



A DATA ENGINEER'S GUIDE TO SEMANTIC MODELLING

Written by Ilaria Maresi

June 2020



About the author



Ilaria Maresi has been a Data Engineer at **The Hyve** since November 2018. A mathematician by training, Ilaria has always been fascinated by research at the intersection of biology, mathematics and engineering. This multidisciplinary interest brought her to **The Hyve**, where she has been developing semantic models and knowledge graphs for the company's pharmaceutical clients. She believes in using linked data to make a positive impact in healthcare and drug discovery, and hopes to show you in this guide just how awesome linked data can be.

Welcome to semantic modelling

Hi! And welcome to this guide on semantic modelling. Let me first explain the origin of this guide. For me, learning about semantic modelling involved more websites than I can count, a few dozen videos, a book, an in-person course, an online course and more than a few extremely patient people. There was never one definite resource where I could quickly look something up and I often forgot where I had seen or read something. More importantly, I found many of the resources to be extremely technical, geared towards a more experienced audience, while others were too wishy-washy. I decided to collect the most useful lessons in one place and to explain what I had learned in simpler terms that made sense to me. That's how this guide was born.

What does it cover? We start off with the basics: what is a semantic model and why should you consider building one. In Chapter 2 we delve into the technical basics, i.e. the Resource Description Framework, Uniform Resource Identifiers, ontologies and more. In Chapter 3 we put what we've learned into practice and discuss how to build your first semantic model. Chapter 4 covers how to query the model you've just built.

Throughout this document you will find links to resources I found useful in my learning process, which are often more detailed and go deeper into the technical side of things. In the Appendix you can find more learning resources and a list of abbreviations.

And with that, I leave you to your reading and modelling. Enjoy, good luck, and have fun!



Table of content

About the author	2
Welcome to semantic modelling	2
CHAPTER 1: A NOT-SO-BRIEF INTRODUCTION	5
What is a semantic model?.....	5
Why should I want to build one?	5
A semantic Led Zeppelin	6
CHAPTER 2: SEMANTIC MODEL BASICS	8
Understanding RDF	8
Almost everything needs a URI	9
RDFS	17
OWL.....	19
Do not reinvent the ontology wheel	21
What is an ontology?	22
Reusing existing ontologies	22
CHAPTER 3: BUILDING A SEMANTIC MODEL	24
RDF syntax	24
Turtle.....	24
Semantic models and ontology builders	27
Protégé	27
Building the model - a step-by-step guide	28
Step 1: Determine the domain and scope (a.k.a use cases)	29
Step 2: Determine important concepts in the model	30
Step 3: Reuse ontologies	30
Step 4: Define classes, class hierarchy and properties	30
Step 5: Define constraints.....	31
SHACL.....	31



Visualising your semantic model	34
CHAPTER 4: QUERYING A SEMANTIC MODEL	36
SPARQL	36
DISTINCT.....	39
OPTIONAL	40
UNION.....	41
FILTER.....	42
VALUES.....	43
ORDER BY.....	44
LIMIT	45
GRAPH.....	45
Beginning your semantic modelling journey	47
Acknowledgements	48
References	49
Appendix	50



CHAPTER 1: A NOT-SO-BRIEF INTRODUCTION

What is a semantic model?

Ironically, definitions tend to vary, though it is generally agreed that:

*"Semantic models of data sources represent the implicit meaning of the data by specifying the **concepts** and the **relationships** within the data."*

1

Essentially, we use semantic models to impart meaning to our data by making it explicit what the data represents, i.e. the **concepts**, and the **relationships** between these concepts. It is imperative that the meaning is understood by both humans and machines. Imparting meaning to a human is easy, or relatively easy I should say, in that we can use figures and words to describe whatever it is we want to model. Imparting meaning to a computer takes a bit more thought. Enter semantic models.

Why should I want to build one?

Giving our data a context is important. Take for instance the word jaguar: are you talking about Jaguar the car or jaguar the animal? Are you talking about the band The Beatles or a group of insects? How do I know you're talking about "Black Dog" the song by Led Zeppelin and not describing a dark-furred canine friend? Of course, as humans, we are able to understand the true meaning given the context of the information. If I tell you: 'I ride my Jaguar ² to work' then you assume I'm probably not roaming around the streets of Utrecht on a giant cat. But maybe I do. Who knows. And that is exactly the point - data by itself, without context, isn't always clear. So let's use semantic models to give meaning to our data, and avoid any potential confusion.

¹ *Semantic Modeling: Automatically building semantic descriptions of sources.* (n.d.). Retrieved April 30, 2020, from <https://usc-isi-i2.github.io/semantic-modeling/>

² Unfortunately, the author does not *actually* own a Jaguar.



A semantic Led Zeppelin

The best way to explain semantic models is through example, so let's build a very small model representing a band of interest. Figure 1 depicts a few **concepts** relating to Led Zeppelin and the **relationships** between these concepts. We see that Led Zeppelin is a band, who recorded an album called "Led Zeppelin IV" that was released on 8th November 1971. One track on the album is "Black Dog". Jimmy Page, who is a person, is a member of the band. Of course, this is only a partial picture of all the data we could use to describe this band, its members and its music.

Semantic models can also be used to order information into a hierarchy. For example, albums and songs are both pieces of creative work and therefore fall under the overarching concept Creative Work. Creative Work could also include things such as Book, Movie or Sculpture.

Most of the relationships in the model in Fig. 1 are connected by a solid arrow, except for "is a", which has a dashed arrow. This is because "is a" represents a special type of relationship, namely: RDF:type. Don't sweat this for the moment; this will be explained in Chapter 2 when we learn about Resource Description Framework (RDF).



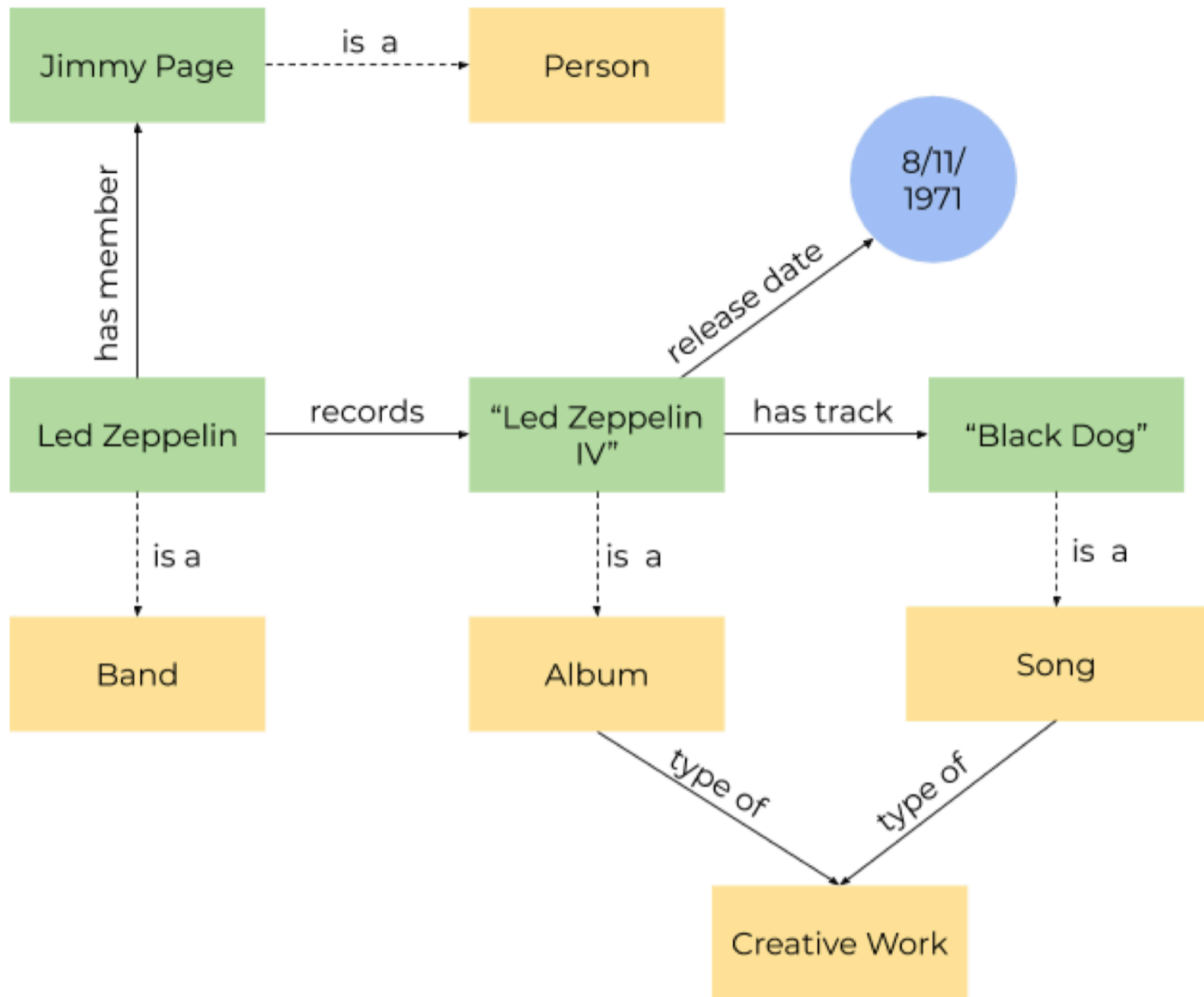


Fig. 1: A diagram showing the band Led Zeppelin and one of their albums "Led Zeppelin IV", released 8th November 1971. One track on the album is "Black Dog". Concepts are marked by rectangles and circles, and the predicates in between are denoted with arrows. Dashed arrows are reserved for the "a" relationship, short for *RDF:type*.



CHAPTER 2: SEMANTIC MODEL BASICS

Now, while I admit it's fun to make pretty diagrams, it's not very useful if computers aren't able to interpret them. So how can a computer extract any meaning from such a Led Zeppelin diagram without serious text-mining efforts? For this, let me I will introduce the first "tool" of this guide: RDF³.

Understanding RDF

The [Resource Description Framework](#)⁴ is, much as the name suggests, a framework for describing data. In an RDF data model you make statements about data in the form of **triples**. Such triples are composed of a **subject** and **object**, with a **predicate** linking them. An **RDF graph** is a collection of these triples, where the graph's objects and subjects form **nodes**.

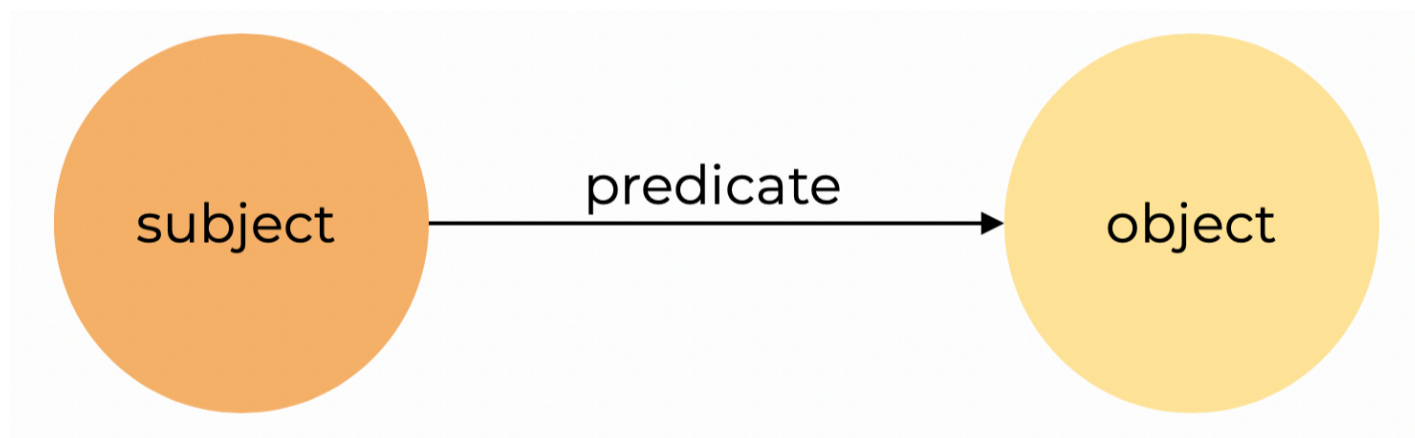


Fig. 2: An RDF triple.

The direction of the predicate is important! Let's take a simple (albeit nerdy) example to understand this:

³ Not all semantic models need to be developed using RDF, but this guide will focus on RDF as the primary modelling resource.

⁴ "Resource Description Framework (RDF): Concepts and Abstract Syntax." Edited by Graham Klyne et al., *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C, 10 Feb. 2004, www.w3.org/TR/rdf-concepts/.



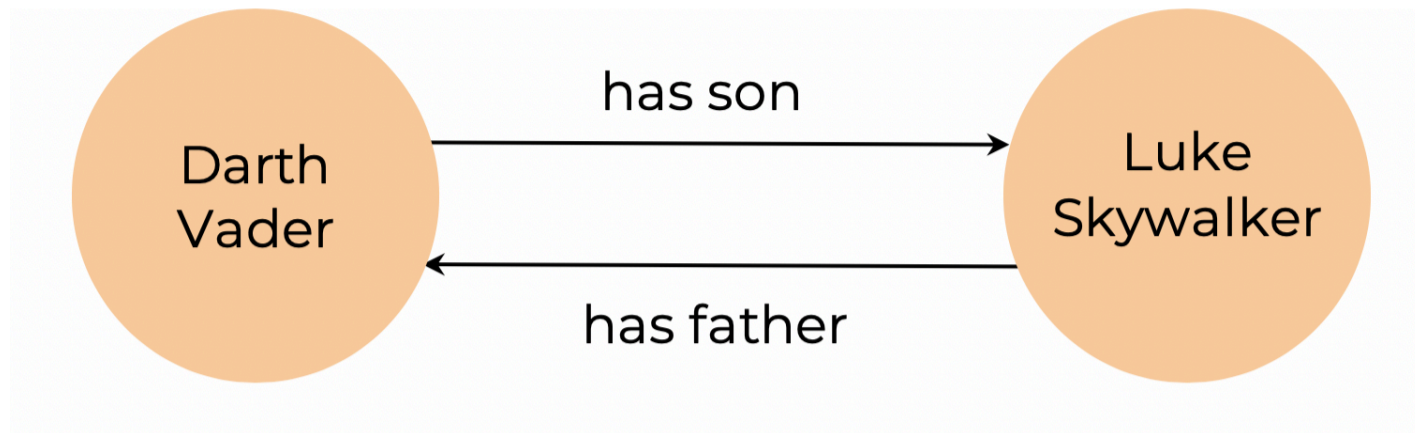


Fig. 3: Directionality matters in RDF triples. Defining reverse relationships is possible, but not always necessary.

We will see later, when we cover the Web Ontology Language (OWL), that there is a simple way to define the “opposite” property using the `inverseOf` relationship. Although it’s not strictly necessary to define reverse properties.

There are 3 different kinds of **nodes** that can exist in your RDF graph:

- **Resources:** A concept that you want to describe (e.g. the rectangles in Fig. 1). All resources must have a *unique identifier*.
- **Literals:** These are values such as strings, numbers and dates (e.g. the circle in Fig. 1)
- **Blank nodes:** A blank node is a resource without a unique identifier. More on this in Chapter 2.

Almost everything needs a URI

All resources and predicates need to have a machine-readable **unique identifier**. Why does uniqueness matter? Well, if, for example, we are making a graph of employees of a large organisation there might be several people with the same name. Aside from giving everyone a nickname, how do we ensure that James White in Accounting is not confused with James White from the Tax Office or James White who delivers the mail? The answer is simple: use Uniform Resource Identifiers (URIs)!



“A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource. To guarantee uniformity, all URIs follow a predefined set of syntax rules.”

URIs are kind of similar to URLs in that they often follow the familiar *https* scheme. URIs can either be **hierarchical** or **opaque**. Without resolving it, an opaque URI will not encode any information; the suffix is usually a randomly generated string of characters. Hierarchical URIs instead contain some level of information. They could encode the location of a resource, i.e. where it exists in the hierarchy of the model or share information regarding the context of a resource. For example, these are two URIs describing the same resource:

- **Hierarchical:** `<http://mycompany.com/people/JediDepartment/LukeSkywalker>`
- **Opaque:** `<http://mycompany.com/AE04801>`.

The hierarchical URI is human readable. From this example we can infer that Luke Skywalker works for My Company in the Jedi Department. The opaque URI also refers to Luke Skywalker but now I am not able to infer that information. Opaque URIs are great to ensure privacy. They also require less upkeep, because if something changes about the resource, say Luke Skywalker were to change to the Retired Jedi Department, then there is no need to update the opaque URI. Even though hierarchical URIs do require updating, they are more human-friendly and thus may be easier to use.

What constitutes a good URI? Properties of sound URIs can be found on the European Bioinformatics Institute ([EBI's page on good practice for URIs](#)⁶). In short they are:

- **Globally unique:** One URI should never refer to two different concepts at the same time, even ones that may seem equivalent.
- **Persistent:** A URI should continue to resolve for the foreseeable future. The URI should survive website re-engineering exercises, for example.

⁵ https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

⁶ Embl-Ebi. “Good Practice for URIs.” *EBI*, European Bioinformatics Institute, www.ebi.ac.uk/rdf/documentation/good_practice_uri/.



- **Stable:** A URI should never be re-used for different things between data releases, even if the original is deleted.
- **Resolvable** (dereferenceable): This simply means: when a user clicks on a URI in their browser, we want them to be redirected to a suitable document.

Let's revisit our beloved rock band from before and assign some URIs. The easiest is to begin with one statement and "translate" it to URIs.

<Led Zeppelin> <has member> <Jimmy Page>.

Before, for Luke Skywalker, I made up some nonsense URIs. If you actually tried resolving the links you would see they both result in a '404 Page Not Found'. However, the last step in the guidelines for good URIs states they should be resolvable and redirect to a document with information relating to that resource. It turns out that for a lot of concepts this already exists. A great example is [Wikidata](https://www.wikidata.org/wiki/Wikidata:Main_Page)⁷, a knowledge base of structured computer and human readable data. Using Wikidata we assign a URI to all three parts of this triple (subject, predicate, object)

<<https://www.wikidata.org/wiki/Q2331>> <<https://www.wikidata.org/wiki/Property:P527>>
<<https://www.wikidata.org/wiki/Q165467>>

which we can visualize as:

⁷ https://www.wikidata.org/wiki/Wikidata:Main_Page



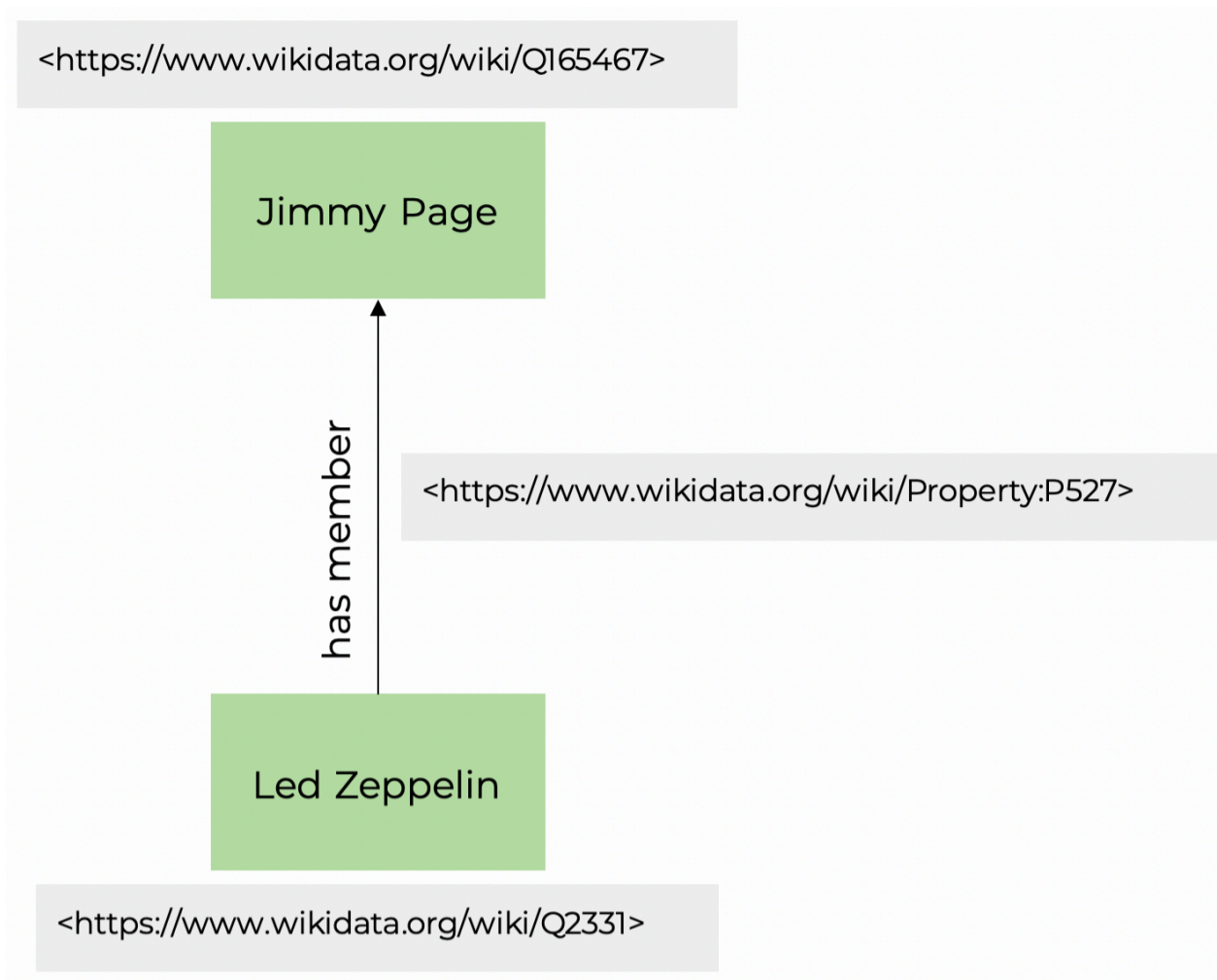



Fig 4: Assigning URIs to a triple from the diagram in Fig. 1. Note that the property 'has member' in Wikidata is actually 'has part'.

If you resolve any of these URIs you get redirected to a page with information on the resource.





WIKIDATA

Main page
Community portal
Project chat
Create a new Item
Create a new Lexeme
Recent changes
Random Item
Query Service
Nearby
Help
Donate

Print/export
Create a book
Download as PDF
Printable version

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Concept URI
Cite this page

Item [Discussion](#)

Led Zeppelin (Q2331)

English rock band [edit](#)
 Led Zep I get the Led out

▼ In more languages
 Configure

Language	Label	Description	Also known as
English	Led Zeppelin	English rock band	Led Zep get the Led out
Dutch	Led Zeppelin	engelse rock groep	
German	Led Zeppelin	englische Rockband	
French	Led Zeppelin	groupe de musique britannique	

All entered languages

Statements


instance of [edit](#)

rock band

▼ 0 references

+ add reference
+ add value

logo image [edit](#)




Led Zeppelin logo.svg
465 x 160; 5 KB

▼ 0 references

+ add reference
+ add value

image [edit](#)



Led Zeppelin 2007.jpg
808 x 582; 225 KB

▼ 0 references

+ add reference
+ add value

inception [edit](#)

1968

▼ 0 references

+ add reference
+ add value

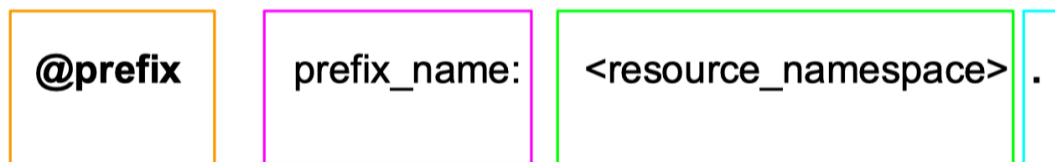
Fig. 5: Wikidata page on Led Zeppelin⁸. Statements are the predicates, with the object of the triples on the right-hand side.

⁸ <https://www.wikidata.org/wiki/Q2331>



This process of assigning a URI to a resource should be applied to the entire graph. However, writing out full URIs quickly becomes tiresome. To alleviate this we use **prefixes**. We define a prefix in the following manner:

- Define a prefix by starting the triple with '@prefix'
- Choose a name for the prefix
- State the namespace of the prefixed resource
- Always end with a period



For example for the Wikidata namespace this would read:

@prefix wd: <<https://www.wikidata.org/wiki/>> .

Now, when we reference a Wikidata URI we simply use "wd:". Revisiting Fig. 4 with prefixes yields:



```
@prefix wd: <https://www.wikidata.org/wiki/> .
@prefix wdp: <https://www.wikidata.org/wiki/Property:> .
```

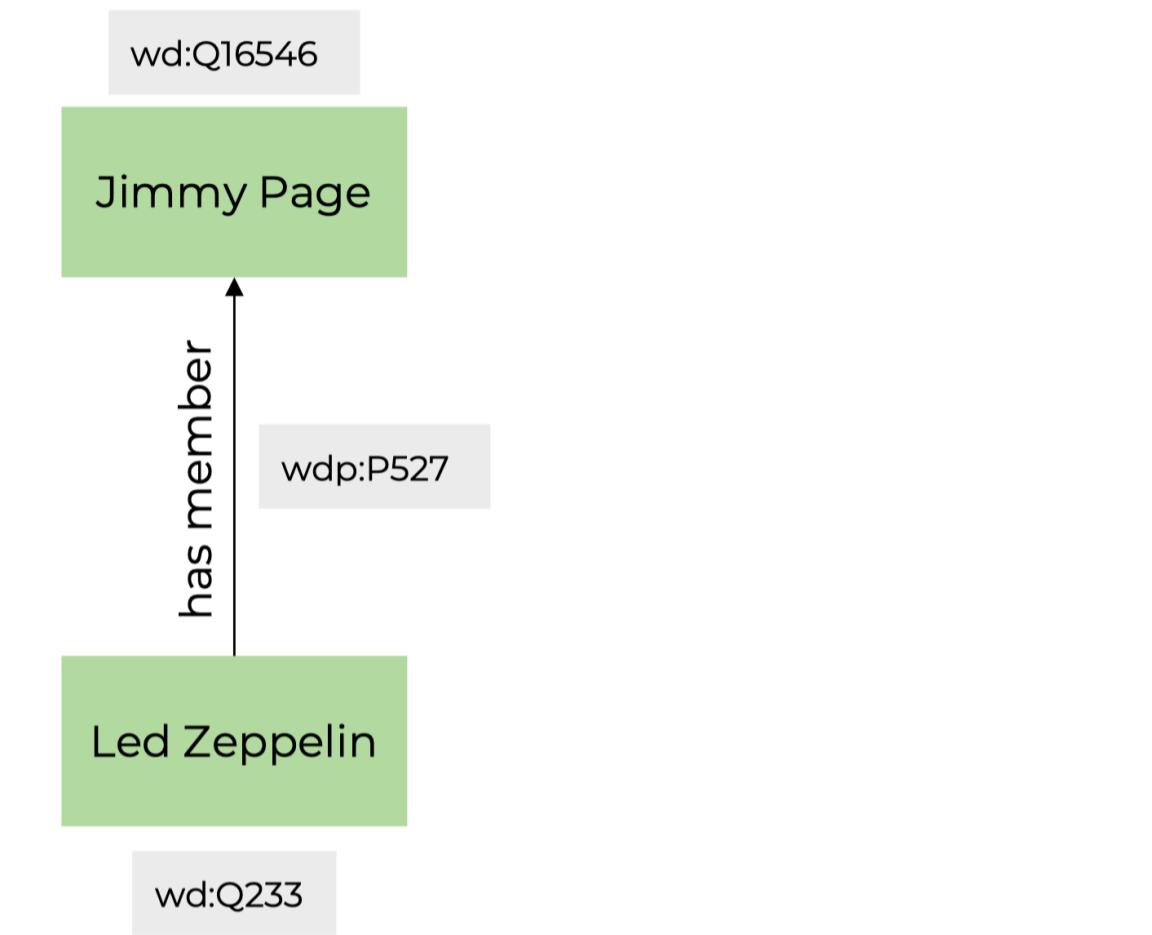


Fig. 6: Using prefixes to represent the same data as in Fig. 4. Prefixes are stated at the top. Wikidata has a slightly different URI for properties and thus has been defined separately as “wdp”.

Now let's do this for the entire graph. In Fig. 7 five prefixes are defined. Most of the URIs originate from Wikidata but there are a few rogue prefixes in there, such as `schema`, `rdf` and `rdfs`. [Schema.org](https://schema.org)⁹ is used for the relationship 'records', which in Schema is actually called 'album'. Why is Wikidata not used here? Strangely enough Wikidata doesn't have a property linking an artist with an album, but rather uses a reverse property called 'performer' linking an album with an artist. What about RDF and RDFS? These are basic RDF vocabularies and are commonly used in RDF graphs.

⁹ <https://schema.org>



```

@prefix wd: <https://www.wikidata.org/wiki/> .
@prefix wdp: <https://www.wikidata.org/wiki/Property:> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <https://schema.org/> .
    
```

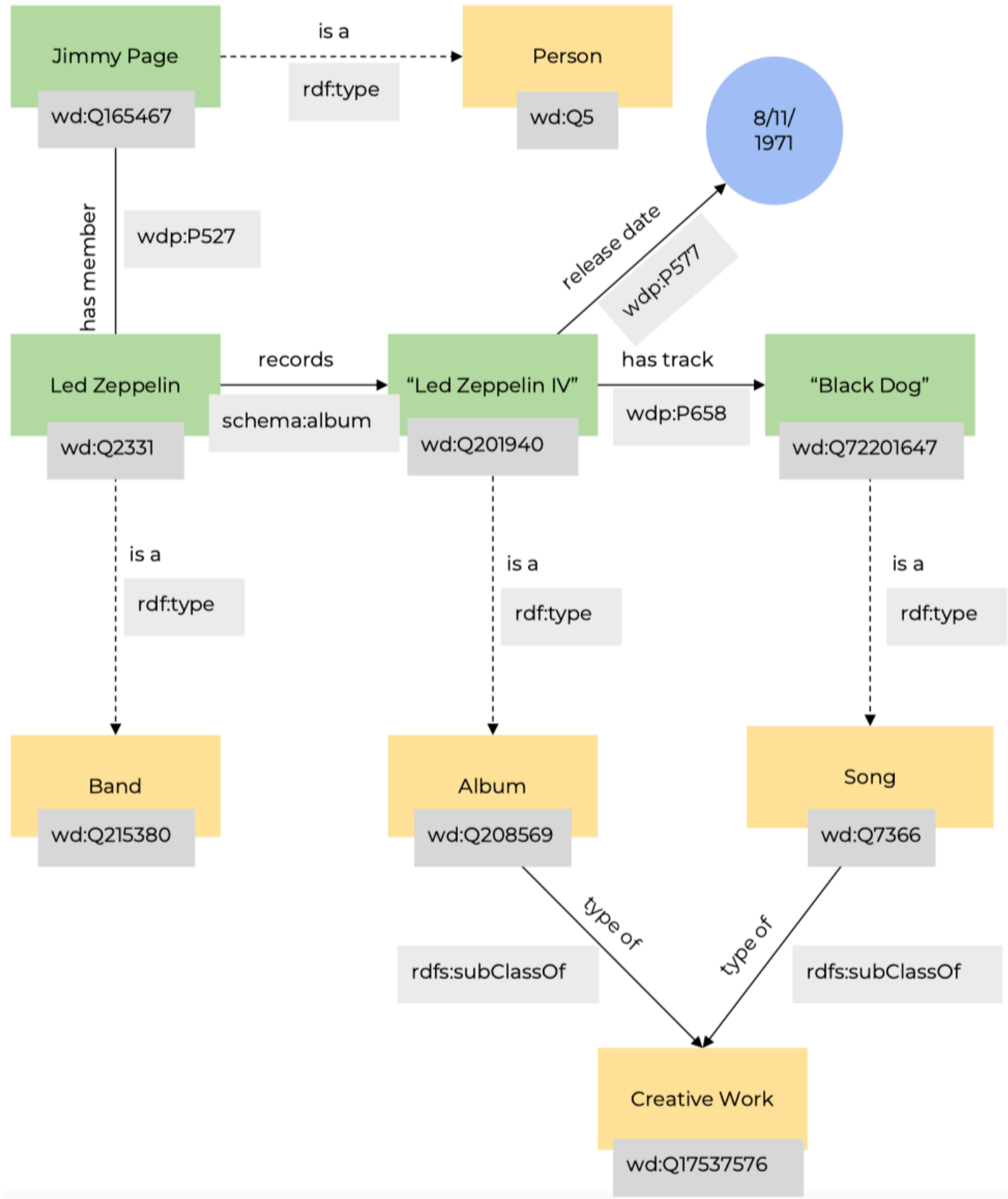


Fig. 7: Using URIs to represent the whole of Fig. 1.



A note about blank nodes (because not everything needs a URI):

Blank nodes are resources which don't have a URI or literal. This sounds illegal but it's actually allowed within RDF. It's not that the URI is unknown but rather that the resource is anonymous. Blank nodes can be used when you don't know much about the resource (which may pose a challenge when creating a URI), when you want to represent complex features of a resource without naming it (think of an address with street, number, city, etc.), or when you want to protect someone's privacy¹⁰.

RDFS

RDFS stands for RDF Schema, you can read more about it [here](#)¹¹. Essentially, RDFS is a data-modelling vocabulary, which is used to model data expressed in RDF. RDFS includes a range of properties and other mechanisms for describing groups of related resources and the relationships between these resources. But first, let's introduce the idea of a class:

- Resources may be divided into groups called **classes**. The members of a class are known as **instances** of the class. Classes and instances are identified by URIs and may be described using RDF properties. The [rdf:type](#)⁵ property may be used to state that a resource is an instance of a class. Classes can also have **subclasses**. All instances of a subclass are also instances of the class.
- **Properties** are relationships between resources, linking subjects to objects.

We can use RDFS to define classes. For example:

- **rdfs:Class** - used to define a resource as a class. For example, in Fig. 7 everything is an rdfs:Class, except for the release date.
- **rdfs:Literal** - used to define a resource as a literal value, i.e. strings or integers. For example, the release date. Note that rdfs:Literal is a class and as an instance of the more generic rdfs:Class.

¹⁰ "Blank Node." *Wikipedia*, Wikimedia Foundation, 1 Jan. 2020, en.wikipedia.org/wiki/Blank_node.

¹¹ "RDF Schema 1.1." Edited by Dan Brickley et al., *RDF Schema 1.1*, W3C, 25 Feb. 2014, www.w3.org/TR/rdf-schema/.



When we go back to the example of our rock band, we see that the colour scheme in Figures 1 and 7 has a purpose: coloured rectangles represent a **rdfs:Class** and a blue circle for **rdfs:Literal**. Instances are represented with green rectangles, and their classes are yellow rectangles. For example, Led Zeppelin is an instance of a Band, and Jimmy Page is an instance of a Person.

RDFS also allows us to define properties, which are a **rdf:Property**. We can define sub-properties of `rdf:Property` using **rdfs:subPropertyOf**. There is a special and often used property, **rdf:type**, which is used to state that a resource is an instance of a class. This property is often abbreviated to **a**.

Using **rdfs:domain** and **rdfs:range** we specify the subject and object of a property, respectively. For example:

```
<hasMember> a rdf:Property .
```

```
<hasMember> rdfs:domain <Band> .
```

```
<hasMember> rdfs:range <Person> .
```

This set of triples states that `<hasMember>` is a property that has `Band` as its subject and `Person` as its object. Again, we make use of prefixes (`rdf` and `rdfs`) in these triples.

Properties can also be used to establish a hierarchy. Namely, classes can be subclasses of other classes, and the property **rdfs:subClassOf** is used for this. Instances of a subclass are also instances of the class. For example, an instance of `Album` is also an instance of `Creative Work`, because `Album` is a subclass of `Creative Work`.

Modelling data in semantic models is not always straightforward. For example, if we want to add genre information to our model we could break down `Album` into subclasses, such as `ClassicalAlbum`, `RockAlbum`, `PopAlbum`, etc. – but we could also impart the genre through a ‘genre’ property with `album` as the subject, or we could make `ClassicalAlbum` an instance of `Album`, rather than a subclass. Confused yet? In truth, any of these options would be correct. What matters is developing a model that satisfies your use cases.

RDFS makes data machine-readable but we still want it to be human-readable. For this there are two properties in RDFS: **rdfs:label** and **rdfs:comment**. `Rdfs:label` is used to give



a human-readable name to the resource, which is especially useful for opaque URIs. `Rdfs:comment` adds a description to a resource. The object of both of these properties is a string. E.g:

```
<hasMember> rdfs:label "has member" .
```

```
<hasMember> rdfs:comment "Property relating a band to one of its band members" .
```

Adding comments to classes and properties is important! It's just like commenting on your code: it seems redundant in the moment but will prove super useful some while down the line when you, or someone else, needs to interpret the cryptic RDF you wrote.

OWL

RDFS is a great way to structure data. As we saw in the last section, with RDFS we can define what type our resource is (class or literal), if it's an instance of another resource, if it has any subclasses or if it is a subclass itself. But there is more to data than structure. There is also *meaning*. And now it turns out this guide to RDF doubles as a guide to life.

Just like RDFS gives structure, OWL gives meaning. Unfortunately, OWL has absolutely nothing to do with actual owls. But I will provide a photo of an adorable baby owl because it seemed somewhat relevant and mostly I wanted to.



Fig. 8: An owl. Of the non-OWL variety. © Megan Lorenz.



[Web Ontology Language](#)¹² (or rearranged into its cute acronym OWL) is a vocabulary extension of RDF, and thus is also expressed in triples. It's mainly used as markup for ontologies, which for all intents and purposes are very similar to the semantic models outlined in this guide.

OWL, like RDFS, also defines classes (namely **owl:Class**), properties and instances. But in contrast to RDFS, OWL is a far richer and stricter vocabulary. You may be asking, if OWL covers most of RDFS *and* is a far more descriptive vocabulary, why even bother with RDFS at all? Well, sometimes using OWL is overkill. Like do you really need to take that 12-week in-person French course for your upcoming holiday in Paris or would it just suffice to do some Duolingo? OWL is a stricter vocabulary and if what you're building is small, then RDFS may be sufficient.

The added bonus of OWL is that it supports and integrates elements of RDF. For example, in OWL you can still use properties such as `rdf:type`, `rdfs:range` and `rdfs:subPropertyOf`.

In addition, OWL defines a whole host of its own properties. These properties are much more detailed than the generic RDFS ones. Some of these properties can be used to impose constraints, such as **owl:allValuesFrom**, which define an allowed class or data range.

```
<hasParent> owl:allValuesFrom <Human> .
```

OWL also supports describing data in terms of set operations. For example, **owl:unionOf** is used to indicate that the class, for example Fruit, includes the individuals from both Sweet Fruit and Non-Sweet Fruit. The subject of an `owl:unionOf` triple can also be a blank node.

```
<Fruit> owl:unionOf ( <SweetFruit> <NonSweetFruit> ) .
```

We can also use OWL to define inverse relationships. Remember the Darth Vader and Luke example? Using **owl:inverseOf** it's possible to state that `hasSon` is the inverse of `hasFather`, i.e.

¹² Bechhofer, Sean, et al. "OWL Web Ontology Language Reference." *OWL Web Ontology Language Reference*, W3C, 10 Feb. 2004, www.w3.org/TR/owl-ref/.



```
<hasSon> owl:inverseOf <hasFather> .
```

We will not explore OWL in full. However, there are two more noteworthy properties which relate to equivalency. The first property, **owl:sameAs**, can be used to state that two *individuals* are the same. For example,

```
<BillClinton> owl:sameAs <WilliamJeffersonClinton> .
```

The second, **owl:equivalentClass**, states the equivalence of two *classes*. E.g.,

```
<USPresident> owl:equivalentClass <PrincipalResidentOfWhiteHouse> .
```

Saying two things are equivalent may seem a bit redundant but it's actually a key advantage of OWL. Using these statements you can now easily reference external models or ontologies. For example, you can state that the Al Pacino on Wikidata is the same as the Al Pacino on IMDB (not that that is the most important thing you could possibly be doing with your OWL time).

Another important thing to note that in OWL it's possible to define two different types of properties:

- *Object properties* link individuals to individuals (e.g. <hasMember>)
- *Datatype properties* link individuals to data values (e.g. <releaseDate>)

Do not reinvent the ontology wheel

In the first section of this chapter we touched upon ways to define URIs for our resources using existing knowledge bases, like Wikidata, and existing vocabularies, like schema.org. In actuality, there are hundreds, if not thousands, of knowledge representations out there. In many cases, experts have taken the time to identify the major concepts in their fields, define them, and then qualify the relationships between them. These knowledge representations are often known as *ontologies*.



What is an ontology?

An ontology is a number of things. To best understand an ontology as a whole let's start with looking at some of its parts:

- A **dictionary**, which is a collection of terms and their definitions (just like a trusty Merriam Webster).
- A **taxonomy**, which is used to create hierarchy (just like a classification of the animal kingdom).
- A **thesaurus**, for describing some basic relationships between terms, mostly by just specifying that two terms are related. For example, *Actor is related to Academy Award*.

And now, finally, there is the **ontology**. An ontology provides all that a dictionary, taxonomy and thesaurus do but the relationships between concepts are better qualified. For example, *Actor wins Academy Award*.

For the purpose of this guide, ontologies are essentially equivalent to semantic models and you can think of them interchangeably.

Reusing existing ontologies

From pharmaceuticals to movies, there is an ontology for (almost) everything. A significant amount of expert time and effort has gone into developing subject-specific ontologies and it's to everyone's benefit to reuse them. There are various tools that can be used to explore existing ontologies.

- [BioPortal](#): for biology-related ontologies
- [Linked Open Vocabularies](#) (LOV): for more general ontologies

And here is a non-exhaustive list of vocabularies/ontologies/knowledge bases that I have found useful in my experience:

- [DCAT](#): to describe datasets and data catalogs
- [PROV-O](#): for provenance information



- [Wikidata](#): for general information, although it includes a good deal of scientific information
- [Schema.org](#): for general information (check bioschemas for biological information)
- [SKOS](#): for knowledge representation
- [Dublin Core Metadata Initiative \(DCMI\)](#): for metadata
- [Friend of a Friend \(FOAF\)](#): to describe people and organizations, and relations between them
- [XSD](#): to describe data type values
- [BioAssay Ontology \(BAO\)](#): to describe biological assays
- [EDAM](#): comprehensive ontology for various topics in bioinformatics

Make sure that the ontology you choose has been updated at least in the last year, ideally in the last 6 months. This prevents you from using obsolete ontologies.



CHAPTER 3: BUILDING A SEMANTIC MODEL

Enough theory! It's time to get our hands dirty. This chapter is dedicated to building a semantic model from scratch. Before we delve into the five steps of modelling, I want to introduce two topics that are going to be very relevant while we model. First: we need to discuss *how* we're going to write the RDF, i.e. what syntax are we going to settle on. And second: let's look at some tools to help you in your model-building mission.

RDF syntax

RDF triples can be expressed in more than one way. How you write the triples depends on the syntax used. The most common syntaxes are: N-Triples, Turtle, JSON-LD and RDF/XML. This guide has been written, and will be continued to be written, in Turtle. Turtle is abbreviated to [TTL](#)¹³, which ironically also stands for Time To Live, and yes, I kid you not: we are yet again on the topic of animals and life. Who knew this guide would be so insightful?

Turtle

Let's take some time to delve into the syntax of TTL. We've already taken a brief look at the use of prefixes in triples. We can use prefixes for any resource, be it a relationship, class or instance. Prefixes just allow us to specify the namespace of that resource without having to write it out every single time.

To explain the ins and outs of TTL I will be borrowing an example from the [TTL](#) resource on w3. This time, I wrote the triples in my text editor of choice: Visual Studio Code. Follow the list of steps below to see how classes and relationships are created to describe two resources, Green Goblin and Spiderman:

```

1  @base <http://example.org/> .
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
5  @prefix rel: <http://www.perceive.net/schemas/relationship/> .

```

¹³ Beckett, David, et al. "RDF 1.1 Turtle." RDF 1.1 Turtle, W3C, 25 Feb. 2014, www.w3.org/TR/turtle/.



- Lines 1-5: Specify prefixes. The start to any good TTL file is defining prefixes. In lines 2-5 we see some of our usual suspects like `rdf`, `rdfs` and `foaf`, and a new vocabulary 'rel'. There is also a special kind of prefix declared in line 1: **@base**. The base prefix serves as the base URI and any term within angled brackets '`<`' and '`>`' belongs to that namespace. Note that each line must be terminated with a whitespace and a period `'.'`.

```
7 <#green-goblin> rel:enemyOf <#spiderman> .
8 <#green-goblin> a foaf:Person . # in the context of the Marvel universe
9 <#green-goblin> foaf:name "Green Goblin" .
```

- Lines 7-9: Describe resource. We begin with Green Goblin. This is a resource in our base namespace so we enclose it with angled brackets. Green Goblin's URI in full is `<http://example.org/#green-goblin>`. The following lines give some information on Green Goblin, i.e. Green Goblin is an enemy of Spiderman, Green Goblin is a type of Person, and Green Goblin's name is - wait for it - Green Goblin. Strings are enclosed with double quotes. Comments are added with the hash sign '#'. Very important to note is that **every triple must end with a whitespace and a period.**

```
7 <#green-goblin> rel:enemyOf <#spiderman> ;
8 a foaf:Person ; # in the context of the Marvel universe
9 foaf:name "Green Goblin" .
```

- It's a bit repetitive to write `<#green-goblin>` for every triple, and this can get clunky and time-consuming. The way around this is **predicate lists**. In predicate lists it's possible to group together statements on the same subject. Predicate-object statements are separated by a semicolon `';`'. Predicate lists must end with a whitespace and period.

```
11 <#spiderman>
12 rel:enemyOf <#green-goblin> ;
13 a foaf:Person ;
14 foaf:name "Spiderman", "Человек-паук"@ru .
```

- Lines 11-14: Describe resource. We go on to describe Spiderman, which is also a resource in the base namespace. In line 14 there are two objects to the `foaf:name` predicate. One is the English name "Spiderman", and the second is the Russian name. Quoted literals are followed by a language tag, datatype IRI, or neither. Language tags are preceded with a '@' and then the language key.



```

1 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
2 @prefix show: <http://example.org/vocab/show/> .
3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
4
5 show:218 rdfs:label "That Seventies Show"^^xsd:string . # literal with XML Schema string datatype
6 show:218 rdfs:label "That Seventies Show"^^<http://www.w3.org/2001/XMLSchema#string> . # same as above
7 show:218 rdfs:label "That Seventies Show" . # same again
8 show:218 show:localName "That Seventies Show"@en . # literal with a language tag

```

In this further example on quoted literals, we use [xsd](#) to specify that “That Seventies Show” is a string. Line 5 uses the xsd prefix to do this, line 6 uses the full URI of xsd and line 7 has no specification. All these lines are equivalent. Line 8 shows the use of a language tag. We can use xsd also to specify other data types like time, date and integer.

```

1 @base <http://example.org/> .
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
5 @prefix rel: <http://www.perceive.net/schemas/relationship/> .
6
7 <#green-goblin>
8   rel:enemyOf <#spiderman> ;
9   a foaf:Person ; # in the context of the Marvel universe
10  foaf:name "Green Goblin" .
11
12 <#spiderman>
13   rel:enemyOf <#green-goblin> ;
14   a foaf:Person ;
15   foaf:name "Spiderman", "Человек-паук"@ru .

```

This is the full code for the Green Goblin and Spiderman example.

If you have written your RDF in one format and realise, for some reason, it needs to be in another - don't fret! A multitude of RDF converters are here to help. Try this [one](#)¹⁴.

¹⁴ <http://www.easyrdf.org/converter>



Semantic models and ontology builders

In my opinion, there are two ways in which you can build a semantic model:

1. Write out triples in a text editor (either manually or automatically)
2. Use a tool that creates the triples for you

Personally, I prefer the former. Writing out triples gives me full control over the model and – more importantly – I don't waste time fighting with a tool's UI. I admit that this may be my own problem, so if there's a tool you like and find it works for you then by all means forgo the text editor. Writing triples in a text editor does not necessarily mean you have to write them all out by hand (although I am not above doing this); you can automate triple generation with the handy [Python RDF](#) package¹⁵.

With regards to tools, the market is actually rather saturated. However, many of these are not open-source and can be quite pricey. [Protégé](#)¹⁶ is a popular open-source tool that gets the job done.

Protégé

Protégé is an ontology editor and framework builder developed by the Stanford Center for Biomedical Informatics Research. There is both a web version and a desktop version.

Essentially, Protégé allows the user to add classes, object and datatype properties, and individuals without having to write out triples. Classes can easily be subclassed in order to create a hierarchy. Once you have created your ontology you can download it as an OWL file, if needed with all the triples written out. [This](#) article¹⁷ is a great way to learn more about Protégé and see it in action.

¹⁵ <https://rdflib.readthedocs.io/en/stable/>

¹⁶ <https://protege.stanford.edu>

¹⁷ Jayawardana, Vindula. "Ontology Generation and Visualization with Protégé." *Medium*, Medium, 5 Dec. 2017, medium.com/%40vindulajayawardana/ontology-generation-and-visualization-with-prot%C3%A9g%C3%A9-6df0af9955e0.



Building the model - a step-by-step guide

Now we've learned about RDF, classes and properties, syntaxes and tools. All of this is fine and dandy but what entities do we actually include in our model? It's easy to realise that there is no limit to describing knowledge. You could infinitely branch out in different directions, like an ever-expanding web. For example, I make a model that sets out describing movies, which feature actors, who are people, and people are homo sapiens, and homo sapiens discovered fire and fire is the rapid oxidation of a material in the exothermic chemical process of combustion and now we are describing the entire universe instead of just making a model that tells me in which movies I can see Brad Pitt naked - I mean, in which movies I can see Brad Pitt act.

Our friends at Stanford not only developed Protégé but also published a very handy Ontology Development 101 [guide](#)¹⁸ detailing how to build an ontology. In the guide they outline 3 rules of ontology design. The first rule of ontology design is that you don't talk about ontology design¹⁹. Just kidding. Here are the rules:

*There is **no one correct way** to model a domain— there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.*

*Ontology development is necessarily an **iterative process**.*

*Concepts in the ontology should be **close to objects** (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.*

18

¹⁸ Noy, Natalya F, and Deborah L McGuinness. "Ontology Development 101: A Guide to Creating Your First Ontology." <https://Protege.stanford.edu/>, Stanford University, protege.stanford.edu/publications/ontology_development/ontology101.pdf.

¹⁹ That was the last Brad Pitt reference, I promise.



The first rule is really about deciding what questions your model or ontology is going to answer. Homing in on the use cases determines which classes and relationships need to be included and in how much detail. As you start to develop and test your model, you will see that revisions are inevitable, and often the final product is very different from the first iteration. Hence, the second rule. Since your model is a model of reality, even if the reality is data and systems, the entities in the model should reflect this reality as closely as possible. There is no need to get unnecessarily abstract and philosophical if it is of no use to the end-user.

Let's go over some basic steps to get you started with your model. Again, these are taken from the Ontology 101 guide.

Step 1: Determine the domain and scope (a.k.a use cases)

Let me harp on a little longer about use cases. Use cases are the driving force behind model development. Without use cases, model development is a bit of a stab in the dark and stabbing in the dark is always a bad idea. Useful questions to help you gather use cases could be:

- What is the domain that the model should cover? (E.g. clinical trials, drug discovery, genomics sequencing pipelines)
- What are the questions that the model needs to answer?
- What is the model going to be used for?
- Who is going to use it?
- Who is going to maintain the model? (E.g. data steward, data curation team)
- Where should the model live?

The second question, what questions the model needs to answer, is a rather important one. These are considered (by the authors of Ontology 101) as the **competency questions** and serve as the litmus test for the model.



Say that you are building a model for a pharmaceutical client who is interested in modelling their clinical trial processes. Some competency questions you could expect for such a model are:

- Is clinical trial AAAB001 interventional or observational?
- Can a patient over the age of 65 be admitted to trial AAAB001?
- Can I see all the genomics data we have for this patient?
- What is the protocol for trials with drug AAAB?
- Where are the clinical trials located that enrol the highest number of patients?

Based on such questions, the model will include information on clinical trials, such as the type of trial, inclusion criteria, geographical location, and protocol, as well as patient information and enrolment numbers. Additionally, patients need to be linked to a trial.

Step 2: Determine important concepts in the model

Based on the questions in step 1 it will become clear which concepts need to be included. Some of these concepts will be classes and others might be properties or instances. From the example above, some concepts could be *clinical trial*, *patient*, *data*, *drug*, *location*, *patient count*.

Step 3: Reuse ontologies

We covered this topic in Chapter 2 but just to reiterate: where possible, include existing ontologies. Search for ontologies that cover the domain being modelled and include some important concepts from Step 2. You can reference ontologies in your triples by making use of the ontology's namespace or import the entire ontology if using a tool like Protégé. Even if you don't end up using an external ontology, looking at how things are modelled elsewhere can help you get started on your own modelling.

Step 4: Define classes, class hierarchy and properties

This is where it gets fun! Start by defining some central classes, like *clinical trial* and *patient*. Then we can start to attribute some properties to these classes, like *clinical trial code*,



patient ID and *location*. Next we need to decide what type of properties these are. It makes sense that *clinical trial code* and *patient ID* are datatype properties as their objects will be literals. But *location* could be either a datatype or an object property. After all, the name of a place could simply be a string but it could also be a resource with a URI. So which is correct?

The rule of thumb I go by is to make something a class if I want to say something else about it and/or if I want control the instances. Let's break this down. If the location of the clinical trial is the Erasmus University Medical Center in Rotterdam then I could encode this simply as a string, i.e. "Erasmus University Medical Center". But maybe I want to say that the medical center is located in Rotterdam, The Netherlands. I can't do this if it's a string. Furthermore, leaving the location as a literal may leave room for error. In some instances it may be entered as "Erasmus MC" or "EUMC" or have a typo like "Eramus University Medical Center".

Carry on defining the contents of the model until it satisfies the use cases. At this point, test the model against the use cases and iterate if needed.

Step 5: Define constraints

RDF on its own has no constraints. There's nothing telling you what can or cannot be a class, property or individual. And much like an unsupervised teenager home alone for the weekend, you could end up (even accidentally) creating a total and utter mess. What we need is a supervisor, who sets the rules for what can and can't happen - enter SHACL.

SHACL

[SHACL](#)²⁰ (pronounced "shackle") stands for Shapes Constraint Language. SHACL is a language for validating RDF graphs against a set of conditions. The conditions themselves are expressed as RDF graphs, referred to as "shapes graphs". The RDF graph that is being validated is a "data graph".

²⁰ Shapes Constraint Language (SHACL)." Edited by Holger Knublauch and Dimitris Kontokostas, *Shapes Constraint Language (SHACL)*, W3C, 20 July 2017, www.w3.org/TR/shacl/.



Let's take the following data graph²¹:

```

68 @prefix ex: <http://example.com/ns#> .
69 @prefix sh: <http://www.w3.org/ns/shacl#> .
70
71 ex:Alice
72   a ex:Person ;
73   ex:ssn "987-65-432A" .
74
75 ex:Bob
76   a ex:Person ;
77   ex:ssn "123-45-6789" ;
78   ex:ssn "124-35-6789" .
79
80 ex:Calvin
81   a ex:Person ;
82   ex:birthDate "1971-07-07"^^xsd:date ;
83   ex:worksFor ex:UntypedCompany .

```

We want to apply the following conditions to our graph:

- An instance of **ex:Person** can have at most one value for the property **ex:ssn** (as you can see Bob is in violation of this rule by having two SSNs), and this value is a literal with the datatype **xsd:string** that matches a specified regular expression.
- An instance of **ex:Person** can have unlimited values for the property **ex:worksFor**, and these values are URIs and instances of **ex:Company**.
- An instance of **ex:Person** cannot have values for any other property apart from **ex:ssn**, **ex:worksFor** and **rdf:type**.

These conditions can be applied using the following SHACL constraints specified in this shape graph:

²¹ This example is taken from <https://www.w3.org/TR/shacl/#dfn-shacl-instance>




```

68 @prefix ex: <http://example.com/ns#> .
69 @prefix sh: <http://www.w3.org/ns/shacl#> .
70
71 √ ex:PersonShape
72   a sh:NodeShape ;
73   sh:targetClass ex:Person ;      # Applies to all persons
74   √ sh:property [                # _:b1
75     sh:path ex:ssn ;              # constrains the values of ex:ssn
76     sh:maxCount 1 ;
77     sh:datatype xsd:string ;
78     sh:pattern "^\\d{3}-\\d{2}-\\d{4}$" ;
79   ] ;
80   √ sh:property [                # _:b2
81     sh:path ex:worksFor ;
82     sh:class ex:Company ;
83     sh:nodeKind sh:IRI ;
84   ] ;
85   sh:closed true ;
86   sh:ignoredProperties ( rdf:type ) .

```

- We first define a PersonShape class in the shapes graph. This class is what we call a *shape*. The target of this shape is the Person class from the data graph.
- The next line introduces a blank node for the property shape. This constraint applies to the property ex:ssn. The next three lines give the constraints for the property. Namely: Person has maximum one ex:ssn property, which has datatype object of type xsd:string. The pattern of the xsd:string is specified by the regex expression.
- A property shape is introduced for ex:worksFor on line 81. The object of the property is ex:Company, and requires an IRI.
- sh:closed set to 'true' to close the shape.
- sh:ignoredProperties can include a list of properties that are also permitted in addition to those explicitly enumerated via sh:property. Here the only one allowed is rdf:type.

This was a very brief introduction to SHACL but should be enough to get you started on writing your own SHACL validation.



Visualising your semantic model

Regardless of how you have chosen to create your model, in the end you want to see what you have made. My favourite visualisation tool is [WebVOWL](http://vowl.visualdataweb.org/webvowl.html)²².

WebVOWL is a web application for the interactive visualisation of ontologies. It provides graphical depictions for elements that are part of OWL, so in other words WebVOWL works best with models that are rooted in OWL. Other vocabularies are not necessarily recognised. To view your ontology, simply upload the OWL or TTL file (under the 'Ontology' tab). There is also a WebVOWL plugin that can be used for Protégé.

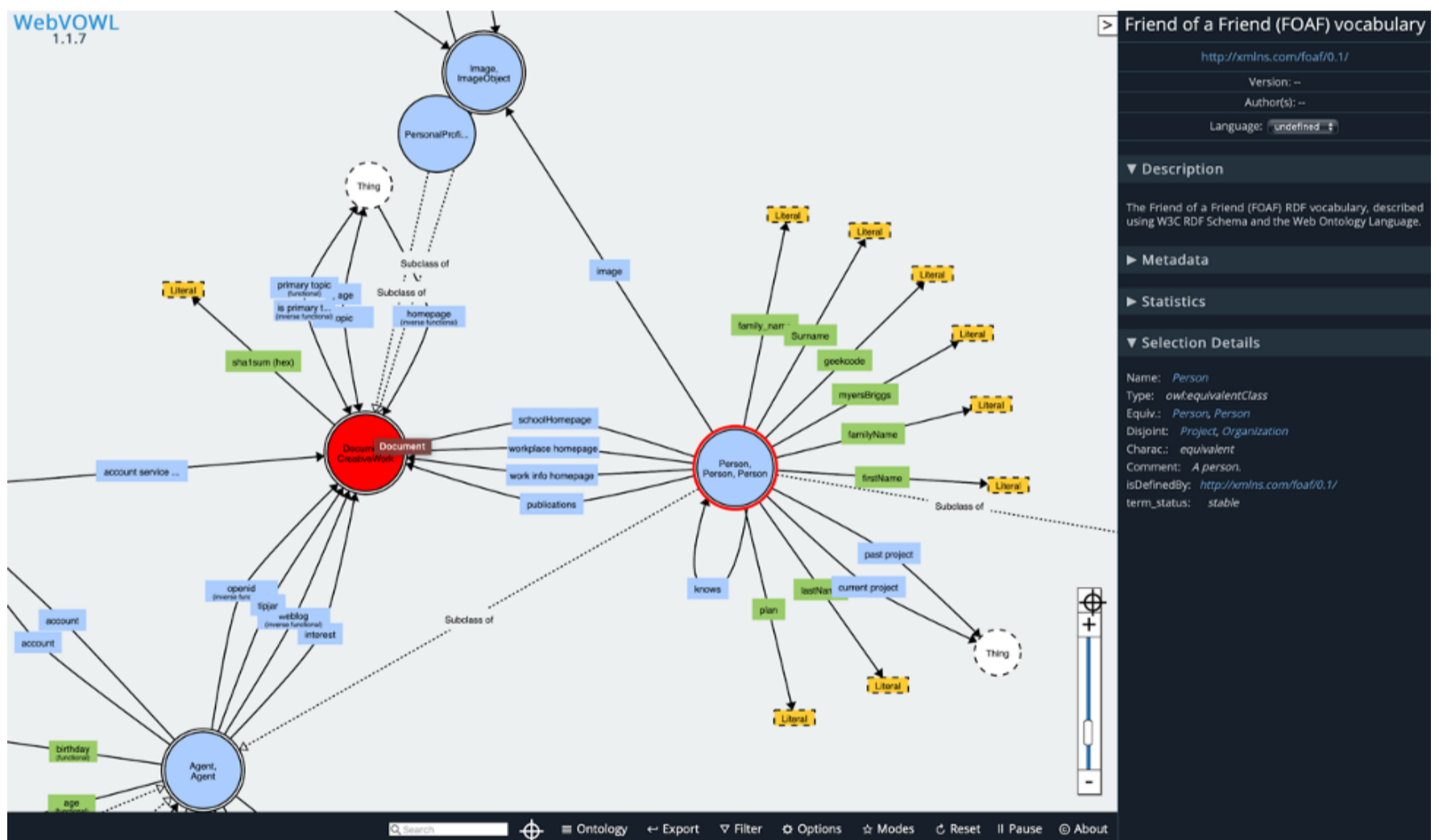


Fig. 9: Excerpt from WebVOWL. This is a visualisation of the FOAF ontology.

WebVOWL makes it easy to interactively explore an ontology. Clicking on resources in the visualisation pulls up some information on the selection. In Figure 9 I have selected the 'Person' class and details on this class can be seen on the righthand side under the tab

²² <http://vowl.visualdataweb.org/webvowl.html>



'Selection Details'. The blue text is actually hyperlinked with the URI of that resource, so clicking on 'Person' redirects me to the FOAF vocabulary page.

Additionally, WebVOWL has numerous customisable options, such as the degree of collapsing (a more collapsed view gives you only the higher classes in the ontology), the distance between classes, and the possibility to pin classes to a particular spot (otherwise they tend to bob around). My go-to options are to untick 'Color externals' and tick 'Compact notation' in the 'Options' tab. The untick is because it makes the graph look nicer, and the tick is to get rid of the 'External' label on classes, which often causes confusion.

It's not great to upload sensitive information to the online version of WebVOWL, it is best to install a local instance on your machine. The instructions for doing this can be found on the WebVOWL main page under 'Installation²³'.

I love WebVOWL. Really, it's great. But I should give a small disclaimer and say that, like everything in life, it's not *perfect*. For instance, do not try and upload ginormous ontologies to webVOWL because this will make it painstakingly slow. Also, if your model is not rooted in OWL you might freak out when you see what you have created, as it will look nothing like what you envisioned. Furthermore, if your model is above a certain threshold of resources it's going to get messy. And lastly, and maybe the most annoying feature, every time you reload an ontology, it redoes the layout. So if you have just memorised the location of all classes and how to navigate between them, be prepared that in the presentation you are just about to give everything will rearrange itself as a result of loading the model on a different machine.

²³ <http://vowl.visualdataweb.org/webvowl.html>



CHAPTER 4: QUERYING A SEMANTIC MODEL

It's not much use making a model if you can't *do* anything with it (aside from admiring it). In Chapter 3 we discussed use cases and the types of questions you ideally want your model to answer. Now we're going to learn how to ask our model these questions.

SPARQL

[SPARQL](#)²⁴ (pronounced "sparkle") is a recursive acronym for SPARQL Protocol and RDF Query Language. SPARQL is to RDF what SQL is to relational databases. And in fact SPARQL is similar to SQL. If you know a bit about SQL then learning SPARQL is easy peasy lemon squeezy. If you don't know any SQL - don't get discouraged! SPARQL isn't all that difficult.

As the name suggests, we can use SPARQL to query RDF. Querying is done by creating statements in the form of triples. The example below shows a query on the data to find all albums for a given band (you guessed it: Led Zeppelin).

Data:

```
1  @prefix schema: <http://schema.org/> .
2  @prefix wd: <http://www.wikidata.org/wiki/> .
3
4  wd:Q2331 schema:album wd:Q201940 .
5  wd:Q2331 schema:album wd:Q209539 .
```

Led Zeppelin (wd:Q2331) has 2 albums, namely wd:Q201940 and Q209539.

²⁴ SPARQL Query Language for RDF." Edited by Eric Prud'hommeaux and Andy Seaborne, *SPARQL Query Language for RDF*, W3C, 15 Jan. 2008, www.w3.org/TR/rdf-sparql-query/



Query:

```

1  @prefix schema: <http://schema.org/> .
2  @prefix wd: <http://www.wikidata.org/wiki/> .
3
4  SELECT ?album_name
5  WHERE {
6  |   wd:Q2331 schema:album ?album_name .
7  }

```

The query consists of two parts: the SELECT clause identifies the variables to appear in the query results (i.e. album_name), and the WHERE clause provides the basic pattern to match against the data. The pattern in this example is a single triple with the variable album_name in the object position²⁵.

The result will be a column called album_name with two entries: wd:Q201940 and wd:Q209539.

We can also use SPARQL to query based on literal matching.

Data:

```

1  @prefix schema: <http://schema.org/> .
2  @prefix wd: <https://www.wikidata.org/wiki/> .
3  @prefix wdp: <https://www.wikidata.org/wiki/Property:> .
4
5  wd:Q201940 a wd:208569 ; #Led Zeppelin IV is a studio album
6  |   wd:P1449 "Led Zeppelin IV" . #has nickname "Led Zeppelin IV"

```

These triples refer to the Led Zeppelin album nicknamed *Led Zeppelin IV*²⁶. If I want to find all albums with the nickname "Led Zeppelin IV", I can do so in the following query:

²⁵ <https://www.w3.org/TR/rdf-sparql-query/>

²⁶ After a somewhat critical response to their previous album, Led Zeppelin decided to release their fourth album untitled. Inside the sleeve there are four symbols representing each band member. The album came to be known as *Led Zeppelin IV* but also went by other names like *Four Symbols*.



Query:

```

1  @prefix schema: <http://schema.org/> .
2  @prefix wd: <https://www.wikidata.org/wiki/> .
3  @prefix wdp: <https://www.wikidata.org/wiki/Property:> .
4
5  SELECT ?album
6  WHERE {
7    |   ?album wd:P1449 "Led Zeppelin IV" .
8  }

```

I use SELECT to identify the variable that will appear in the results (i.e. album) and the WHERE statement encloses the pattern I'm looking to find in the data. The result of the query will be one column named album with one entry, namely wd:Q201940.

These are very basic queries in SPARQL. [This](#) Stardog tutorial is a good guide to get started on more elaborate queries.

Oftentimes knowledge bases and other similar resources will have a SPARQL endpoint which can be used to query the data. Wikidata has such an endpoint called the [Wikidata Query Service](#). Under the 'Examples' tab you can find some example queries, but you can also make some yourself and run them to see what you get. If you hover over any predicate, object or subject a small information box (see Fig. 10) will appear, which is very useful given the opacity of Wikidata's URIs.



The screenshot shows the Wikidata Query Service interface. At the top, there is a navigation bar with the Wikidata logo, the text "Wikidata Query Service", and buttons for "Examples", "Help", and "More tools". Below the navigation bar is a text area containing a SPARQL query. The query is as follows:

```

1 #A network of Drug-disease interactions on infectious diseases (Source: Disease Ontology, NDF-RT and ChEMBL)
2 #added before 2016-10
3 #defaultView:Graph
4 SELECT DISTINCT ?item ?itemLabel ?rgb ?link
5 WHERE
6 {
7   VALUES ?toggle { true false }
8   ?disease wdt:P699 ?doid;
9     wdt:P2
10    wdt:P2
11   ?drug rdfs:labe
12   FILTER(LAI
13   ?disease rdfs:label ?diseaseLabel.

```

A tooltip is visible over the property `wdt:P699` in the query. The tooltip contains the following text:

```

Disease Ontology ID (P699)
identifier in the Disease Ontology
database

```

Fig. 10: Information box for `wdt:P699` showing human readable information for the property, i.e. name and description.



Let's cover some useful SPARQL query forms and modifiers. These are taken from the [W3 page on SPARQL](#)²⁴.

DISTINCT

Eliminates duplicate solutions.

Data:

```

48  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
49
50  _:x foaf:name "Alice" .
51  _:x foaf:mbox <mailto:alice@example.com> .
52
53  _:y foaf:name "Alice" .
54  _:y foaf:mbox <mailto:asmith@example.com> .
55
56  _:z foaf:name "Alice" .
57  _:z foaf:mbox <mailto:alice.smith@example.com> .

```

Query:

```

1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2
3  SELECT DISTINCT ?name
4  WHERE {
5  |   ?x foaf:name ?name
6  }

```

Result:

name
"Alice"



You may be wondering about the odd-looking subjects in these triples. What on earth is “_:x”? It's a blank node! We've discussed blank nodes a bit but they can be confusing. Check out the example below from [this](#)²⁷ Wikipedia entry:

```
60  ▾ _:address a ex:Address ;
61      ex:streetAddress "123 Main St." ;
62      ex:postalCode "A1A1A1" ;
63      ex:addressLocality "London" .
```

Such a blank node is useful in two ways: first of all, we don't have to explicitly name this resource, and secondly, we make sure to protect the privacy of the individual.

OPTIONAL

A graph pattern in a query returns *only* those results that match the pattern. Statements that are OPTIONAL are returned if present but not excluded otherwise.

Query:

```
10  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
11
12  SELECT ?name ?mbox
13  WHERE { ?x foaf:name ?name .
14          OPTIONAL { ?x foaf:mbox ?mbox }
15          }
```

Result:

name	mbox
"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>
"Bob"	

²⁷ "Blank Node." *Wikipedia*, Wikimedia Foundation, 1 Jan. 2020, en.wikipedia.org/wiki/Blank_node.



UNION

UNION concatenates results of two queries in the WHERE statement. Unlike OPTIONAL which returns the results in separate columns, UNION merges the result into a single column.

Data:

```

18 @prefix dc: <http://purl.org/dc/elements/1.1/> .
19 @prefix : <http://example.org/book/> .
20 @prefix ns: <http://example.org/ns#> .
21 @prefix schema: <https://schema.org> .
22
23 :book1 dc:title "SPARQL Tutorial" .
24 :book1 ns:price 42 .
25 :book2 dc:title "The Semantic Web" .
26 :book2 ns:price 23 .
27 :book3 schema:name "A data engineer's guide to semantic models" .
28 :book3 ns:price 0 .

```

Query:

```

31 PREFIX dc: <http://purl.org/dc/elements/1.1/>
32 PREFIX schema: <https://schema.org>
33
34 SELECT ?title
35 WHERE {
36   { ?x dc:title ?title } UNION { ?x schema:name ?title }
37 }

```

Result:

title
"SPARQL Tutorial"
"The Semantic Web"
"A data engineer's guide to semantic models"



FILTER

Restrict solutions to those for which the filter expression evaluates to TRUE. FILTER can be used on strings or on numeric expressions.

Data:

```

18  @prefix dc:    <http://purl.org/dc/elements/1.1/> .
19  @prefix :     <http://example.org/book/> .
20  @prefix ns:   <http://example.org/ns#> .
21
22  :book1  dc:title  "SPARQL Tutorial" .
23  :book1  ns:price  42 .
24  :book2  dc:title  "The Semantic Web" .
25  :book2  ns:price  23 .

```

Query to restrict values of **strings**. Regex can be used to match expressions but this is not necessary. A query with FILTER (?title "SPARQL Tutorial") would yield the same result.

Query:

```

28  PREFIX  dc:    <http://purl.org/dc/elements/1.1/>
29
30  SELECT  ?title
31  WHERE   { ?x  dc:title  ?title
32          |      FILTER  regex(?title, "^SPARQL")
33          }

```

Result:

title
"SPARQL Tutorial"



Query to restrict based on **arithmetic expression**. This query selects only books that have a price lower than 30.5.

Query:

```

36 PREFIX dc: <http://purl.org/dc/elements/1.1/>
37 PREFIX ns: <http://example.org/ns#>
38 SELECT ?title ?price
39 WHERE { ?x ns:price ?price .
40         FILTER (?price < 30.5)
41         ?x dc:title ?title . }
```

Result:

title	price
"The Semantic Web"	23

VALUES

VALUES is a new feature in SPARQL 1.1. It's similar to FILTER in that it's used to restrict results based on a condition.

Query:

```

28 PREFIX dc: <http://purl.org/dc/elements/1.1/>
29
30 SELECT ?title
31 ✓ WHERE { ?x dc:title ?title
32         VALUES ?title {"SPARQL Tutorial"}
33         }
```

Result:

title
"SPARQL Tutorial"



ORDER BY

Establish an order to the result, e.g. descending alphabetical. Results can also be ordered based on numeric values, datetimes or booleans.

Data:

```

18 @prefix dc: <http://purl.org/dc/elements/1.1/> .
19 @prefix : <http://example.org/book/> .
20 @prefix ns: <http://example.org/ns#> .
21 @prefix schema: <https://schema.org> .
22
23 :book1 dc:title "SPARQL Tutorial" .
24 :book1 ns:price 42 .
25 :book2 dc:title "The Semantic Web" .
26 :book2 ns:price 23 .
27 :book3 schema:name "A data engineer's guide to semantic models" .
28 :book3 ns:price 0 .

```

Query:

```

31 PREFIX dc: <http://purl.org/dc/elements/1.1/>
32 PREFIX schema: <https://schema.org>
33
34 SELECT ?title
35 WHERE {
36   { ?x dc:title ?title } UNION { ?x schema:name ?title }
37 }
38 ORDER BY DESC(?title)

```

title
"A data engineer's guide to semantic models"
"SPARQL Tutorial"
"The Semantic Web"



LIMIT

Cap the number of solutions displayed in the results. Useful for large datasets to improve query time.

Query:

```

31 PREFIX dc: <http://purl.org/dc/elements/1.1/>
32 PREFIX schema: <https://schema.org>
33
34 SELECT ?title
35 WHERE {
36     { ?x dc:title ?title } UNION { ?x schema:name ?title }
37 }
38 ORDER BY DESC(?title)
39 LIMIT 2

```

Result:

title
"A data engineer's guide to semantic models"
"SPARQL Tutorial"

GRAPH

Our RDF triples don't need to exist in a single graph. In fact, we can easily have a collection of named graphs on which we want to query. If we want to limit the query to one of these named graphs we can use the GRAPH keyword.

For this example half the data (i.e. "SPARQL Tutorial" and "The Semantic Web") exists on a named graph that has the following URI: <http://examplebooks.org/semanticBooks/>.



Query:

```
31 PREFIX dc: <http://purl.org/dc/elements/1.1/>
32 PREFIX schema: <https://schema.org>
33
34 SELECT ?title
35 WHERE {
36     GRAPH <http://examplebooks.org/semanticBooks/>
37     { ?x dc:title ?title } UNION { ?x schema:name ?title }
38 }
```

Result:

title
"The Semantic Web"
"SPARQL Tutorial"

The best way to familiarise yourself with SPARQL is to try out these different keywords yourself. A good place to start is the Wikidata Query Service.



Beginning your semantic modelling journey

Well, congratulations! That was the 4th and final chapter of this guide. Although the guide may be at its end, your journey into semantic modelling is far from over. Maybe, for some, it has just begun.

Learning the theory is a great place to start, and I'm honored you chose this guide as one of your resources, but the best way to learn is by putting theory into practice. A note of advice: when you decide to build a model, try to build it together with a team. Since modelling is so subjective, it's always a great idea to bring together a group of diverse thinkers.

Semantic modelling doesn't necessarily need to start and end with a semantic model. For instance, semantic models are the starting point for building knowledge graphs, acting as the graph's semantic backbone. And that's pretty cool because knowledge graphs are everywhere, sometimes on a huge scale. You know, when you Google your favorite actor and that infobox appears on the right hand side of your search giving you a short bio, their height, spouse, etc.? That's made possible thanks to Google's knowledge base, humbly named 'The Knowledge Graph'. But applications of knowledge graphs extend far beyond Hollywood - thank goodness - spanning from healthcare to finance to real-estate to the arts and beyond. In fact there's so much I would like to say on knowledge graphs that it may warrant another guide...

Anyway, after 40-odd pages, I think it's time to part ways. Thank you for taking the time to read this guide, however much of it you may have read. I hope I leave you a little bit more knowledgeable on the semantic modelling universe and, most importantly, I hope you feel even the tiniest bit excited about linked data. After all, we don't need more data. We need more meaningful data.



Acknowledgements

Thank you to **The Hyve** for the continued opportunity to learn about, implement and explore the world of linked data and for this opportunity to share my experience on paper. An especially big thank you to all my wonderful colleagues, both past and present, who have toiled away on many semantic models with me, and from whom I always learn something new.

And thank you to Anca Boon for her help in editing this document and tying it together from beginning to end.

If you find this guide useful, if you have additional content, or if you have a question, please let me know by sending an email to SemanticModels@thehyve.nl.



References

Bechhofer, Sean, et al. "OWL Web Ontology Language Reference." *OWL Web Ontology Language Reference*, W3C, 10 Feb. 2004, www.w3.org/TR/owl-ref/.

Beckett, David, et al. "RDF 1.1 Turtle." *RDF 1.1 Turtle*, W3C, 25 Feb. 2014, www.w3.org/TR/turtle/.

"Introducing Linked Data And The Semantic Web." *Linked Data Tools*, <http://www.linkeddatatools.com/semantic-web-basics>.

Noy, Natalya F, and Deborah L McGuinness. "Ontology Development 101: A Guide to Creating Your First Ontology." <https://Protege.stanford.edu/>, Stanford University, protege.stanford.edu/publications/ontology_development/ontology101.pdf.

"Resource Description Framework (RDF): Concepts and Abstract Syntax." Edited by Graham Klyne et al., *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C, 10 Feb. 2004, www.w3.org/TR/rdf-concepts/.

"RDF Schema 1.1." Edited by Dan Brickley et al., *RDF Schema 1.1*, W3C, 25 Feb. 2014, www.w3.org/TR/rdf-schema/.

"Shapes Constraint Language (SHACL)." Edited by Holger Knublauch and Dimitris Kontokostas, *Shapes Constraint Language (SHACL)*, W3C, 20 July 2017, www.w3.org/TR/shacl/.

Sirin, E. (n.d.). Learn SPARQL - Tutorial: Stardog. Retrieved from <https://www.stardog.com/tutorials/sparql/>

Jayawardana, Vindula. "Ontology Generation and Visualization with Protégé." *Medium*, Medium, 5 Dec. 2017, medium.com/%40vindulajayawardana/ontology-generation-and-visualization-with-prot%C3%A9g%C3%A9-6df0af9955e0.



Appendix

Abbreviations and acronyms

- **RDF**: Resource Description Framework
- **RDFS**: Resource Description Framework Schema
- **TTL**: Turtle
- **URI**: Uniform Resource Identifier
- **OWL**: Web Ontology Language
- **SHACL**: Shapes Constraint Language
- **SPARQL**: SPARQL Protocol and RDF Query Language

Further reading and learning

Some useful resources for those starting out with RDF and linked data:

- *Linked Data Tools 5-part course*: <http://www.linkeddatatools.com/semantic-web-basics>
- *Semantic Web Technologies course from Dr. Harald Sack*: <https://open.hpi.de/courses/semanticweb>
- *Apache Jena's Ontology documentation*: <https://jena.apache.org/documentation/ontology/>
- *Ontotext GraphDB Fundamentals*: www.ontotext.com/knowledgehub/fundamentals/graphdb-fundamentals/.

