Understanding SPARQL

Create journaling micro-blogs with the semantic Web

Skill Level: Advanced

Andrew Matthews (matthews.andrew@gmail.com)
Architect and developer
Freelance

22 Apr 2008

The Semantic Web, a knowledge-centric model for the Web's future, supplements human-readable documents and XML message formats with data that can be understood and processed by machines. SPARQL Protocol and RDF Query Language (SPARQL) is to the semantic Web as SQL is to a relational database. It allows applications to make sophisticated queries against distributed RDF databases, and is widely supported by many competing frameworks. This tutorial demonstrates its use through the example of a team tracking and journaling system for a virtual company.

Section 1. Before you start

This tutorial is aimed at developers with little or no experience in producing semantic Web applications. It doesn't require any programming or developer tools, but it does assume some familiarity with Web fundamentals. When you complete this tutorial, you will know how to produce RDF and OWL ontologies in the Turtle language. You will know how to host the ontologies using Jena and Joseki and you will know how to query them using SPARQL.

About this tutorial

Frequently used acronyms
This tutorial introduces SPARQL and the data formats it is based on. It also covers the RDF, RDF Schema, OWL, and Turtle knowledge representation languages. With these languages, you build ontologies or domain models. For the example used throughout this tutorial, you'll build the ontologies and queries for a journaling and booking system to produce semantically tagged twitter-like micro-blogs. You'll query those blog entries to find those in your company with the skills to make up the team for a project that you are bidding for.

The tutorial follows these steps:

- An introduction to the semantic Web including RDF, OWL and SPARQL
- How to set up an RDF Triple Store using Joseki
- Writing the SPARQL queries for the journal system

Prerequisites

You will need the following tools to follow this tutorial:

- Java™ environment—The Java runtime environment is needed to allow you to run Joseki, the SPARQL server.
- Joseki—This is an open source SPARQL server produced by Hewlett-Packard. You can get it from http://www.joseki.org/. Follow the links to the download area at SourceForge. Download the latest version of Joseki to your machine, and choose an area to unzip the file to. Take a note of that location, because you will make use of it to create a simple batch file that will fire up the server.
• Jena—The Semantic Web framework (provided as part of Joseki).
• Any text editor.

A team at Hewlett-Packard Labs has been developing Joseki since around 2003. It is a layer providing HTTP support to the Jena semantic Web framework, also produced by Hewlett-Packard. It is probably the most popular platform for semantic Web work, and one of its developers is also an editor for the SPARQL standard, so it seldom lags behind the standard by very much, and in some cases is a test bed for new ideas.

Later on, in Configuring Joseki, you can see detailed instructions on how to configure Joseki with the RDF files that you will create.

Section 2. The Semantic Web layer cake

This section will define Semantic Web terminology and explain what RDF and OWL are, how they work, and how you can use them to build domain models for a Semantic Web application.

SPARQL history

SPARQL is built on top of several key technologies, in the same way that HTTP and HTML (the foundations of the World Wide Web) are built on deeper and lower level systems like TCP/IP. Before you see what SPARQL is, look at the key standards, why they’re there, and what that means to you as a budding Semantic Web developer.

In 1997, Tim Berners-Lee pointed out that HTML and the World Wide Web were limited. It was not designed for dynamic Web-based applications let alone the complex distributed systems produced these days. HTML and HTTP were just a (vital) step towards a grander vision of direct, semi-autonomous machine to machine communications that will be for us what the WWW was when all we had was FTP. That vision is founded on RDF (the Resource Description Framework).

Since RDF can describe anything, it can describe itself, allowing it to be built up in thin layers that add more and more richness. This thin-layers approach is intended for the production of a vocabulary stack. The diagram in Figure 1 shows the layers defined by the W3C. The layers that sit on top of RDF currently include RDFS and OWL, (there is expected to be future work to build on OWL). RDFS, the RDF Schema language, adds classes and properties to RDF. OWL (the Web Ontology
Language) extends RDFS, providing a richer language to define the relationships between classes. A richer language makes it possible to use automated inference engines to create smarter systems.

**Figure 1. The Semantic Web layer cake: the technology stack for the W3C Web architecture**

In the next section, you see how RDF is constructed, and how you can use it to build models of the world.

**RDF**

RDF has been called a "meta description language," but that's just a fancy way to say that it describes things. It describes things in the same way that people do; with sentences such as "The crow eats corn" or "Joni loves Chachi". Each one of those sentences has a subject (the crow, Joni), a predicate (eats, loves), and an object (corn, Chachi). In RDF, those subject-predicate-object sentences are *triples*. RDF uses triples to describe anything.

One way to represent these triples visually is to use an RDF Graph, which is a collection of statements in RDF. A graph is defined in terms of nodes and arcs. In RDF the nodes are resources and the arcs are predicates—that is, they are statements about the relationships between the subject and object nodes. At its core, the RDF spec is all about defining a graph; other things like serialization formats are very much of secondary importance. The subject and object elements define nodes in the graph (also known as resources since they are the target of URIs). Each predicate defines a relationship between the two nodes that the triple references.

To make the graph available to the rest of the Web, you host the RDF files in a *triple store* — in other words, a place to store the triples that make up your graph. Once you store the RDF graph in a triple store and expose it to the Web, others can query that graph using SPARQL (see Figure 2).

**Figure 2. A visual representation of an RDF graph with one statement of a predicate relationship between Subject and Object**

In RDF, each node and predicate is identified by a URI. RDF can also allow nodes that are not identified using URIs. These are called Blank Nodes or Blank Node Identifiers and are used as temporary, internally visible identifiers for local references. The RDF specification states that although they have provided a standard for the serialization of RDF as XML, any equivalent structure will also be okay. The following RDF XML is for a triple that says something about the author (see Listing 1).

**Listing 1. RDF XML for a triple that says something about the author**
This says that "Andrew plays a guitar", or more precisely "Something referred to as Andrew plays an instrument referred to as a guitar". You'll agree that considering the minimal amount of information that it imparts it is quite long. Here's the same graph represented in a language called Turtle, which is also controlled by the W3C (see Listing 2).

Listing 2. The same RDF as Listing 1, but using Turtle

```
@prefix : <http://aabs.purl.org/music#> .
```

Much better! That's a lot higher signal to noise ratio, and is worth using just for that reason alone. The W3C intends Turtle as a human readable and writable language for the production of RDF. SPARQL uses Turtle, so all the examples from now on will use it too. There is a full-stop delimiter at the end of a triple, and the whitespace doesn't matter that much (see Figure 3).

Figure 3. A graph with a single statement asserting that 'Andrew' is related to 'guitar' by the 'playsInstrument' relationship

RDF has no intrinsic data types other than an XML Literal type for embedding XML in an RDF file. It is able to make use of data types defined using XML Schema and those are expected to be most frequently used.

In Turtle the data type of a literal is added to the end of the data like Listing 3.

Listing 3. Using XML Schema data types within RDF

```
@prefix xsdt: <http://www.w3.org/2001/XMLSchema#>.
@prefix mu: <http://aabs.purl.org/music#>.
```

The `^^` part attaches the string literal representation of the data type (the "1987-06-13T10:30:00" part) with the data type declaration (the `xsdt:dateTime` part). So this states that "Andrew bought the guitar on 13th June 1987 at ten thirty". The exception to this rule is any type that can be unambiguously parsed without requiring clues as to its type. So an unadorned number like 5 is clearly an integer. Likewise an unadorned literal string, Andrew, is only parsable as
a string. The same case applies to boolean and decimal types. Integers, booleans, and decimals can be given just as a bare number without quotes. Here's a statement using an integer: :guitar :timesRestrung 500.

Strings can be quoted (according to the normal rules of python, Tim Berners-Lee's current favorite language) like so: :guitar :makersModel "GL350" or like Listing 4.

Listing 4. Using Python-style triple quotes to span lines

```
:guitar :makersModel """GL350
some more text on a new line (provided you use triple quotes)"
```

Figure 4 shows the graph of Listings 2, 3, and 4.

Figure 4. The graph of statements about Andrew and his guitar
RDF defines no standard data structures. Just as with normal programming languages, the language designers have extended it with simple structures. With RDF you get 'Bags', 'Sequences' and 'Alternates lists'. The implementation of these structures is achieved through the use of triples, but is much too involved to bother with here. If you want to delve deeper, check out the RDF Primer (see Resources for a link). Turtle has native syntactic support for these data structures. Here's how a list is declared: :andrew :child (:emily :thomas).

This is equivalent to Listing 5.

Listing 5. Using lists to apply a subject and predicate to multiple objects

```
```

You can make use of lists within the subject of a triple as well: (:thomas :emily) :parent :andrew .

To declare a resource in RDF without giving it a URI, you can use a blank node identifier—a temporary name for local reference. Listing 6 shows a sample from the JournalEntries ontology:

Listing 6. Using predicate-object lists to make many statements about a single subject

```
_:JohnConnor a u:User;
    u:domainLogin "someDomain/john.connor";
    u:displayName "John Connor" .
```
_:JohnConnor is a blank node (signified by the leading underscore) that can be used locally for references. You might want to write your RDF like that rather than use ":JohnConnor" (which is externally visible) simply because domain logins tie the resource to LDAP (Lightweight Directory Access Protocol), which may be your local source of truth. Both options are open to you.

Another blank node syntax uses "[]" to represent a resource that you don't even want to give a local name to. It's only scoped to the current triple or predicate-object list (see Listing 7).

**Listing 7. Using blank node syntax with predicate-object lists to create a full object without any identifier**

```rdfs
[] a j:JournalEntry;
  j:date "20080205T09:00:00"^^xsd:dateTime;
  j:user _:JohnConnor;
  j:notes "Today I learnt how to defraud ATM machines and how to field strip a machine gun blindfolded.";
  j:tag "armaments";
  j:tag "cash".
```

You might choose this idiom if you didn't have a name or didn't want to pollute the namespace of your application with meaningless identifiers. After all, with SPARQL you can query resources by the data they are related to, rather than just URIs. When using a blank node, the triple store will independently allocate a blank node identifier. This form of notation is for one-off resource uses because it will not be directly linkable after the final full stop at the end of the resource definition.

One thing you'll notice from the two examples above is the use of semi-colons to allow multiple triples to share the same subject. That's known as *predicate-object lists* and is frequently used, since it allows you to make several statements together about a subject. In fact, the blank node notation used above would be completely useless without predicate-object list notation.

**RDFS and OWL**

RDF was consciously designed to support its own layered extension with more abstract vocabularies. The first way that it was extended was with the RDF Vocabulary Description Language, more commonly known as RDF Schema or RDFS. RDFS extends RDF with features to define classes, properties, and inheritance. It's pretty much the full toolkit of the object oriented designer. OWL extends RDFS to provide an extremely rich toolkit to describe the properties and relationships of a class. OWL is particularly over-endowed with properties to describe the exact nature of the relationship between two classes. The main motivation of OWL is to place the semantics of an ontology on a firm footing to allow inference engines to make novel deductions about your data.
RDF classes and properties

RDFS defines a triple predicate called rdfs:type that declares a resource to be of a type. It also allows a resource that is a class to declare that it descends from another class. Listing 8 is an example of a type declaration.

**Listing 8. Defining a class hierarchy using rdfs:type and rdfs:subClassOf**

```rdfs
:HeavenlyBody rdfs:type rdfs:Class.
```

Figure 5 defines a small class hierarchy.

**Figure 5. The class hierarchy created in Listing 8 (as UML represents the graph)**

Defining class instances is very similar to class declaration. RDFS distinguishes between class declaration triple and class membership with the rdfs:subClassOf predicate used to indicate that the subject is also a class and not an instance (see Listing 9).

**Listing 9. Defining instances of a class using rdfs:type**

```rdfs
:Ceres, :Vesta rdfs:type :Asteroid
# ...
```

Turtle provides convenient shorthand to make type declaration easier (see Listing 10).

**Listing 10. Using a as a shorthand for rdfs:type**

```rdfs
:GasGiant rdfs:subClassOf :Planet.
```

a is merely a short form of rdfs:type; it adds no extra meaning.

RDFS provides properties on classes, too. Listing 11 shows how they are declared.

**Listing 11. Defining a property on the class :HeavenlyBody**

```rdfs
:massKg
 rdfs:domain :HeavenlyBody ;
 rdfs:range xsd:double .
```
This declares a property called :massKg that maps from :HeavenlyBody to a
double value. In other words, it declares that all heavenly bodies can have a mass.
The earlier instance declarations can now be rewritten (see Listing 12).

Listing 12. Using a property on an instance

```ruby
:Earth a :Planet;
  :massKg "5.9742e24"^^xsd:double.
```

RDFS's facilities are quite bare in comparison to OWL. OWL has a vast collection of
ways to specify the subtleties of a relationship between two classes. There's not
enough space to really do it justice, but here are a few features that will give you a
taste of what you might achieve with OWL (see Listing 13).

Listing 13. Using OWL to give a more detailed description of a property

```ruby
:hasUsedSkill a owl:ObjectProperty;
  rdfs:domain :User;
  rdfs:range :Skill;
  owl:equivalentProperty :hasSkill;
  owl:inverseProperty :hasBeenUsedBy.
```

This tells you that the property :hasUsedSkill relates two objects of type :User
and :Skill. It is the same as using the :hasSkill property (they mean the same
thing). If elsewhere, it asserts that a skill :hasBeenUsedBy someone, that also
means that person :hasSkill and :hasUsedSkill as well. In other words, OWL
allows you to give guidance to an inference engine to let it know the equivalent
meaning implicit in the various properties defined in an ontology. There's much more
to OWL, but for now you can use RDFS, since this tutorial doesn't cover the use of
inference engines.

Another topic that I won't cover here is the age old debate over whether the object
oriented approach to data modelling is superior. It's clear that for now object
orientation will rule the world of application development, so it is critical that there is
an easy mapping possible between the ontological and the object domain. Systems
like LinqToRdf show that such a mapping is indeed possible, and the real issue
you're likely to face is the sheer wealth of information that you can place in an OWL
ontology. That's a blessing in disguise.

That's quite enough theory for now. You now know enough RDF to focus on
SPARQL. This brief introduction has hardly touched the layers that build on RDF,
nor any of the ontologies that have gained real popularity worldwide; like FOAF,
SIOC, or Dublin Core. It hasn't delved into the fascinating world of inference
engines, rules mark-up languages, or formulae, all of which have great power and
promise. All that would probably fill several other tutorials each. Now to move on to
more practical matters. You need to set up a triple store and SPARQL endpoint.
That done, you can then begin to develop some ontologies.

Defining an ontology in OWL

This tutorial develops a simple program that allows you to define journal entries describing what you've done. Later on you'll develop the SPARQL queries to ask various questions about who in the virtual company has done what.

Earlier it was mentioned that these journal entries were a twitter-like micro-blog. You need to know what all the moving parts of the tutorial example are going to be for the explanations that follow to make sense. The journal system will need a UI to display entries. You won't cover that here, since it's familiar territory for most developers, and is covered in detail by many other tutorials on developerWorks (see Resources for links).

An ontology will define the format of the data that you will create. The ontology defines classes with properties. Your micro-blog entries will be written in fragments of Turtle that conforms to the ontology. The data when it is stored in your triple store is then available for querying using SPARQL. The results you get back from a SPARQL query are in an XML results format that packages the variables that the query matches. You can easily extract the information from the XML and display it on the Web.

First, you need to define the journal ontology (see Listing 14). This will be really simple—just two classes. This defines two classes called JournalEntry and User. Each User can define as many journal entries as they like, and each journal entry can contain a date, a reference to the user and a set of tags.

Listing 14. A small (but complete) ontology for the journaling system

```
@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@prefix string: <http://www.w3.org/2000/10/swap/string#>.
@prefix os: <http://www.w3.org/2000/10/swap/os#>.
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix j: <Journal.n3#>.
@prefix : <#>.

#JournalEntry class
:JournalEntry a owl:Class .
:dates rdfs:domain :JournalEntry;
    rdfs:range xsdt:datetime;
    owl:cardinality 1.
:users rdfs:domain :JournalEntry;
    rdfs:range :User ;
    owl:cardinality 1.
:notes rdfs:domain :JournalEntry;
    rdfs:range xsdt:string;
    owl:cardinality 1.
:tags
```
Now you'll define a few JournalEntry elements in the JournalEntries.n3 file. The first thing you should do is define a :User instance (see Listing 15).

**Listing 15. An entry in the ontology describing the author**

```n3
_:AndrewMatthews a :User;
  :domainLogin "someDomain/andrew.matthews";
  :displayName "Andrew Matthews" .
```

You can see that this entry conforms to the definition in Journal.n3. Now that you have a user entry, you can define some journal entries (see Listing 16).

**Listing 16. Creating anonymous journal entries using the ontology**

```n3
[] a :JournalEntry;
  :date "20080204"^^xsdt:datetime;
  :user _:AndrewMatthews;
  :notes "Today I wrote some more content for the great new SPARQL tutorial that I've been preparing. I used some N3 to do it in, and I defined a simple ontology for defining journal entries. This is an example of one of those entries!"
  :tag "N3";
  :tag "RDF";
  :tag "OWL";
  :tag "tutorial".

[] a :JournalEntry;
  :date "20080205"^^xsdt:datetime;
  :user _:AndrewMatthews;
  :notes "Today, I wrote some more content for the tutorial. I wrote a section that describes how you set up Joseki."
  :tag "N3";
  :tag "Java";
  :tag "Joseki";
  :tag "configuration";
  :tag "Jena".
```

This time you've not given names to the entries in any way, just defined anonymous
instances that conform to the class definition of the :JournalEntry class. In the tutorial source code files, you can find more of these entries so the queries are a little more interesting.

You'll find the rest of the journal entries, supporting ontology definitions and everything else you'll need in the source code.

1. Download the zip file at the top of the tutorial
2. Copy the .bat file to the Joseki directory
3. Copy the config file into the Joseki directory as well to overwrite the existing config file Joseki.ttl.
4. Modify the paths in the RunJoseki.bat file to point to where you downloaded Joseki to.
5. To start the triple store and SPARQL server, just double click on the RunJoseki.bat batch file.

The console screen that appears should initially resemble Listing 17.

**Listing 17. A typical listing produced by Joseki when it starts up**

```plaintext
Starting Joseki
12:19:17 INFO Configuration :: ===== Configuration =====
12:19:17 INFO Configuration :: Loading : <joseki-config.ttl>
12:19:18 INFO ServiceInitSimple :: Init: Example initializer
12:19:18 INFO Configuration :: ===== Datasets =====
12:19:18 INFO Configuration :: New dataset: JournalDataset
12:19:18 INFO Configuration :: Default graph : Journal.n3
12:19:18 INFO Configuration :: ===== Services =====
12:19:18 INFO Configuration :: Service reference: "sparql"
12:19:18 INFO Configuration :: Class name: org.joseki.processors.SPARQL
12:19:18 INFO SPARQL :: SPARQL processor
12:19:18 INFO SPARQL :: Locking policy: none
12:19:18 INFO SPARQL :: Dataset description: true // Web loading: true
12:19:18 INFO Configuration :: Dataset: JournalDataset
12:19:18 INFO Configuration :: ===== Bind services to the server =====
12:19:18 INFO Configuration :: Service: <sparql>
12:19:18 INFO Configuration :: ===== Initialize datasets =====
12:19:19 INFO Configuration :: ===== End Configuration =====
12:19:19 INFO Dispatcher :: Loaded data source configuration: joseki-config.ttl
12:19:19 INFO log :: Logging to
12:19:19 INFO log :: org.slf4j.impl.Log4jLoggerAdapter@29d65b via
12:19:19 INFO log :: org.mortbay.log.Slf4jLog
12:19:19 INFO log :: jetty-6.1.4
12:19:19 INFO log :: NO JSP Support for /, did not find
12:19:19 INFO log :: org.apache.jasper.servlet.JspServlet
12:19:19 INFO log :: Started SocketConnector@0.0.0.0:2020
```

You are now ready to start making SPARQL queries. Joseki contains its own Web server that hosts a SPARQL query form. The form will be located at
http://localhost:2020/sparql.html. Once you start to run Joseki, click on this link and the SPARQL query form will appear (see Figure 6).

**Figure 6. SPARQLer is a handy Web page to try your queries on Joseki**

Joseki is an HTTP endpoint that receives and responds to SPARQL queries. It sits on top of the Jena Semantic Web framework that provides triple store functionality. Joseki also provides a simple Java servlet engine to host the Web endpoint. It uses this to render a Web query form to allow you to present your queries by hand. In production code you won’t use the query form. Instead, you’ll use a SPARQL API (such as the Jena framework provides) to present the queries programmatically. In fact you can use SPARQL without ever going out onto the Web. If you have a local triple store, you can talk to it direct using SPARQL. The SPARQL protocol defines how you communicate if you went onto the Web.
SPARQL in action

You've covered RDF, OWL, and Turtle as well as the process of downloading and installing the Joseki SPARQL server. Now you will see SPARQL in action. You'll use the SPARQLer query page to produce and test your queries. It's probably best if you use Firefox for the query development since it doesn't erase the query if you jump back and forth between the query and results pages of SPARQLer. The queries developed from now on will relate to the example app. You'll create queries to extract slices of data from the journal example.

SPARQL allows you to query for triples from an RDF database (or triple store). Superficially it resembles the Structured Query Language (SQL) used to get data from a relational database. The resemblance is more to help developers who are familiar with databases, since a triple store and a relational database are fundamentally different beasts. A relational database is table based, meaning that data is stored in fixed tables with a foreign key relationship that defines the relationship between rows in the tables. A triple store stores only triples, and you can pile the triples as high as you like while describing a thing. With relational databases you are confined to the layout of the database.

RDF doesn't use foreign and primary keys either. It uses URIs, the standard reference format for the World Wide Web. By using URIs, a triple store immediately has the potential to link to any other data in any triple store. That plays to the distributed strengths of the Web.

Because triple stores are large amorphous collections of triples, SPARQL queries by defining a template for matching triples, called a Graph Pattern. As mentioned in the section on RDF and graphs, the triples in a triple store make up a graph that describes a set of resources. To get data out of the triple store using SPARQL, you need to define a pattern that matches the statements in the graph. Those will be questions like this: find me the subjects of all the statements that say 'plays guitar'. Listing 18 shows a query over data defined using the ontology about music in Listing 1.

Listing 18. A SPARQL query that finds out what instrument andrew plays. This works on the graph defined in Listing 1

```sparql
PREFIX : <http://aabs.purl.org/music#>
SELECT ?instrument
WHERE {
}
```

Turtle is also used in this tutorial because SPARQL itself uses a form of Turtle to represent the query graph pattern. The query says "find all the triples that have a subject of :andrew and a predicate of :playsInstrument, then get the objects of
the matching triples and return them”. Of course, there’s a lot more to SPARQL than
that, but you'll get to that once the triple store is up and running. You want to
become productive as quickly as possible without too much theory, but you can’t run
a triple store without an ontology, and you can’t test queries without a triple store. So
first you define the ontology, then you host it, then you make the queries. Now that
you have defined the ontology and defined some instance data based on it, you are
ready to configure Joseki with your data.

Configuring Joseki for the journal dataset

This tutorial uses the Joseki SPARQL server because it is free and open source,
and because it is a widely popular platform. You can opt to use other triple stores
and SPARQL endpoints to host the ontology and run the queries.

Joseki and Jena are written in the Java language, but for this tutorial you don't need
to code against it directly. You only need to properly configure the server to point to
your files, and tell it what graphs to construct for you. A Joseki configuration file is a
Turtle file that defines resources that describe the graphs to be served. You describe
data files and you can optionally define a reasoning engine to process rules defined
in your RDF. In a real-world application, you would not use the SPARQLer query
page. Instead, you would use an API that allows you to programmatically send
queries to Joseki and decode the results for use inside your program (see Figure 7).

Figure 7. A typical architecture for a Semantic Web application
First you define a service that hosts the data that you will serve. Call it
"JournalService". Listing 19 shows what the service configuration will look like.

Listing 19. Service configuration

```
[] rdf:type joseki:Service ;
  rdfs:label "SPARQL on the Professional Journal model" ;
  joseki:serviceRef "journal" ;
  joseki:dataset _:JournalDataset ;
  joseki:processor joseki:ProcessorSPARQL_FixedDS .
```

Jena and Joseki (the tools you're using for this tutorial) both use Turtle for their
configuration settings. In short, this snippet defines an entity of type
joseki:Service that is labeled "SPARQL on the Professional Journal model."
Other entities in the configuration file indirectly refer to it by the
joseki:serviceRef "journal". It will host the dataset defined later on and called
_:JournalDataset.

Now you must define the dataset referred to earlier as _:JournalDataset. To do
this, define an entity of type ja:RDFDataset as in Listing 20.
Listing 20. Dataset configuration

_:JournalDataset rdfs:label "JournalDataset" ;
  ja:defaultGraph _:JournalGraph ;
  rdfs:label "JournalDataset" ;
  ja:namedGraph [
    ja:graph _:JournalGraph ] ;

This piece of RDF defines an RDFDataset identified by _:JournalDataset that has a default graph defined as _:JournalGraph. It defines a URI that can access the data in the graph, and it also proved default access to the contents of the dataset. Lastly, you need to define a Graph for the ontology that you're working on, and for the data based on it. Earlier, you referred to the graph as _:JournalGraph. Listing 21 show how it is defined.

Listing 21. Graph configuration

_:JournalGraph rdfs:label "JournalGraph" ;
  rdfs:label "JournalGraph" ;

This last element defines a graph—your own little piece of the Giant Global Graph, or GGG, as Tim Berners-Lee recently tried to label the Semantic Web. Remember, all RDF is only for defining graphs. It's no surprise, then, that the configuration of a triple store is all about finding data and putting it into graphs, or specifying ways to expose those graphs to the world.

Remember that wherever you unzipped the tutorial files is the location where you point the ja:content’s URIs should be pointed. You will define a memory model object called JournalGraph linked to two external files on disk called Journal.n3 and JournalEntries.n3. Those two files are where you will next turn your attention—they're where you define your ontology (the Semantic Web term for what programmers would call an object or domain model).

Section 3. Writing queries in SPARQL

That's a lot of preliminaries to get out of the way, just to be able to write your first query. You've covered a lot of ground, though, representing ideas in RDF, RDFS, OWL as well as Turtle, which is probably the best way to represent RDF if you produce it by hand. You've also seen how to put together and host an ontology and
how to configure a SPARQL endpoint that you can publish on the Web.

You can move, at last, to the team journaling and tracking system. Each query will serve to enhance team communication and to help a resource manager find the right staff for a job in advance of making commitments to clients. Here’s the list of queries you will develop. They are ordered so that you can use them to gradually introduce progressively more complex features of the SPARQL query language.

1. Get a list of all bookings in date order
2. Get notes for all journal entries filtered by keyword
3. Get the list of all skills/technologies registered by a given user
4. Get a list of all the users who have skills in a specific set of required skills
5. Get the list of users booked for a date range
6. Answer questions about current status of teams and clients

SPARQL provides four different forms of query: SELECT, ASK, DESCRIBE, and CONSTRUCT. I’ll provide a few queries that show you the different forms of each query type with a commentary that illustrates any quirks of syntax, variant forms and what the purpose of the query was. Each of these query types share a lot of common features. In most cases, I’ll introduce everything in the SELECT form, since that is probably the most frequently used type of query.

The SELECT form of query is used to form standard queries. Its results come back in the standard SPARQL XML result format. Most of the example queries covered in this section require the use of SELECT queries. ASK is used to get yes/no answers without providing specifics, and is shown later in Listing 33. DESCRIBE, which is used to extract portions of the ontology as well as the instance data, and CONSTRUCT, which generates RDF based on the results of graphs that you query for, are not as commonly used, however, you will find some examples of these queries in the sample code found in Downloads.

To save space, this tutorial follows a rule of removing all but the first two results unless you really need to see everything. Most of the queries will have many more than two results. You’re probably not alone if you hate those manuals and books where they repeat themselves or pad out the book. This tutorial tries to avoid that wherever possible. The one area that the tutorial does repeat is in the prefixes of the example queries. The queries should be instantly usable, so they need to be self contained. You should be able to drag them onto a query form and just have them work.
Breaking down a query

The query in Listing 22 gets all of the notes and returns them ordered by date. In this section you will take a look at the syntax of a typical SPARQL query. In the following section you'll take a look at the algorithm that is used by a triple store to find matches to the query that you sent it.

Listing 22. A query to get all notes, ordered by date

```sparql
PREFIX : <http://aabs.purl.org/ont/journal#>
SELECT ?notes
WHERE {
  ?e a :JournalEntry .
  ?e :date ?date .
}
ORDER BY ?date
```

Each SPARQL SELECT query consists of a set of parts in order. It starts with a prologue that contains an optional BASE definition followed by a set of prefix definitions. It then contains a SELECT part that consists of SELECT followed by an optional dataset part that describes which graph to search in, followed by a WHERE clause that contains the graph patterns that describe the desired results. After the WHERE clause comes a set of solution modifiers: an Order clause, a Limit clause, or an Offset clause. I will introduce each of these modifiers later on.

Listing 23 shows the results.

Listing 23. Results from query in Listing 22

```plaintext
notes
"Today I wrote some more content for the great new SPARQL tutorial ..."
"Today I learnt about insane asylums"
```

To reverse the order by which the results are returned, you can choose to modify the ordering through the order modifiers `ASC()` or `DESC()`. Here, the results come in the opposite order:`ORDER BY DESC(?date).`

If you also want to order by another variable within the date ordering, you can add another variable to the ORDER BY clause:`ORDER BY DESC(?date) ASC(?notes).`

If you want to restrict the results to the top 5, you can use the LIMIT operator (see Listing 24).
Listing 24. Restricting the number of results to return

```sql
SELECT ?notes
WHERE
{
?e a :JournalEntry .
?e :date ?date .
}
ORDER BY ?date
LIMIT 5
```

And to skip some of the results before you take however many you plan to take, you use the OFFSET modifier (see Listing 25).

Listing 25. Skipping results

```sql
SELECT ?notes
WHERE
{
?e a :JournalEntry .
?e :date ?date .
}
ORDER BY ?date
LIMIT 5
OFFSET 150
```

These modifiers should be thoroughly familiar to you, if you've done any SQL development.

Searching a triple store

The process of getting results using a graph pattern is quite simple. Most triple stores maintain a store, either in memory or in a database, of triples that can be queried based on their subject, predicate, or object. SPARQL presents it with a set of triples in a query graph pattern. Assume, to begin, that just one triple is in the graph pattern. That query might provide a concrete URI for the subject. If so, the triple store can ignore all of the triples in the store that don't have that subject. From what's left, it can then filter out all of the predicates that don't match the predicate supplied in the graph pattern. Lastly, if SPARQL supplies a concrete object then it can be used to eliminate triples that it doesn't match.

If a variable is used in the subject, predicate, or object of the query, then rather than eliminate non-matching triples the triple store will reserve them all as potential matches for the variable. In the example above the first triple reads ?e a :JournalEntry. ..a is short for rdfs:type so the triple store can ignore all triples other than those with predicate rdfs:type. From what's left it can eliminate those without an object of :JournalEntry. What's left is a fund of possible triples
that can furnish it with results for \(?e\) as the subject.

The query above provided a graph pattern with multiple triples, so the triple store needs to follow the same procedure with the other triples before it can then try to marry up the variables. If a variable is found in multiple places then it can use all those triples where identical values intersect. In the example above, all matching triples must have a matching subject. If triples don't fit this scheme, they will be discarded and the result set is all the triples that remain. There might be multiple ways to match up the variables and, if so, then you will get multiple results. The last step is for the triple store to create a result set based on those variables that were required in the result set.

At the end of the search the triple store will have a result set containing (\(?e, ?notes, ?date\)) since those were all the variables defined in the query. If the SELECT query was presented like this "SELECT ?date ?notes" then \(?e\) would not be returned, even though it was vital for the query.

The procedure described above is the abstract form of what happens. You can speed it up in many ways (for instance by doing multiple matches in a single pass). They will all be equivalent in terms of the job they have to do.

Get a list of all bookings in date order

The query in Listing 22 allows you to create a simple micro-blog with each of the notes displayed in date order. That's about all there is to twitter. This query follows the same form, allowing you to get a complete listing of all the bookings that you have in date order. It's a necessary query to allow you to filter out the employees who are booked, as you try to construct a team for a new project.

In Listing 26 you must join data from several classes: EmployeeBooking, Customer, and User. SQL has a pretty cumbersome syntax to represent the joins between two tables. SPARQL thankfully does not. It's totally geared towards doing these kinds of joins, so it's easy. You need to define a graph matching pattern that defines all of the properties from each of the classes that you need to get back. A join in the world of triples is just another triple. Listing 26 provides the query.

Listing 26. Get all bookings in date order

```sparql
PREFIX u: <http://aabs.purl.org/ont/users#>
PREFIX j: <http://aabs.purl.org/ont/journal#>
PREFIX b: <http://aabs.purl.org/ont/avail#>

SELECT ?dn ?custName ?startDate ?endDate
WHERE
{
?booking a b:EmployeeBooking ;
b:startDate ?startDate ;
b:endDate ?endDate ;
}
```

Understanding SPARQL
Page 20 of 30 © Copyright IBM Corporation 1994, 2008. All rights reserved.
To start, define three namespace prefixes: u, j, b for users, journals and bookings. Then you tell Joseki that you are interested in matches for variables ?dn, ?custName, ?startDate and ?endDate. ?dn is just a shorter way to write 'display name' and that’s what it will contain after the query is run.

In the basic graph pattern you define a booking (called ?booking) and its start and end date (called ?startDate and ?endDate). What you are saying is that there are some instances defined in the graph of type b:EmployeeBooking. Whatever matches it finds of type b:EmployeeBooking will have properties of type b:startDate and b:endDate. Whatever matches those can be put into the variables ?startDate and ?endDate.

The query results definition states that you must get the u:displayName of the employee and the b:name of the customer they were with. The query has to define who or what those properties are to get the right matches. You therefore define the ?emp and ?cust instances and their properties. The query only needs to link them together to introduce the triples that link the instances of the booking, user and customer classes together. EmployeeBooking has an b:employee object property to link it to a specific user instance. It also had a b:with property that links to the Customer class.

By adding the 'b:with ?cust' triple (remember the subject is still ?booking) you tell SPARQL that you only want results for ?cust that match the result returned for ?booking (see Table 1). That's a join in SPARQL. The same applies to the :User class.

<table>
<thead>
<tr>
<th>dn</th>
<th>custName</th>
<th>startDate</th>
<th>endDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andrew Matthews&quot;</td>
<td>&quot;IBM&quot;</td>
<td>&quot;2008-03-01T09:00:00&quot;</td>
<td>&quot;2008-03-08T09:00:00&quot;</td>
</tr>
<tr>
<td>&quot;John Connor&quot;</td>
<td>&quot;IBM&quot;</td>
<td>&quot;2008-03-01T09:00:00&quot;</td>
<td>&quot;2008-03-08T09:00:00&quot;</td>
</tr>
</tbody>
</table>

Notice how easy the join was? You just state that there is a relationship between the instances of the two classes, and SPARQL only returns matches where that relationship existed.
Get all users (employees)

This query is again fairly similar to those that have gone before. The query in Listing 27 uses a default prefix since it’s only dealing with URIs from a single ontology. Because you can assume the URI for the class and the object properties, you can simply prefix them with colon (:).

Listing 27. Query using a default prefix

```
PREFIX : <http://aabs.purl.org/ont/users#>

SELECT DISTINCT ?dl ?dn
WHERE {
  ?u a :User ;
  :domainLogin ?dl ;
  :displayName ?dn .
}
ORDER BY ?dn
```

Table 2 shows the results of the query in Listing 27.

<table>
<thead>
<tr>
<th>dl</th>
<th>dn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;someDomain/andrew.matthews&quot;</td>
<td>&quot;Andrew Matthews&quot;</td>
</tr>
<tr>
<td>&quot;someDomain/john.connor&quot;</td>
<td>&quot;John Connor&quot;</td>
</tr>
<tr>
<td>&quot;someDomain/john.doe&quot;</td>
<td>&quot;John Doe&quot;</td>
</tr>
<tr>
<td>&quot;someDomain/sarah.connor&quot;</td>
<td>&quot;Sarah Connor&quot;</td>
</tr>
</tbody>
</table>

Get notes for all journal entries filtered by keyword

As a service to other employees in the company you might aggregate the journal entries based on the tags that the authors attached to the journal entries. That allows you to provide a stream devoted just to RDF, say, or just to Java development. Interested users who want to see what RDF-related work is under way, can subscribe to that stream and get a tailored feed.

This query is interesting because it introduces the FILTER keyword. FILTER provides you with an expression-based way to define properties of matching variables. In this query, you will use it to indicate that you are only interested in notes that have the word SPARQL in them.

The query to get all journal entries in Listing 28 just gets the notes terms—you don’t want to be swamped with text. This is the first query where you use the FILTER keyword. Filter provides a non-graph oriented method to define the properties of a
matching result. The example below uses a regular expression to match the entries that contain the word SPARQL.

**Listing 28. Get all journal entries that have the word 'today' in the notes of the entry**

```
PREFIX : <http://aabs.purl.org/ont/journal#>
SELECT ?notes
WHERE {
  ?e a :JournalEntry .
  FILTER regex(?notes, "today")
}
ORDER BY ?date
```

The query makes use of a default namespace prefix colon (:). The query doesn't refer to entities defined in more than one ontology, so there's no confusion over the meaning of predicates and types. The FILTER expression uses a regular expression to indicate that you want journal entries that contain the word SPARQL. There are two results shown in Listing 29.

**Listing 29. Two results**

```
notes
"Went to work on the SPARQL tutorial today. I seem to have a terminator fixation."

"Today I wrote some more content for the great new SPARQL tutorial that I've been preparing. I used some Turtle, and I defined a simple ontology for defining journal entries. This is an example of one of those entries!"
```

Regex is amongst a number of functions and operators that can be used. It was taken from the XPath and XQuery systems. FILTER expressions support the usual syntax of infix and prefix operators and parentheses, so the following from Listing 30 are also valid.

**Listing 30. More examples of using FILTER to restrict graph matches**

```
FILTER (?t = "RDF" || ?t = "OWL" || ?t = "cash")
FILTER (?start > "2008-03-05T09:00:00"^^xsd:dateTime &&
  ?start < "2008-03-07T09:00:00"^^xsd:dateTime) ||
  (?start < "2008-03-05T09:00:00"^^xsd:dateTime &&
  ?end > "2008-03-05T09:00:00"^^xsd:dateTime)
```

Understanding SPARQL
© Copyright IBM Corporation 1994, 2008. All rights reserved.
You should consult the SPARQL documentation for the full list of operators, functions, and type compatibilities.

Get all users who have required skills

When you form a team (for the tutorial's example), you will need a list of people who have skills in the technologies that you'll use. To search for them you need a way to provide all of the skills or technologies that you seek and to find out who matches against each of the skills. This query is of interest because it introduces the UNION operator. UNION allows you to merge the results of separate queries together. You'll use it to merge individual queries for each technology together.

This example gets the display name for all users who have written journal entries tagged with RDF, OWL, or SPARQL. The UNION operator allows you to merge the results from alternate graph patterns.

Generally, all graph patterns have to match before a result is returned. With UNION, a result from any of the sub-graph patterns is sufficient. In the query in Listing 31, any of {?e j:tag "RDF".}, {?e j:tag "OWL".}, or {?e j:tag "SPARQL".} can match in order for a result to be accepted. That's the equivalent of an OR operator, and shortly you'll see how the || operator can achieve the same effect.

Listing 31. Get all users with matching skills

```sparql
PREFIX u: <http://aabs.purl.org/ont/users#>
PREFIX j: <http://aabs.purl.org/ont/journal#>

SELECT DISTINCT ?dn
WHERE
{
?u a u:User;
    u:displayName ?dn .
?e a j:JournalEntry;
    j:user ?u .
    
    { ?e j:tag "RDF". } UNION
    { ?e j:tag "OWL". } UNION
    { ?e j:tag "SPARQL". }
}
```

As usual, you define the namespaces of the ontologies that you work with, and the variables that you're interested in. Then you define the properties of the users that you want to match. In this case, you define two instances that you want to join (?u for users and ?e for journal entries). To join the two, you describe the j:user property on the journal entry. After the instances and their relationships are defined, another graph pattern is introduced that merges together matches against tags attached to the journal entries (see Table 3).
Another way to represent the same query is through the use of a FILTER.

The simple query in Listing 31 joins journal instances to instances of the user class. This query (in Listing 32) introduces a string literal as the object of the triple pattern `<e j:tag "RDF">`. Its meaning ought to be fairly clear, but if you haven’t looked at the introduction to RDF and Turtle at the beginning of this tutorial, now might be a good time.

Listing 32. This query uses a FILTER to achieve the same results as the UNIONS in Listing 31

```
PREFIX u: <http://aabs.purl.org/ont/users#>
PREFIX j: <http://aabs.purl.org/ont/journal#

SELECT DISTINCT ?dn
WHERE
{ ?u a u:User;
   u:displayName ?dn .
?e a j:JournalEntry;
   j:user ?u ;
   j:tag ?t.
   FILTER (?t = "RDF" || ?t = "OWL" || ?t = "SPARQL")
}
```

The logical meaning of the filter is identical to the UNION based query above, but might be easier to understand if you're not yet familiar with the meaning of multiple graph patterns.

Which employees start with a client on a given date

For many queries, you need to get an overview of the data, or some outline answer that gives you a yes/no answer to a broad question, such as "Do we have someone working at X this week?" SPARQL can cater to those kinds of questions using the ASK query type.

The query in Listing 33 query asks whether anyone is starting at IBM on the 18th of March. You don't care who it is, you just want to know.

Listing 33. ASK whether someone will start with IBM on March 18th

```
PREFIX b: <http://aabs.purl.org/ont/avail#
```
The ASK query helps by returning only whether it found results, thus sparing you bandwidth on needless return data. The format of the query is similar to SELECT. You don't, however, define return variables since no variables are returned. Instead the query will return a true or false to indicate whether the query (if it had been a SELECT) would have returned data (see Listing 34).

Listing 34. ASK query returns true

ASK => true

Using these queries as examples, you should now have a solid grounding in SPARQL. Although I didn't have room to cover them in depth, take a look at the sample code in Download for queries that cover various ways to export data from your journaling example.

Section 4. Summary

Wrapup

You explored how the Semantic Web works using RDF, OWL and Turtle. You know the easiest ways to write RDF by hand with Turtle and you saw how SPARQL uses Turtle. You are now fully equipped to read the various semantic Web standards published by the W3C and others. You can author your own ontologies, host them, and query them using SPARQL. The technologies underpinning the Semantic Web are complex and you haven't seen them all here. However, you are now armed with enough knowledge to explore them with confidence.

The queries you saw cover many, but not all, of the capabilities of SPARQL. They
allowed you to effortlessly join classes and filter the results using a sophisticated expression language. You constructed RDF on the fly to maximize the value and interoperability of the data that you stored. You wrote queries that dispensed with needless network bandwidth, and others that provided you with the metadata to define the information in your triple store. You also, hopefully, got a sense of the depth and breadth of the capabilities of SPARQL.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example source code</td>
<td>x-sparql.zip</td>
<td>12KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

Information about download methods
Resources

Learn

- **The RDF Primer from W3C**: Get an indispensable guide to representing knowledge using RDF.
- **The SPARQL specification from W3C**: Read the specification of the SPARQL query language.
- **The Turtle specification from W3C**: Get a detailed description of the syntax and capabilities of Turtle.
- **The W3C Semantic Web portal from W3C**: Find a broad view of the technologies being developed at the W3C that relate the semantic Web.
- **The W3C Web architecture document from W3C**: See a good source for best practice relating to use of URIs.
- **302 Semantic Web Videos** (James Simmons, February 2008): Browse hundreds of hours of insight into the current state of the art, research directions, and commentary. Well worth a look.
- **PlanetRDF**: Browse a good aggregation of semantic Web-related blog posts.
- **The Wandering Glitch**: Read Semantic Web blog posts by Andrew Matthews, the author.
- **Introduction to Jena** (Philip McCarthy, developerWorks, June 2004): Check out a guide to using Jena.
- **Search RDF data with SPARQL** (Philip McCarthy, developerWorks, May 2005): Read an intermediate level guide to SPARQL.
- **An introduction to RDF** (Uche Ogbuji, developerWorks, December 2000) Read another piece providing an overview of some RDF basics.
- **W3C's OWL Overview**: Visit the best place to get an understanding of what OWL can do.
- **FOAF-Project**: Learn about a standard RDF ontology for describing people and their relationships.
- **IBM XML certification**: Find out how you can become an IBM-Certified Developer in XML and related technologies.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
• **developerWorks technical events and webcasts**: Stay current with technology in these sessions.

• **developerWorks XML zone**: Learn all about XML.

• **developerWorks podcasts**: Listen to interesting interviews and discussions for software developers.

**Get products and technologies**

• **Joseki Web site**: Visit the home site of the Joseki server.

• **IBM trial software**: Build your next development project with trial software available for download directly from developerWorks.

**Discuss**

• **Participate in the discussion forum for this content**.

• **XML zone discussion forums**: Participate in any of several XML-related discussions.

• **developerWorks XML zone**: Share your thoughts: After you read this article, post your comments and thoughts in this forum. The XML zone editors moderate the forum and welcome your input.

• **developerWorks blogs**: Check out these blogs and get involved in the developerWorks community.

**About the author**

Andrew Matthews

Andrew Matthews is a British Architect and Developer specializing in .NET technologies based in Melbourne, Australia. He’s been in the software development industry since the mid-1990s and initially got into computers as a child because of a magazine article about AI that explained it as a test-bed for philosophical ideas. Hardly surprising that he is so excited by the beauty, power, and promise of the Semantic Web. You can find out more about Andrew and read more about the semantic Web and other topics at his blog The Wandering Glitch.