Semantic University

Introduction

This set of lessons is an introduction to OWL and an introduction to RDF Schema (RDFS). OWL and RDFS are key Semantic Web technologies that give you a way to write down rich descriptions of your RDF data. The lessons include introductory tutorials suitable to beginners and also an OWL reference for beginners and experienced practitioners alike.

If you haven't already completed the RDF tutorials in *Learn RDF* (http://www.cambridgesemantics.com/semantic-university/learn-rdf), now's the time to do so. RDF underlies OWL and RDFS, and an understanding of RDF will make it easier to go through the lessons in this section.

RDFS Introduction

In RDF Nuts & Bolts (https://cambridgesemantics.com/semantic-university/rdf-nuts-bolts) you learned how to create RDF data and documents. In RDF 101 (https://cambridgesemantics.com/semantic-university/learn-rdf/) we talked about the significance of *vocabulary* in sharing RDF data between systems, but in neither lesson did we detail how to create your own vocabulary for RDF data.

RDFS (**R**esource **D**escription **F**ramework **S**chema) is the most basic schema language commonly used in the Semantic Web technology stack. It is lightweight and very easy to use and get started with. In fact, many of the most popular RDF vocabularies are written in basic RDFS.

In this lesson we continue our technical journey through the Semantic Web technologies stack, introducing RDFS and detailing the usage of RDFS statements used to create your own data models.

Objectives

After completing this lesson, you will know:

- RDFS Basics
- RDFS Examples
- RDFS Syntax Details
- RDFS Introspection

Today's Lesson

RDFS does for the Semantic Web what the universal translators do for Captains Gorn and Kirk, above. Kirk does not need to speak Gornish, and the Gorn does not need to speak English. The universal translator does all the work for them.

RDFS is similar to the universal translator in that it allows data created by different teams for different uses at different times *to be connected* using RDFS and Semantic Web technologies. It provides a kind of universal translation between alien languages. The first step down this path is to define *common vocabularies*.

RDFS Basics

RDF, as we discussed in RDF 101 (https://cambridgesemantics.com/semantic-university/learn-rdf/), is a graph database. RDFS, on the other hand, is *object oriented* in its nature. That is, RDFS is fundamentally about describing classes of objects.

For those of you who have never programmed in Java, C++, Ruby, or any other object oriented language, think of a "Class" as a "Type" of thing, and an "Instance" as a specific example.

For example, if the class is *Used Car For Sale*, then some instances of that class might include:

- "The Red 1966 Ford Mustang for sale in Maryland" or
- "The Silver 1978 Chevrolet Corvette with flip-up headlights" or
- "The Blue 1994 Honda Civic whose open door alarm won't stop beeping."

On the other hand, if the class is *Financial Transaction*, then some instances of that class might include

- "Sell ACME at 10:11AM for $3.14" or
- "Buy ACME at 10:12AM for $7.82."

Some Basic RDFS

So let's make some RDFS.

First of all, RDFS *is RDF!* That's a key point. There is no special syntax, nothing new that you have to learn. RDFS is expressed as RDF. We are going to use the Turtle serialization of RDF.

Before we create some RDFS, let's show some usage of RDFS.

@prefix csipeople: <http://www.cambridgesemantics.com/people/about/ (https://cambridgesemantics.com/people/about/) > .
@prefix csi: <http://www.cambridgesemantics.com/ (https://cambridgesemantics.com/)> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> (http://www.w3.org/2000/01/rdf-schema#>) .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> (http://www.w3.org/1999/02/22-rdf-syntax-ns#>) .
@prefix foaf: <http://xmlns.com/foaf/0.1/> (http://xmlns.com/foaf/0.1/%3E) .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> (http://www.w3.org/2001/XMLSchema#>) .

csipeople:David
rdf:type foaf:Person ;
rdfs:label "David Tester" ;
rdfs:SeeAlso <http://www.linkedin.com/pub/david-tester/33/737/2aa> (http://www.linkedin.com/pub/david-tester/33/737/2aa%3E) .

Here we're describing an RDF resource representing the author of this lesson. They triple is the csi:people rdf:type foaf:Person. This is saying that the resource is of *type* foaf:Person, where foaf:Person is an RDFS class defined elsewhere.

What if we wanted to define our own Person class? We'd do so like this:

csi:MyNewPersonClass
rdfs:label "My Person Class" .

It's really that easy. However, this is pretty naïve. What does it *mean* to be a csi:MyPersonClass? In order to give richness to classes, we have to define properties for those classes, so let's do that.

csi:admires
rdf:type rdfs:Property ;
rdfs:label "Admires" ;
rdfs:domain csi:MyNewPersonClass ;
rdfs:range csi:MyNewPersonClass .

Now we've defined a property that we can use with the csi:MyNewPersonClass.Let's see it in action by modifying our earlier instance data:

csipeople:David
rdf:type foaf:Person ;
rdf:type csi:MyNewPersonClass ;
rdfs:label "David Tester" ;
rdfs:SeeAlso <http://www.linkedin.com/pub/david-tester/33/737/2aa> (http://www.linkedin.com/pub/david-tester/33/737/2aa%3E) .

csi:people:Lee
rdf:type csi:MyNewPersonClass ;
csi:admires csipeople:David .

There are two things to notice there. The first is that I used my new property that I created. The second is that the resource csipeople:David has *two types*. Unlike a language like Java, in which an instance object can have only one type, in RDF there is no such restriction. You can declare an object to be *anything at all*.

RDFS will put some restrictions on how vocabulary is used (rdfs:domain and rdfs:range are two such restrictions used here), and some triple stores can be configured to enforce such restrictions. Regardless, this should reinforce the flexibility of the RDF data model in that it does not rigidly enforce the shapes of instance data in the way that XML or relational databases do. If you need to extend data or use it in a slightly new way, it's as easy as we just illustrated here by creating a new class and a new property and using them immediately.

RDFS Syntax Details

RDF:type

First off, every database has *classes* (for example, *Used Cars For Sale*) of which some things in the database are instances (for example, "The Blue 1994 Honda Civic whose open door alarm won't stop beeping").

In a traditional relational database, each class is represented by a table, and every row in that table is an instance of that class. For example, you can know that a particular Honda Civic is a type of used car for sale simply because you found it in the *Used Cars For Sale* table.

However, in RDFS you would simply make the following statement:

Example:MyHondaCivic RDF:type Example:UsedCarForSale

This statement-based approach is considerably more flexible than storing data in a table because most resources have many different types. For example, all of these might also be true of this car:

Example:MyHondaCivic RDF:type Example:PassengerVehicle

Example:MyHondaCivic RDF:type Example:ThingForSale

Example:MyHondaCivic RDF:type Example:ThingThatCanBeInsured

In RDFS, declaring types is as simple as making statements such as these. In contrast with traditional relational database systems, RDFS types do not have any association with tables, so there is no need to maintain the usual primary and foreign key relationships in order to implement these sorts of multiple types. Rather, you simply define what kind of class something is by using a single **RDF:type** statement that says what kind of class that resource is.

Another feature common to all database systems is that all instances of the same class have similar kinds of properties. For example, all instances of the *Used Car For Sale* class will typically have a *Price*, an *Odometer Reading*, a list of *Previous Owners*, and so on.

In relational databases, you specify the properties that belong to a class by adding columns to a table associated with that class. Or, for properties that can themselves have more than one property (i.e., properties such as *Previous Owner*, which might have its own properties, such as *Names* and *Addresses*) you would have to define a new table with an appropriate foreign key relationship.

RDF:Class and RDFS:Property

In RDFS, this common task of relating classes and properties is greatly simplified and rendered far more flexible. For example, to define the new *Used Car for Sale* class, simply insert this statement into the database:

Example:UsedCarForSale RDF:type RDF:Class

To define properties of this new class, insert statements such as these:

Example:Price RDF:type RDFS:Property

Example:PastOwners RDF:Type RDFS:Property

But then again, how would you actually say that **Example:Price** is not just a random *Property*, but is actually a property of *Used Car For Sale*? And how do you say what kind of data **Example:Price** is (number, string, URL, etc)?

In general, the domain of a property is the set of *all* things to which that property applies. In relational databases, you specify domains by choosing which tables to add columns to.

However, in RDFS, you simply make yet another direct statement, such as:

Example:Price RDFS:Domain Example:UsedCarForSale

RDFS:Domain and RDFS:Range

Similarly, the range of a property is the set of values that the property can accept. In relational databases, ranges are specified by choosing the “data type” of the various columns.

Again, however, RDFS requires only a simple statement:

Example:Price RDFS:Range XSD:int

Here, **XSD:int** refers to an integer number. For more details on syntax, see the brief lesson on XSD Datatypes (https://www.w3.org/TR/xmlschema-2/).

RDFS ranges are not limited to simple primitive data types. They can also refer to any class that you define. For example, you might want to state:

Example:PastOwners RDFS:Range Example:Person

Here, **Example:Person** just refers to another **RDFS:Class**, to which you can then go and add other properties (*Name*, *Address*, etc.) in the same way, if you like.

Aside from the schema-oriented properties discussed above, RDFS also provides for several other constructions that are commonly used across databases. These include:

Other RDFS Basics

- **RDFS:Label** – A string of text describing the resource
- **RDFS:Comment** – A potentially longer comment about the resource
- **RDFS:SeeAlso** – Links to other "relevant" resources
- **RDFS:Literal** – Something that is a primitive data type

Constructing a basic data model using RDFS is easy because all you have to do is add statements similar to those described above. One reason that RDFS is so flexible is because these statements can be updated or removed at any time.

More Advanced RDFS

RDFS:subClassOf

Most *Cars* have a *Miles per Gallon* rating, including *Used Cars For Sale*. And all resources that are for sale have a *Price*, including *Used Cars for Sale*.

Wouldn't it be nice if RDFS provided a way for more specific classes (**Used Car for Sale**) to inherit properties from more general classes (**Car**)?

In RDFS, such relationships are easy to build and, once again, require only one statement each:

Example:UsedCarForSale RDFS:subClassOf Example:Car, Example:ResourceForSale

Specifically, any statement such as X RDFS:subClassOf Y simply maintains

- that all instances of class X are also an instance of class Y and
- that all the properties of Y are also properties of X.

So, for example, rather than asserting:

Example:Price RDFS:Domain Example: UsedCarForSale

Example:MilesPerGallon RDFS:Domain Example:UsedCarForSale

You could instead assert:

Example:Price RDFS:Domain Example:ThingForSale

Example: MilesPerGallon RDFS:Domain Example:Car

Example:UsedCarForSale RDFS:subClassOf Example:Car, Example:ThingForSale

Since the number of properties per class can sometimes be quite large, the ability to specify these sorts of subclass relationships can dramatically cut down on "clutter" and increase the understandability of your data.

RDFS:subPropertyOf

A final common structure worth considering is **rdfs:subPropertyOf**, which behaves similarly to **subClassOf**. For example, consider the following pair of statements:

Example:Mother RDFS:subPropertyOf Example:Parent

Example:Fred Example:Mother Example:Franny

These two statements say that *Fred's Mother* is *Franny* and also that *Mother* is a sub-property of *Parent*. What **subPropertyOf**means is that we now also know this:

Example:Fred Example:Parent Example:Franny

Semantic Introspection using Classes and Instances

Since every known database in the (semantic) universe shares this distinction between classes and instances, it *is* possible for them to understand a little bit about each other. For example, two database systems can talk about how many different classes they have and how many instances they have of each class.

At first glance, that might not seem to be a very interesting conversation. However, like any two victims of a blind date, these alien datasets might actually be surprised at how many things they do, in fact, have in common. RDFS attempts to describe all these commonalities.

If you know the concept of *reflection* in Java and other languages, being able to ask a database about its classes and instances is analogous. In fact, some tools—such as cwm (http://www.w3.org/2000/10/swap/doc/cwm.html)—do not require specific classes or properties to exist in order to display RDF for browsing.

Other Semantic Web frameworks starting with MIT's Haystack have used this quality to create very dynamic user interfaces that dynamically construct themselves around the data that the user pulls in. For example, if you pull data from an RDF database and some of it happens to be rdf:type Author and your Haystack browser has a *Lens* to view things of rdf:type Author, then Haystack will use that Lens. In this way, the Haystack browser, via extension, could display any conceivable kind of day and be extended on the fly. Other RDF browsers have since used the Lens concept.

But we digress. The fact is that the simplest application of RDFS is to define a vocabulary that is shared between two applications to ease data interchange between those applications. And, as stated many times through Semantic University, two applications do not have to agree on a data model completely, or even ahead of time, in order for this to work.

Conclusion

Although RDFS provides a solid base for understanding and sharing information across datasets, it is possible to do much more. In fact, for all but the simplest shared vocabularies in the Semantic Web world, practitioners eschew RDF in favor of the far more expressive OWL. The next lesson, OWL 101 (https://cambridgesemantics.com/semantic-university/owl-101), on OWL (Web Ontology Language) goes into more detail on this topic.

‹ PREVIOUS LESSON
(HTTPS://CAMBRIDGESEMANTICS.COM/BLOG/SEMANTIC-

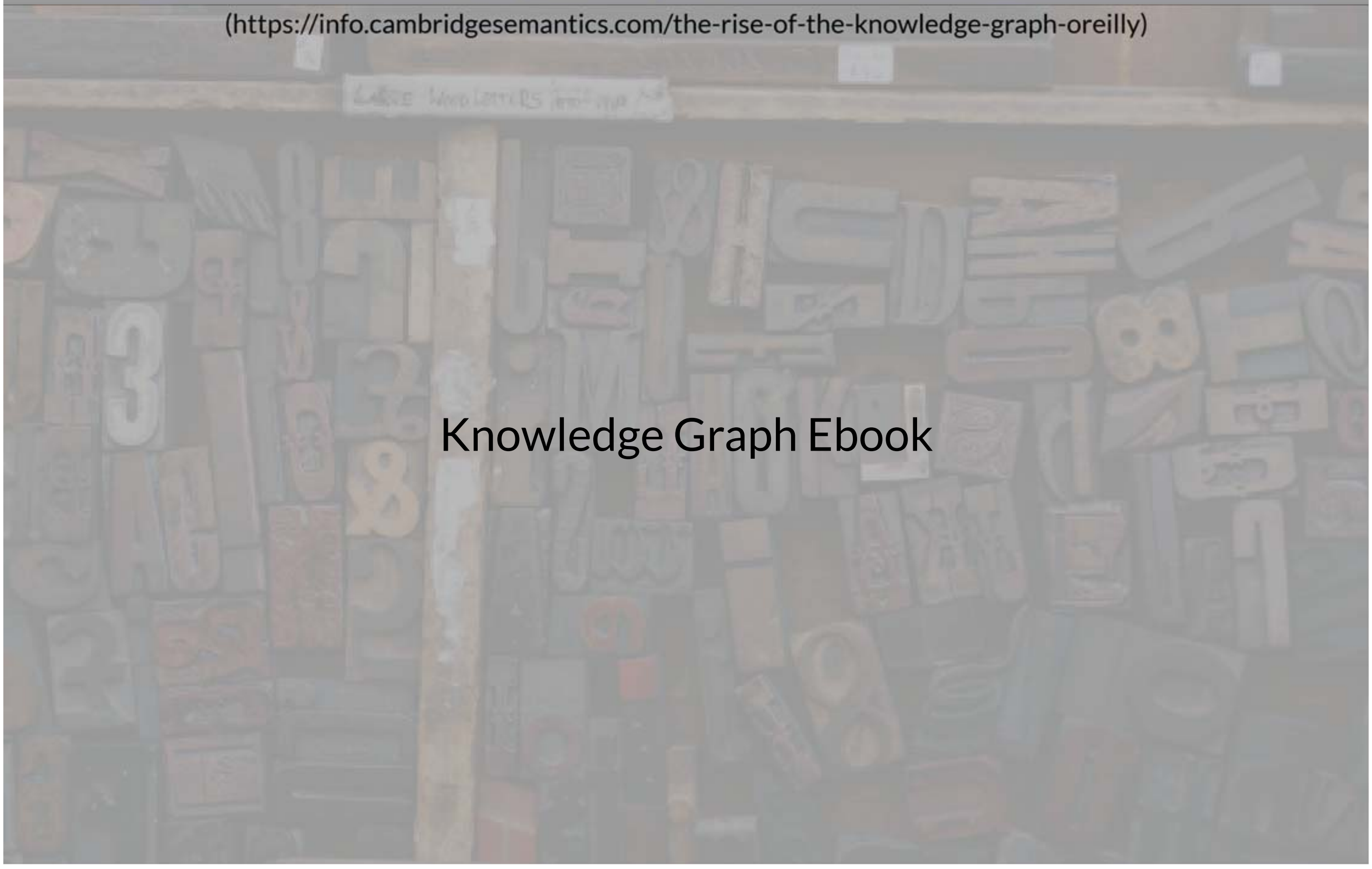
NEXT LESSON ›
(HTTPS://CAMBRIDGESEMANTICS.COM/BLOG/SEMA

UNIVERSITY/LEARN-RDF/RDF-NUTS-BOLTS-2/) UNIVERSITY/LEARN-OWL-RDFS/OWL-101/)

(https://info.cambridgesemantics.com/what-is-data-fabric-design)

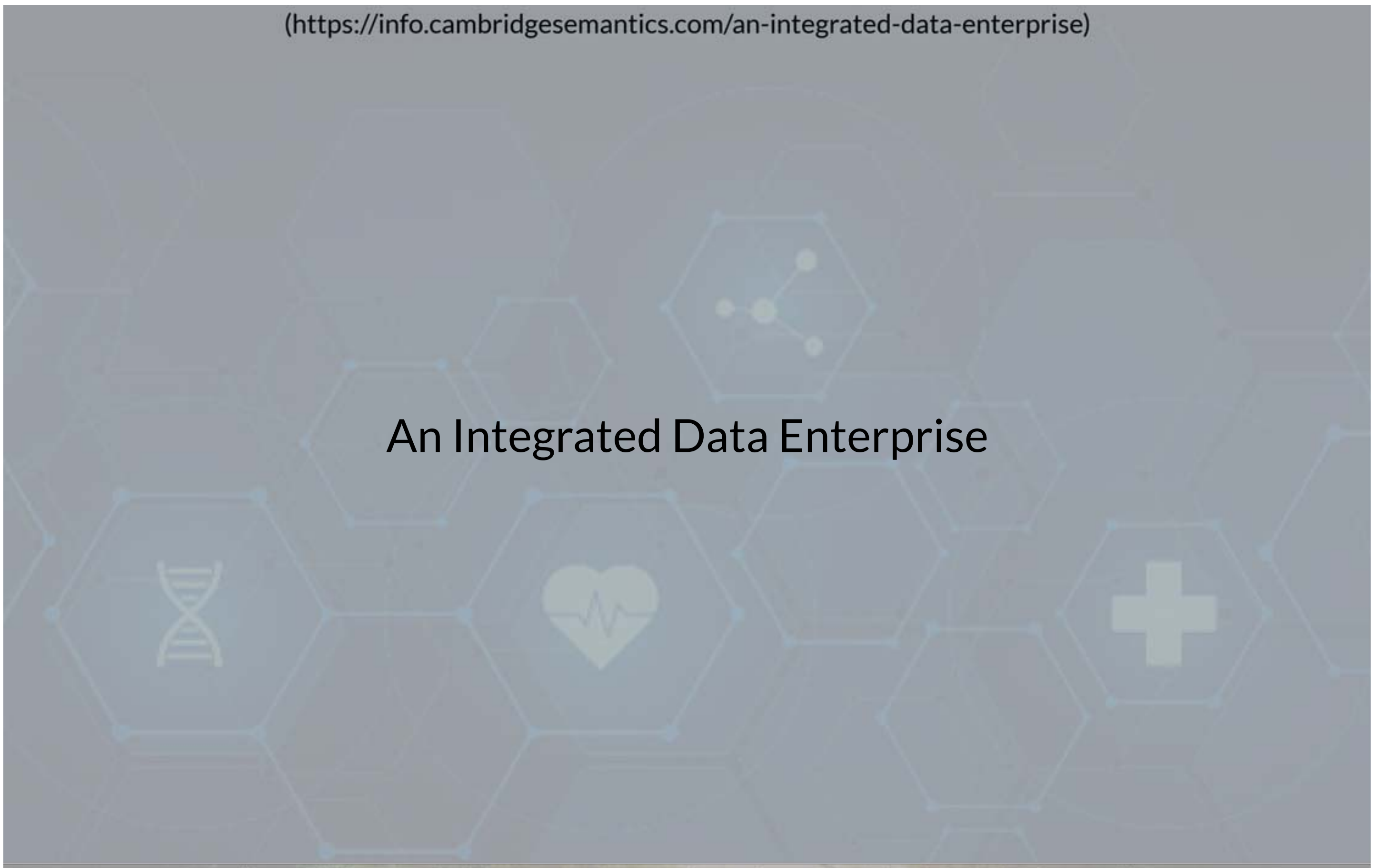
What is Data Fabric Design?

(https://info.cambridgesemantics.com/the-rise-of-the-knowledge-graph-oreilly)



Knowledge Graph Ebook

(https://info.cambridgesemantics.com/an-integrated-data-enterprise)



An Integrated Data Enterprise

(https://blog.cambridgesemantics.com/sparql-one-standard-to-rule-them-all)

SPARQL - One Standard to Rule them All

CAMBRIDGE SEMANTICS

(http (htt
(htt (htt

1 Beacon St | 15th Floor | Boston, MA 02108