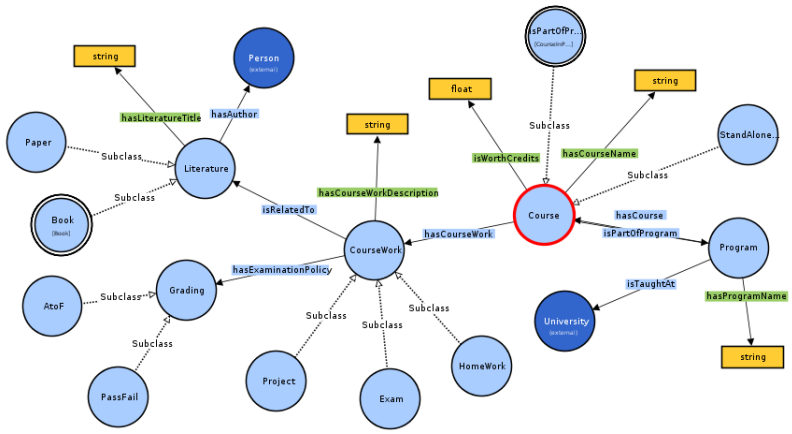


Figure 9: Visualization of semwebprogram.owl with VOWL (Protege plugin)

OWL is quite a powerful language and there are many steps we can do to improve the ontology and enable software agents to infer more about the domain we are

describing, but this works as a base ontology. You can find the complete ontology in RDF/XML format in the source code associated with this tutorial. To visualize or modify an existing ontology in Protege you can import it.

Refining the Ontology (Optional)

1. Add assertions to the classes, such as "sameAs", "subClassOf", "disjointWith" etc. For instance in our "Book" class we might add `owl:sameAs` http://dbpedia.org/ontology/Book. By default, OWL classes are overlapping, meaning for example that a single instance/individual in our domain can be both a Course and a Program at the same time. To forbid this, it is necessary to explicitly specify "disjointWith" for the classes in the ontology.

2. Add restriction classes, OWL allows to specify classes in terms of properties like "all individuals having property B and property C form a class", it is a way to define anonymous (unnamed) classes. The anonymous classes are useful since they can be used as sub or super classes in the taxonomy to add implicit semantics to the named classes. There are three different type of restrictions in OWL: quantifier (existential or universal) restrictions, cardinality restrictions, and hasValue restrictions. For instance, you could define a "CourseWork" of type "Exam" and grading policy "PassFail" as an anonymous "PassFailExam" class. Another example of using quantifier restrictions is to express that a program must have at least one course by first defining a restriction class representing all programs that have at least one course, and then declaring "Program" to be a sub-class of the anonymous restriction class [5].

3. Add assertions to the object and data properties. In owl it is possible declare that properties are symmetric, asymmetric, reflexive, etc. For example, we can add the assertion that the object property "program hasCourse" and "course isPartOfProgram" are the inverse of each other, implying that if program A hasCourse B, we can infer that course B is part of program A. We could also specify that a coursework instance can only have one examination policy and not multiple, this is defined by declaring the property "hasExaminationPolicy" as being a functional property. You can imagine how this could be used to make inferences about data, *if A has examinationPolicy B* ***and*** *A has examinationPolicy C* ***then*** *C=B*. We could also define a "StandAloneCourse" as a course not belonging to any program by creating a subclass of the class "Course" and

[5] To do this in Protege, select the class "Program" and then click the plus-sign to create a new "SubClass Of" and there you are faced with an editor for creating object restrictions.

then adding a cardinality constraint that a "StandAloneCourse" should be part of maximum zero programs. For data properties, the most common assertion is to constrain the range of values. For instance we could add an assertion that a course must have ≥ 2 credits.

4. Add meta-data to the classes and properties in form of annotation properties like "comment", and "seeAlso". You can consider annotation properties as meta-data variants of data- and object- properties. The annotation properties are not intended to be used for reasoning.

5. Add individuals, in our ontology there is currently no instances of the classes. In this tutorial there will be no individuals added to the base ontology, rather the individuals of the classes will be defined by the different web services that uses this shared ontology.

6. Process the ontology by a reasoner to verify that the ontology is consistent. The reasoner can help you with this by inferring as much as possible from the facts you have stated, and if the reasoner can infer something that is unsatisfiable it means that your ontology is inconsistent. Protege has a built-in reasoner for doing this. For instance in my ontology the reasoner inferred to me that the "Book" class was equivalent to owl:Nothing, meaning that no individual could ever fulfill the class requirements. The reason for this was that I had accidentally put my "Literature" class as disjoint with "dbo:Book" and at the same time declared my Book class as equivalent with "dbo:Book", this is unsatisfiable since the "Book" class is also a subclass of the "Literature" class.

The Logic Interpretation of Ontologies (Optional) OWL has formal mathematical foundations in Descriptive Logics[6], which explains much of the design decisions made in OWL. Description Logics is a fragment of first-order predicate logic. Description logic is less powerful and less elegant than first order logic. The reason for using description logics to describe ontologies is because it is decidable, as opposed to full first order logic which is undecidable. If there is one thing you should know about description logic when interpreting ontologies from the logic perspective, it is that description logic makes the open world assumption (OWA). As opposed to most other logic formalisms that rely on the closed world assumption. What this

[6] https://en.wikipedia.org/wiki/Description_logic

means is that all assertions made in an ontology are considered true, but assertions not made can not be interpreted as false, rather they are treated as unknown. You can think of the open world assumption as follows. When the ontology is empty, we know nothing about the domain and everything is possibly true. When we gradually add things to the ontology, the domain gets more and more constrained and specified. In addition, OWL does not make the unique name assumption (UNA), meaning that two concepts in OWL can be equivalent despite having different names (declared with owl:sameAs). Some examples of how we can interpret ontologies as logical knowledge-bases are given in table 1. Classes in OWL represents sets of individuals, and the class owl:Thing ($\top$ in description logic) is the set that represents all individuals. Thus, all classes are sub-classes of $\top$. Additionally, owl:Nothing is the class representing the empty set of individuals ($\emptyset$), denoted $\bot$ in description logic. In summary, desciption logic is based on three types of elements (1) individuals/constants; (2) concepts/unary relations; (3) roles/binary relations. As these elements are combined they form a logical knowledge base, filled with terminology definitions and assertions.

OWL	Logic Rule	Description Logic
Class:Male(pascal)	$Male(pascal)$	$Male(pascal)$
Male subClassOf owl:Thing	$Thing(pascal) \leftarrow Male(pascal)$	$Male \subseteq \top$
Male superClassOf owl:Nothing	$Male(pascal) \leftarrow Nothing(pascal)$	$\bot \subseteq Male$
hasHusband subPropertyOf Married	$married(x, y) \leftarrow hasHusband(x, y)$	$hasHusband \subseteq married$
Book sameAs dbo:Book	$Book(x) \leftarrow dbo : Book(x) \quad dbo : Book(x) \leftarrow Book(x)$	$Book \equiv dbo : Book$
Husband = Married intersection Man	$Husband(x) \leftarrow Married(x) \wedge Man(x)$	$Husband \equiv Married \cap Man$
Parent = Mother union Father	$Parent(x) \leftarrow Mother(x) \vee Father(x)$	$Parent \equiv Mother \cup Father$
NotMother = Complement Mother	$NotMother(x) \leftarrow \neg Mother(x)$	$NotMother \equiv \neg Mother$
Mother = Restriction: hasChild, Female	$Mother(x) \leftarrow \exists y \wedge hasChild(x, y) \wedge female(x)$	$\exists hasChild.Female$
Mother = Parent intersection Woman	$Mother(x) \leftarrow Parent(x) \wedge Woman(x)$	$Mother \equiv Parent \cap Woman$

Table 1: Some examples of how OWL statements can be interpreted as logical rules and assertions.

OWL Version Zoo (Optional) In some cases it is important to distinguish which OWL version is used. OWL 2 is the latest OWL version, but there are different subsets of OWL 2 that are given different names. RDF is a subset of OWL Full, OWL DL is a subset of OWL Full that restricts OWL Full to a subset that is correct with respect to Description Logic. OWL Lite is a subset of OWL DL, designed to enable software packages to be able to enjoy most of OWL but still being rather simple to implement. To summarize, OWL Lite $\subset$ OWL DL $\subset$ OWL Full. Protege supports editing all of OWL versions, and if you create an ontology with the basic operations inside Protege you most likely create an ontology conforming to OWL-DL.

Semantic Web Reasoning (Optional) Reasoning refers to the action of some software agent to infer knowledge that is not explicitly stated. Inference and reasoning is just a tiny part of the semantic web and the majority of applications does not make use of it. An example of a semantic reasoner is a software agent that go through an ontology or RDF data-set to make all inferences it can and try to discover inconsistencies. OWL is powerful but still quite limited in the type of reasoning and inference it can enable. In particular, closed world reasoning is not possible in pure OWL. Therefore, it is common for more advanced semantic web reasoners to add rules, proofs and other logical frameworks on top of the underlying OWL knowledge-base.

Hash URLs As mentioned, HTTP URIs are used as identifiers in the semantic web, and web-services in general. A good practice, and one of the linked data principles, is that URIs should be dereferenceable. Meaning that a client or agent that is given a RDF triple can lookup the concepts in the relation (subject and object), and also look up the meaning of the predicate in some ontology. The granularity of resources that are given unique identifiers is much more fine-grained than whole documents. A single RDF or OWL file typically contains many concepts and relations with different URIs. To not mix up the actual web documents containing the concepts and the individual entities when dereferencing URIs that are embedded in larger documents, two approaches exists. The first option is to have the server respond with a HTTP response code `303 See Other`, and a link to the web document in which the entity with the given URI is published when it receives a HTTP request for a URI that is embedded in some other web document. The second approach is to use so-called "hash URIs". The hash URI strategy means that a URI contains two parts, one base part indicating the document URL, and one fragment part identifying the individual entity in the document. The two parts are separated by the hash symbol "#". When dereferencing a hash-URI, only the base part of the URI is used when sending the HTTP request. In Protege, by default the hash-URL strategy is used, and it seems to be the most popular approach in practice. Figure 10 shows a snippet from the ontology and how the hash-URIs look.

Defining the Semantic Web Services

One of the big benefits of the semantic web and linked open data is that the data can grow in a distributed fashion, where each service can add data to the linked data graph of the web independently, while still allowing clients (agents) to treat the data as a large collection by dereferencing the URIs in the graph. Thus in reality, the

Figure 10: A snippet from semwebprogram.owl. A URI like http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#Project refers to the Class "Project" inside the ontology hosted at http://localhost:8080/id2208/tutorial/ontologies/semwebprogram.

```
<!-- http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#Project -->

<owl:Class rdf:about="http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#Project">
    <rdfs:subClassOf rdf:resource="http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#CourseWork"/>
    <rdfs:comment xml:lang="en">Project assignment part of coursework</rdfs:comment>
    <rdfs:label xml:lang="en">Project</rdfs:label>
    <rdfs:seeAlso xml:lang="en">http://dbpedia.org/ontology/Project</rdfs:seeAlso>
</owl:Class>
```

two institutions, the ontology, and the academic program of figure 11 would be four different services, perhaps distributed geographically. However, to make it simple in this tutorial, the different services are represented as different endpoints on the same service and host.

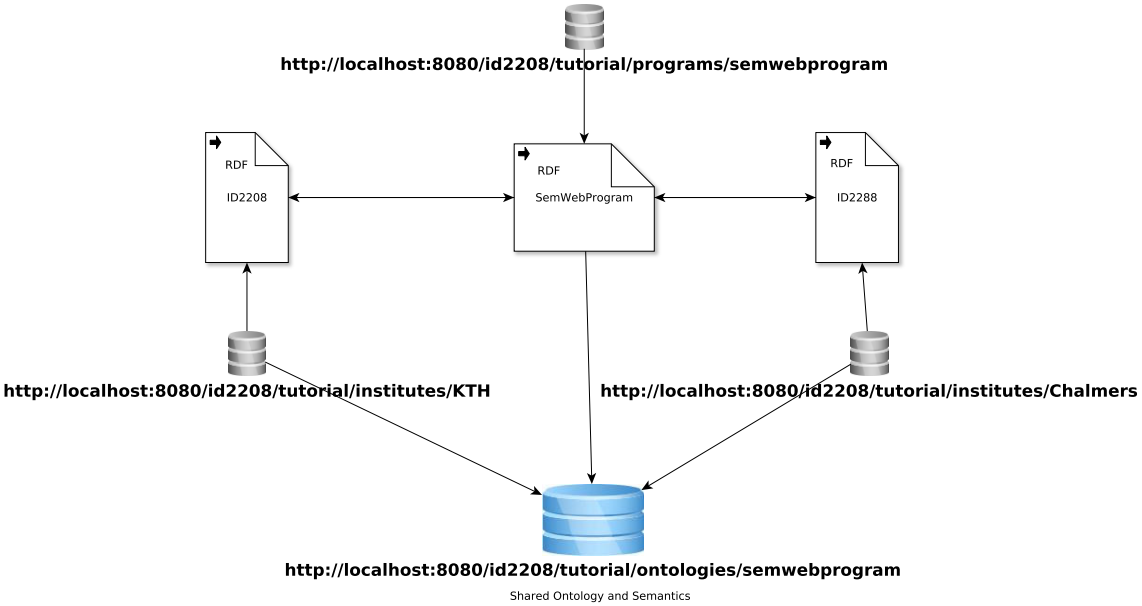


Figure 11: The small world of web-services in our domain.

The Boilerplate In a real-world application the different services would likely fetch the data from some database and convert it to RDF automatically, or they might even have the data stored as RDF directly in a triple store. In this tutorial, no database is used for simplicity. Instead, the services creates their mock-up data

as in-memory Java objects, and then convert it to RDF format using Apache Jena[7]. The boilerplate code to setup for this tutorial is listed below [8].

- A java data model for representing the data. This can be a one-to-one mapping between classes in the ontology and Java classes, as well as mapping between object and data properties of the classes to java class variables.
- Four different web-services or endpoints.
  - `http://localhost:8080/id2208/tutorial/institutes/kth`,
  - `http://localhost:8080/id2208/tutorial/institutes/chalmers`,
  - `http://localhost:8080/id2208/tutorial/ontologies/semwebprogram`,
  - and `http://localhost:8080/id2208/tutorial/programs/semwebprogram`.

  Each endpoint will host its data both as RDF and text/html, serving the appropriate type based on the accept-header of the HTTP request.

Converting the Data to RDF First, let's just recap what RDF is, RDF is a flexible data model that is centered around the idea of *linking*. In RDF, you model data with RDF-triples consisting of $\langle subject, predicate, object \rangle$. Unlike usual HTTP links, RDF links connects fragments/concepts of documents together. In RDF you model your data as graphs of RDF-triples that are inter-linked, and considering that you'll more often than not have some link to an external graph, you can view the whole semantic web as a giant graph. If we just zoom in on the concept `http://localhost:8080/id2208/tutorial/institutes/kth#ID2208` we get the graph view of figure 12. An important detail in RDF is that the links are typed, typically a link between two concepts can be looked up in an ontology. This idea of typed linked is really key to the semantic web as it allows to represent data from diverse domains with completely different schemas in the same graph. That is really all there is to RDF, there is also some variants to the simple triples in RDF such as using collections or reification, but these techniques are not widely used and some people even advocate to avoid using them and instead sticking to the basic RDF-triples. RDF is a data-model and there are multiple ways of serializing it in order to be able to send it over the network, or saving it in a file. Some example serialization formats of RDF are RDF/XML, RDF/a (html), Turtle, N-triples, RDF/JSON.

[7] `https://jena.apache.org/`
[8] See example code in the repository.

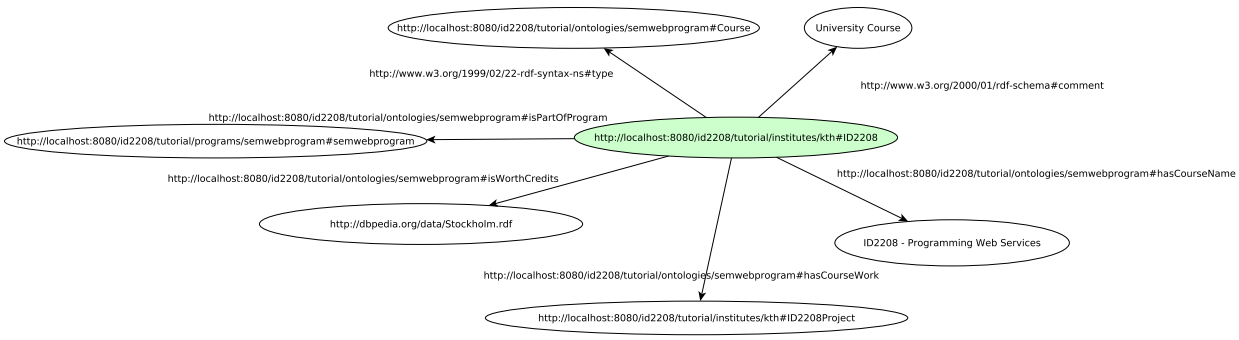


Figure 12: RDF graph

Apache Jena is a free and open source Java framework for building semantic web and Linked Data applications. The framework is quite sophisticated and includes several different APIs, in this tutorial we'll only need to scratch the surface of those APIs. In particular, Apache Jena has a RDF API (the API we make most use of), an ontology API (that we'll make some use of in this tutorial), a SPARQL API (which we will also use), an inference API (we will not use it), and a storage API (we will not use it).

The Jena RDF API is quite straightforward, with classes for representing a Resource, a Property, and a Literal. Combinations of multiple triples with resources, properties, and literals forms a graph, which is called a *model* in Jena. A triple of Resource, Property, and Literal or Resource, is called a *statement* in Jena. Figure 13 contains a code snippet using the Jena RDF API to programatically build a RDF graph from data stored in a data object representing a Course in our domain. The code in figure 13 should be quite straightforward except the `OntModel`

Figure 13: Code snippet demonstrating how to create a RDF graph programatically with the RDF API of the Apache Jena framework.

```
public Resource toRDF(OntModel ontModel, Model rdfModel) {
    Resource course = rdfModel.createResource(ID);
    course.addProperty(RDF.type, ontModel.getOntClass(SemWebProgramOntology.Course));
    course.addProperty(RDFS.comment, "University Course");
    course.addProperty(ontModel.getProperty(SemWebProgramOntology.isWorthCredits), Float.toString(credits));
    course.addProperty(ontModel.getProperty(SemWebProgramOntology.hasCourseName), courseName);
    for (CourseWork cw : courseWork) {
        course.addProperty(ontModel.getProperty(SemWebProgramOntology.hasCourseWork), cw.toRDF(ontModel,rdfModel));
    }
    course.addProperty(ontModel.getProperty(SemWebProgramOntology.isPartOfProgram), program.ID);
    course.addProperty(ontModel.getProperty(SemWebProgramOntology.isTaughtAt), university);
    return course;
}
```

and `SemWebProgramOntology` classes, which brings us to the ontology API in Jena.

`SemWebProgramOntology` is just a class I created with some string constants representing the URIs needed to create the graph. The ontology API in Jena is an extension to the RDF API with some add-ons for dealing with ontologies. Just as in the RDF API, an ontology graph is represented with a *model*. Specifically, a model in the ontology API is an instance of the `OntModel` class. In the code of figure 13 the `ontModel` instance represents the semwebprogram ontology loaded into memory[9].

Serving the Data Once the RDF graph is created, the web services can serve the RDF as RESTful resources, demonstrated in figure 14. The services should ideally return different formats of the data depending on the accept-header of the HTTP request.

Figure 14: Serving a RDF document as a RESTful resource.

```
@GET
@Produces(MediaType.TEXT_HTML)
public String kth() {
    StringWriter stringWriter = new StringWriter();
    rdfModel.write(stringWriter, "RDF/XML");
    //TODO this should be some more user-friendly format, e.g a HTML table
    return "<h1> KTH Courses</h1>\n" + stringWriter.toString();
}
@GET
@Produces("application/rdf+xml")
public String arlanda() {
    StringWriter stringWriter = new StringWriter();
    rdfModel.write(stringWriter, "RDF/XML"); //Serve raw RDF is just a single call to rdfmodel
    return stringWriter.toString();
}
```

SPARQL

The standard query language to concisely be able to extract data in large databases of RDF data is called SPARQL. Those of you who have ever used SQL should have no problem reading SPARQL code. SPARQL is declarative just like SQL and re-uses a lot of terminology from SQL. I like to think of SPARQL as pattern matching, where you specify some pattern that you would like to extract from the data-store and the SPARQL engine will search for triples matching the given pattern and return it. To specify the unknowns in the pattern (the part of the triples you want to select) you use SPARQL variables, denoted with a "?". SPARQL provides a full set of relational algebra operations just like SQL, such as SELECT, JOIN, AGGREGATE etc. With the PREFIX keyword, it is possible to specify prefixes of namespaces, making the query more concise. Figure 15 represents a SELECT query matching the triple

[9] see the source code project for an example of how to load the ontology from disk into memory with the Jena API.

pattern of $\langle semwebprogram, isTaughtAt, X \rangle$. At http://dbpedia.org/sparql you can utilize the SPARQL endpoint of DBPedia to practice SPARQL queries against the DBPedia triple store.

Figure 15: A SPARQL query for selecting all universities of a program.

```
PREFIX smp2: <http://localhost:8080/id2208/tutorial/programs/semwebprogram#>
PREFIX smp: <http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#>

SELECT ?uni
WHERE
{
  smp2:semwebprogram smp:isTaughtAt ?uni
}
```

Using the Services

Now the services are ready to be consumed by clients. There are three ways a software agent can consume semantic web-services with the Jena API: (1) The agent can download the raw RDF into a Jena model (a one-liner) and programmatically search through the RDF graph; (2) The agent can download the raw RDF into a Jena model and search through the RDF graph with Jena's SPARQL API; (3) The agent can send SPARQL queries over the network directly to the web-service, this requires that the web-service has a SPARQL endpoint and is capable of answering queries. In the sample application of this tutorial, the services only serve raw RDF and has no SPARQL endpoint so that rules out the last alternative. Figure 16 demonstrates how you can download an RDF file over the network and searching it programatically with Jena's API.

Figure 16: Searching local RDF graph programatically with the Jena API.

```
//Download the RDF from the service located at programURI
TypedInputStream inputStream = HttpOp.execHttpGet(programURI, "application/rdf+xml");
Model programModel = ModelFactory.createDefaultModel();
programModel.read(inputStream, null);
//Select triples with RDF type http://localhost:8080/id2208/tutorial/ontologies/semwebprogram#Course
Selector programSelector = new SimpleSelector(null, RDF.type, ontModel.getOntClass(SemWebProgramOntology.Program));
return programModel.listStatements(programSelector).nextStatement().getSubject();
```

Programmatic search through large RDF graphs is quite tedious, a better alternative then is to use SPARQL. Figure 17 shows how to use SPARQL to search through a RDF graph in Jena.

Figure 17: Example of how SPARQL can be used to query RDF graphs in Jena.

```
//Download the RDF from the service located at programURI
TypedInputStream inputStream = HttpOp.execHttpGet(programURI, "application/rdf+xml");
Model programModel = ModelFactory.createDefaultModel();
programModel.read(inputStream, null);
//Select triples universities from a RDF file representing a program
String queryString = "PREFIX smp: <" + DataUtils.ontNS + ">\n" +
                     "PREFIX smp2: <" + DataUtils.semWebProgramNS + ">\n" +
                     "SELECT ?uni WHERE {\n" +
                     "smp2:semwebprogram smp:isTaughtAt ?uni .\n" +
                     "}";
Query query = QueryFactory.create(queryString);
QueryExecution qexec = QueryExecutionFactory.create(query, programModel);
ResultSet results = qexec.execSelect();
Map<Integer, String> sortedUnis = new TreeMap<Integer, String>();
while (results.hasNext()) {
            QuerySolution querySolution = results.nextSolution();
            String uni = querySolution.getLiteral("uni").getString();
    }
```

With these primitives for consuming RDF with Jena you can quite succinctly write clients that exploits the real power of semantic linked data, namely treating multiple independent web services as a shared global data space by dereferencing the links between nodes. In the source code accompanying this tutorial you'll find two example queries: (1) a query where the agent is asked to collect all coursework necessary to complete a degree, and the agent does this by dereferencing the links to each of the individual courses part of the program and aggregating all the coursework, and (2) a query where the agent collects information about all universities part of a program, it does so by dereferencing links to DBPedia entries of the universities, and making use of the SPARQL endpoint of dbpedia[10] to collect data from DBPedia.

Going Further

Some next steps to extend on this basic application (out of scope for ID2208 project).

SPARQL Endpoint Currently the services serve their RDF data as raw RDF files over HTTP, an extension is to add a SPARQL entry-point to allow the client to make SPARQL queries directly to the services.

Triple store In this application the services hold the RDF data in memory as Java objects, in practical applications you'll want to store the data in a triple store and have the services read the data from there when serving client's requests.

[10] http://dbpedia.org/sparql

Semantic Reasoner You could add some predicate logic and build logical rules on top of the current storage, enabling to do inference about the data.

More Data Obviously more data and larger graphs would make the application more interesting, currently the query possibilities are limited simply by the lack of data.

Semantic XML-based Web Service In this tutorial we built a semantic RESTful web application, you could apply the same ideas when building XML-based web services. In particular, when building semantic XML-based web services, there exist standardized semantic annotations for WSDL in the form of SAWSDL, and a standardized ontology in form of the OWL-S ontology.