The ultimate mashup -- Web services and the semantic Web, Part 3: Understand RDF and RDFs

Build and interpret Resource Description Framework (RDF) and RDF Schema Language (RDFs)

Skill Level: Intermediate

Nicholas Chase (ibmquestions@nicholaschase.com)
Freelance writer
Backstop Media

19 Sep 2006

Updated 08 Mar 2007

The power of the ultimate mashup is the intelligence you'll build into it by using semantic Web techniques, specifically the Web Ontology Language (OWL). But before you can tackle OWL, you want to be familiar with its base language, the Resource Description Framework (RDF) and RDF Schema Language (RDFs). This tutorial gives you a good background in both RDF and RDFs so you'll be ready to build ontologies for your Web services, and also able to make use of RDF's power with other projects as well.

Section 1. Before you start

This tutorial is for developers who are interested in learning about the Resource Description Framework (RDF), as well as for those interested in learning more about the semantic Web and semantic Web services in general and the building of ontologies in particular. You will learn how to build and interpret RDF information in both its familiar XML form and its non-XML shorthand form.

This tutorial does not involve any programming, but assumes that you are familiar

with the general concepts behind XML.

About this series

It seems you can't turn around on the Web these days without running into a Web site that either offers access to its data through a Web-services-based API or uses data from another site obtained through a Web-services-based API. When you consider the advantage of using existing information in your own applications, that's probably not terribly surprising. It was also just a matter of time before someone started to combine the data from these disparate systems to create something entirely new. These applications, called mashups, are the latest rage on the Web, from community-based sites to specialized search sites to the ever-present mapping mashups.

Mashups are (almost) all useful, but one thing that they have in common is that they were all developed for a specific set of services. If one of those services changes, or if the preference for a specific service of a particular type changes, you'll have lots of work to do.

The purpose of this tutorial series is to create a mashup application so smart that users can literally add and remove services at will, and the system will know what to do with them. The series progresses as follows:

In Part 1, I introduce the concept of mashups, showing how they work and building a simple version of one. You also discover serious performance problems involved in making potentially dozens of Web calls.

In Part 2, you solve some of that problem by using DB2®'s new pureXML™ capabilities to build an XML cache, which saves the results of previous requests and also enables you to retrieve specific information.

Ultimately, you will need to use *ontologies*, or vocabularies that define concepts and their relationships, so here in Part 3 you start that process by learning about RDF and RDFS, two key ingredients in the Web Ontology Language (OWL), which I discuss in Part 4. In Part 5, you take the ontologies created in Part 4 and use them to enable users to change out information sources.

In Part 6, things get really fun. At this point, you have a working application and the framework in place so that the system can use semantic reasoning to understand the services at its disposal. In this part, you give the user control, enabling him or her to pick and choose the data that is used for a custom mashup.

About this tutorial

In previous parts of this series, you created the foundation for the mashup: the servlet that checks the database cache and then displays the saved or fresh data. Now you need to start putting the "semantic" in semantic Web. In Part 4, you'll create an ontology that will enable you to perform logic on your services, but first you need to understand the language in which you will work with that ontology, the Resource Description Framework, or RDF.

This tutorial gets you up to speed with RDF and its offshoot, RDF Schema (RDFs). In the course of this tutorial, you will learn:

- What RDF is and what it's used for
- The relationships between RDF, RDF Schema, OWL, and the semantic Web
- The basics of RDF
- Dealing with resources, properties, and other RDF structures
- Representing RDF in XML -- and without it
- Creating classes and instances using RDF Schema

In this tutorial, you will look at these structures from the standpoint of representing the components and data from your mashup.

Prerequisites

Although you do need various software components to run the mashup itself -- see Part 2 for the list of requirements for the current state of the application (see Resources)-- this tutorial is mostly conceptual, so you will need no special software to follow along.

Section 2. A quick overview

Before you get too far along, let's discuss the mashup example application, the semantic Web, RDF and ontologies.

The story so far

A *mashup* is an application that takes data -- usually Web service data, and usually

from more than one source -- and uses it to create something new. So far, in your quest to create the ultimate mashup, you created a system that displays information for an arbitrary number of Web services and displays it on a Web page.

The application is built to be as generic as possible. You can define the services in their own class, with an array of services to display. To add to or remove services from the final output, you can manipulate the contents of the array, and an XML template that specifies the ultimate display accompanies each service definition.

The end result is that you can control the services used, the appearance of their data on the final Web page, and even subservices linked to the main service by simply manipulating what amounts to input instructions.

The reason I made things so generic is partly to achieve flexibility and maintainability, and partly to prepare the application for what comes next.

And what comes next is a new way to find information.

The coming semantic Web

Right now, when you go to a search engine such as Google and enter a search term, it's hard to predict exactly what you're going to get back. If you enter the term SOAP, are you going to get back a selection of material on Web services, or advice on cleaning your laundry? Or an update on who killed the character Greg Madden on the television soap opera, "All My Children"?

Google does a lot of work finding out which category a particular Web page belongs to, by looking at what's linked to what, but in the long run, the string "SOAP" is just a string of letters, and discerning its meaning and context is a task currently best left to the human brain.

But things are different on the semantic Web. On the semantic Web, information is identified in a way that is understandable to machines, and allows software to accomplish this processing.

The example that is frequently used for this discussion involves travel arrangements or other appointments. On the semantic Web, an intelligent agent can check your schedule, find you a proper specialist with an appointment you can make, get ratings from current and former patients, and arrange for a babysitter to take care of your kids while you go.

All of this is possible only if the information published on the semantic Web is identified in a machine-readable -- and understandable -- way.

But here's the trick. The semantic Web isn't some new playground for which users must abandon the current Web. No, the semantic Web is just the current Web, but

with more information.

Semantic Web services

If you label the Web services available to the mashup and their data in this way, with more information, you can enable the application to make intelligent choices. For example, the application is able to understand which services represent mapping information, and which represent online stores, or even further, online bookstores. It would know which information from those services represent the title, description, price, and so on.

That's the promise of semantic Web services; a system that understands what the underlying data represents, making it possible to use it in specific ways.

In this tutorial, it means your users can automatically swap out services of the same type -- using Barnes and Noble instead of Amazon, for example -- or even create new mashups of their own.

The structure in which you identify this information is called an ontology.

Ontologies

In their book, Website Indexing 2nd Edition, Glenda Browne and John Jermey define the word *ontology* this way:

> "**Ontology**: specification of a conceptualisation of a knowledge domain. An ontology is a controlled vocabulary that describes objects and the relations between them in a formal way, and has a grammar for using the vocabulary terms to express something meaningful within a specified domain of interest. The vocabulary is used to make queries and assertions. Ontological commitments are agreements to use the vocabulary in a consistent way for knowledge sharing. Ontologies can include glossaries, taxonomies and thesauri, but normally have greater expressivity and stricter rules than these tools. A formal ontology is a controlled vocabulary expressed in an ontology representation language."

In other words, an ontology is an agreed-upon way to communicate specific ideas. So, if I wanted to make information about my blog available, I might publish something like this (see Listing 1):

Listing 1. Information encoded using the Dublin Core

```
<rdf:RDF
```

```
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/">

    <rdf:Description rdf:about="http://www.chaosmagnet.com/blog">

       <b><dc:creator>Nicholas Chase</dc:creator>
       <dc:title>Chaos Magnet</dc:title>
       <dc:description>The personal and professional ramblings
                         of technology author Nicholas Chase</dc:description>
       <dc:date>2006-06-30</dc:date></b>

    </rdf:Description>
</rdf:RDF>
```

This information is represented using terms from a structure called the *Dublin Core*. Whether or not you can call the Dublin Core an ontology is up for debate, but what is certain is that everyone agrees on the semantic meanings of creator, title, date, and so on.

In this case, you might create an ontology that specifies concepts such as:

- Service
- Map
- Store
- Bookstore
- Price
- Comment
- Image
- Thumbnail
- Title

You can then specify individual services or data as "instances" of one or another of these concepts. To do that, you'll use the Web Ontology Language, or OWL.

Web Ontology Language (OWL)

Yes, yes, I know. The acronym for Web Ontology Language should be WOL, not OWL. But it's not, for reasons that range from references to Winnie the Pooh (and the wise Owl, who spelled his name W-O-L) to a tribute to early ontology projects such as Bill Martin's One World Language to the fact that it's just easier to say and to design logos for "OWL" than it is for "WOL".

All right, so now that is out of the way, what exactly is it?

Web Ontology Language, or OWL, provides a vocabulary for specifying information on the Web in such a way that machines can interpret it. Consider this example (see Listing 2):

Listing 2. A (very) simple ontology

```
<rdf:RDF
    xmlns     ="http://www.example.com/mashup#"
    xml:base  ="http://www.example.com/mashup#"
    xmlns:owl ="http://www.w3.org/2002/07/owl#"
    xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<owl:Class rdf:ID="Store">
   <rdfs:label>Online Store</rdfs:label>
</owl:Class>

<owl:Class rdf:ID="Bookstore">
   <rdfs:subClassOf rdf:resource="#Store"/>
   <rdfs:label>Bookstore</rdfs:label>
</owl:Class>

<Bookstore rdf:ID="Amazon.com"/>
</rdf:RDF>
```

The code in Listing 2 creates a main type of object, a `Store`, and a subclass of that type, the `Bookstore`. You can then use that new type to specify that Amazon.com is a `Bookstore`.

Not terribly exciting, until you consider that OWL also enables you to make various assertions about `Bookstores`, such as the fact that they're accessible online, they sell `Books` (which you can define as your own type), that they might also sell other specific types of things, and so on. By specifying Amazon.com as a `Bookstore`, all of that information automatically gets associated with it.

You'll learn all about OWL in Part 4 of this series, but before you get there, you need to address the form of the information, the Resource Description Framework.

Resource Description Framework (RDF)

The Resource Description Framework is an unambiguous way to state information about something. Many people consider it to be overly complicated, but when you come right down to it, RDF is just a way to specify the properties of resources. Going back to the blog example (see Listing 3):

Listing 3. Specifying properties of resources

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/">
```

```
   <rdf:Description rdf:about="http://www.chaosmagnet.com/blog">

      <dc:creator>Nicholas Chase</dc:creator>
      <dc:title>Chaos Magnet</dc:title>
      <dc:description>The personal and professional ramblings
                       of technology author Nicholas Chase</dc:description>
      <dc:date>2006-06-30</dc:date>

   </rdf:Description>
</rdf:RDF>
```

In Listing 3, the code creates a `Description` that is about a resource, http://www.chaosmagnet.com/blog. The code specifies four properties, `dc:creator`, `dc:title`, `dc:description`, and `dc:date`, and the values of those properties. In this tutorial, I'll talk about graphs and triples and resources and all kinds of things that can make RDF seem overly complicated, but in the end, this is really all it is: assigning properties to resources in an agreed-upon way.

Resource description Framework Schema (RDFs)

As I said before, RDF is a way to assign properties to resources, and that's it. RDF itself doesn't even assign any meaning to those properties. Any meaning in a straight RDF document comes from the properties themselves, and not from the RDF. Unfortunately, that means that by itself, RDF isn't very useful for tasks such as defining vocabularies, in which you need a way to determine at least the basic relationships between concepts.

Enter RDF Schema. Part of the group of specifications thought of as "The RDF Recommendation", RDF Schema provides a way to create the basic relationships that define a vocabulary. Consider this example (see Listing 4):

Listing 4. A simple RDF Schema example

```


<rdf:RDF
    xmlns     ="http://www.example.com/mashup#"
    xml:base  ="http://www.example.com/mashup#"
    xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Service">
   <rdfs:label>Web Service</rdfs:label>
</rdfs:Class>

<rdfs:Class rdf:ID="Store">
   <rdfs:subClassOf rdf:resource="#Service"/>
   <rdfs:label>Online Store</rdfs:label>
</rdfs:Class>

<rdfs:Class rdf:ID="Bookstore">
   <rdfs:subClassOf rdf:resource="#Store"/>
   <rdfs:label>Bookstore</rdfs:label>
</rdfs:Class>

<rdf:Property  rdf:ID="endpoint">
```

```
   <rdfs:domain rdf:resource="#Service"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#anyURI"/>
</rdf:Property>

<Bookstore rdf:ID="Amazon.com">
   <endpoint>
      http://soap.amazon.com/onca/soap?Service=AWSECommerceService
   </endpoint>
</Bookstore>
</rdf:RDF>
```

In Listing 4, the code creates three Classes and a Property, `endpoint`. (I'll leave OWL out of the discussion until Part 4.) You can assign that property to any `Service` (which means it's applicable to `Bookstore`, as a subclass of `Store`, a subclass of `Service`) and it must have as a value an `anyURI` value as defined by the XML Schema recommendation.

RDF Schema provides these concepts as a set of basic building blocks you can use to build new languages and ontologies. In Part 4, we'll see how RDFs concepts go into making OWL, but first, in this tutorial, you'll find out how it works.

Let's start with RDF itself.

Section 3. RDF basics

I have seen programmers with years of experience practically burst into tears at the prospect of getting a handle on RDF, but it's really not that difficult. In this section, you'll learn the basics so that you too can say, "Get over it, it's not that difficult."

RDF resources

The very name Resource Description Framework clues you in to the fact that you're describing resources, but just what are they?

A resource can be just about anything, whether it's tangible (such as a person) or intangible, such as a job title or opportunity. The important thing is that you must be able to reference it using a Uniform Resource Identifier, or URI. One type of URI is the Uniform Resource Locator, or URL, that you're used to seeing on the Internet, such as http://www.ibm.com or mailto:ibmquestions@nicholaschase.com. But URIs can also be more general, such as urn:backstopmedia, which is a Uniform Resource Name (URN) instead of a URL.

That means that you can use URLs to identify resources such as Web pages and

e-mail boxes -- resources for which a URL can provide information on finding it -- and URNs to identify other resources. For example, you might decide to identify a person (at least in the United States) with a URN referencing their Social Security Number, as in ssn:078-05-1120. Of course, in this age of identity theft you likely wouldn't want to do that, so you often see people referenced as a URL with some agreed-upon meaning, such as their home page (http://www.nicholaschase.com) or internally consistent URL (such as http://www.example.com/employees/empto1138.html).

The important thing is that you must be able to identify the resource with some sort of URI. The form it ultimately takes isn't important.

RDF properties

To describe a resource using RDF, assign it properties. You can phrase these property assignments in sentences. For example (see Listing 5):

Listing 5. Assigning properties

```
Nick's blog has a name of Chaos Magnet
Nick's blog has a creator of Nicholas Chase
```

In each of these sentences, you have a subject (Nick's blog) a predicate (has a name, has a creator) and an object (Chaos Magnet, Nicholas Chase). This subject represents the resource, the predicate the property, and the object the value of the property. So if you convert these sentences into something close to an RDF triple, you get what is shown in Listing 6:

Listing 6. Breaking things up

```
Nick's blog           name             Chaos Magnet    .
Nick's blog           creator          Nicholas Chase  .
```

Of course, I represent the resource as a URI, so instead it would be something like (see Listing 7):

Listing 7. Representing resources as URIs

```
http://www.chaosmagnet.com     name      Chaos Magnet      .
http://www.chaosmagnet.com     creator   Nicholas Chase    .
```

This still isn't actually an RDF triple because it has plain strings for the properties. The problem here is that properties, like resources, must be represented as a URI.

URI references

Actually, you represent properties as a URI reference, or URIref, which consists of the base for the property -- in a very real sense, the namespace to which the property belongs -- followed by a fragment.

For example, in the previous example, Listing 7, I referred to the `creator` property. In actuality, this is the `creator` property as defined by the Dublin Core namespace, so its full URIref would be (see Listing 8):

Listing 8. Full URIref for creator

```
http://purl.org/dc/elements/1.1/#creator
```

If you're a Web developer, that syntax at the end, the fragment that starts with the pound sign (#) might look familiar. It's the same syntax you use to send the browser to a particular spot on a page, such as (see Listing 9):

Listing 9. URL, including fragment

```
https://www6.software.ibm.com/developerworks/education/ws-understand-
web-services3/#N1014F
```

To create that spot on the page, you can create an anchor tag of (see Listing 10):

Listing 10. Creating an anchor tag

```
<a name="N1014F">:</a>
```

With the advent of XHTML, the XML version of HTML, this changes to (see Listing 11):

Listing 11. XHTML anchor tags

```
<a id="N1014F" />
```

So the fragment that refers to that element is a pound sign and then the value of the ID attribute. (For those familiar with XPointer, this might have already been obvious.) Later, when you create resources using the rdf:ID attribute, you'll see this again, but for now my point is that a URIref is a URL, probably with a fragment attached.

It's also the only acceptable way of specifying an RDF property. So the sentences

become (see Listing 12):

Listing 12. Properties as URIrefs

```
http://www.chaosmagnet.com  http://example.com#name  Chaos Magnet  .
http://www.chaosmagnet.com  http://purl.org/dc/elements/1.1/#creator
                                                Nicholas Chase    .
```

In some cases, you want to abbreviate all that, so you might assign a prefix to represent the actual namespace. That enables you to cut the sentences down to Listing 13:

Listing 13. Shortening the sentences

```
http://www.chaosmagnet.com  ex:name       Chaos Magnet      .
http://www.chaosmagnet.com  dc:creator    Nicholas Chase    .
```

This is closer to true RDF form, but not quite there yet.

Property values

So far all of the properties you've seen have simple string values, but in real life that's actually the minority of cases. In order to understand why, you need to take things literally. When I say "Nick's blog has a creator of 'Nicholas Chase'", that's actually a bit of a nonsense statement. A string of characters ("Nicholas Chase") didn't create my blog (even if it seems that way sometimes). What I really mean is "The resource known as 'Nick's blog' was created by the resource known as 'Nicholas Chase'."

So the actual triple would look something more like this (see Listing 14):

Listing 14. Getting closer to the triple

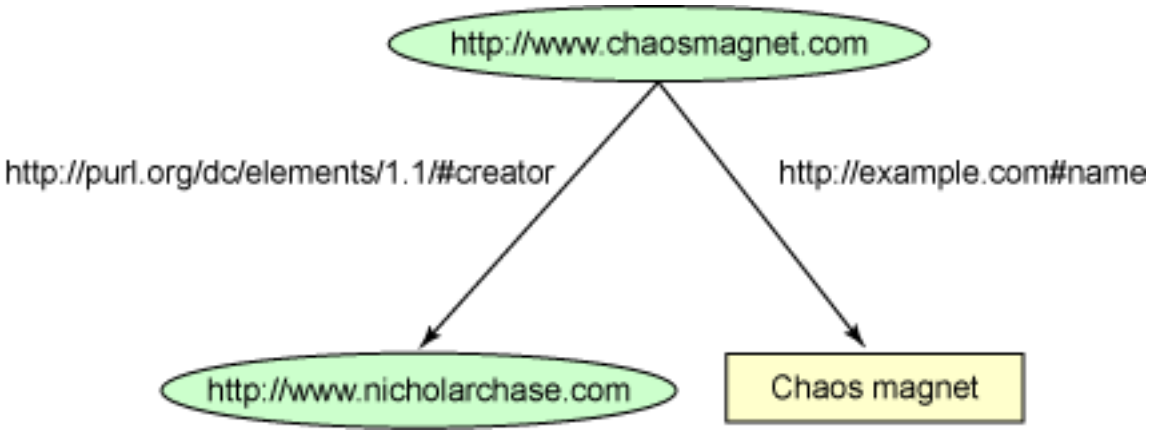
```
http://www.chaosmagnet.com  dc:creator  http://www.nicholaschase.com
```

You can represent this in an RDF graph.

RDF graphs

One way to represent all of this information is through the use of RDF graphs. In these graphs, you can see how things fit together. For example, see Figure 1:

Figure 1. An RDF graph



The oval nodes represent resources, with the arrows between them representing properties. Property values that are resources are shown as ovals, with literal values shown as boxes.

RDF notations

So far you've seen a small glimpse of RDF information, but you haven't taken a formal look at how to represent it. You can use triples, in which resource URIrefs are written out in their entirety (see Listing 15):

Listing 15. Full URIref triples

```
<http://www.chaosmagnet.com> <http://example.com#name> Chaos Magnet .
<http://www.chaosmagnet.com>
                  <http://purl.org/dc/elements/1.1/#creator>
                                    <http://www.nicholaschase.com> .
```

(Space constraints prohibit it here, but each triple is normally written on one line.)

You can also use shorthand, replacing the namespaces with prefixes (see Listing 16):

Listing 16. Shorthand notation

```
@prefix ex: <http://example.com> .
@prefix dc: <http://purl.org/dc/elements/1.1/#creator> .
<http://www.chaosmagnet.com>  ex:name         "Chaos Magnet" .
<http://www.chaosmagnet.com>  dc:creator
                              <http://www.nicholaschase.com> .
```

Notice that only the full URIrefs are represented in brackets. Perhaps the most familiar way of seeing RDF, however, is as XML. For example, you'd show these relationships as (see Listing 17):

Listing 17. XML notation

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ex = "http://example.com#"
   xmlns:dc="http://purl.org/dc/elements/1.1/#">

   <rdf:Description rdf:about="http://www.chaosmagnet.com">
      <ex:name>Chaos Magnet</ex:name>
      <dc:creator rdf:resource="http://www.nicholaschase.com" />
   </rdf:Description>
</rdf:RDF>
```

You'll see more about representing RDF in XML in the next section.

Section 4. RDF/XML

XML isn't the only way to represent RDF. In some cases, particularly when you try to visualize the relationships between resources and properties, it's not even the best way. But it's certainly the most popular way, and when it comes to the semantic Web, the most convenient way. Take a look at how this works in more detail.

Describing existing resources

The most commonly seen use of RDF is to describe resources that already exist, such as all or part of an RSS feed. In those cases, the RDF is typically included in another XML document and distinguished through the use of XML namespaces. To keep things simple, however, let's look at the RDF in isolation, so to speak.

I'll start with a simple document describing the data returned by a query to Technorati's "movies" tag (see Listing 18):

Listing 18. Describing an existing resource

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:tapi="http://www.technorati.com/tag#">

   <rdf:Description
rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results
for Movies</tapi:Title>
      <tapi:PubDate>Sun, 30 Jul 2006
11:38:42 PST</tapi:PubDate>
<tapi:Inboundlinks>321580</tapi:Inboundlinks>
```

```
   </rdf:Description>
</rdf:RDF>
```

Here is a simple `rdf:RDF` element, with a Description that contains properties for the resource described in the `rdf:about` attribute. In this case, the properties are from the namespace http://www.technorati.com/tag#, so if you expand the namespaces, we see that the `Title` property, for example, is really the `http://www.technorati.com/tag#Title` property.

In this way, you can add as many properties to the resource as you like.

Also note that this is not the only way to represent RDF in XML. Because you often include RDF in other documents, you want to do it so that they don't disturb the original information. The way I have presented the data here, however, is if I added this information to the browser, the text content of the elements will appear on the page. This is just as it should be, given the browser's default behavior of simply displaying the content of any tags it doesn't understand.

To prevent that, we can use an alternate method that places the information in attributes instead of elements (see Listing 19):

Listing 19. RDF data in elements

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies"
      tapi:Title="[Technorati] Tag results for Movies"
      tapi:PubDate="Sun, 30 Jul 2006 11:38:42 PST"
      tapi:Inboundlinks="321580" />

</rdf:RDF>
```

The information is the same, but now it's contained in attribute values rather than element content, and it won't be displayed by a browser.

Specifying data types

Occasionally, you want to be specific in how specific data should be treated, so it's helpful to be able to specify an intended data type. For example (see Listing 20):

Listing 20. Specifying data types

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#">
```

```
   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
            rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
      </tapi:PubDate>
      <tapi:Inboundlinks
         rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>

   </rdf:Description>
</rdf:RDF>
```

In this case, I specify that the `PubDate` and `Inboundlinks` elements should be the date and integer types as defined in the XML Schema specification, respectively. You can tell it's the XML Schema version of these types based on the namespaces in their URIrefs. There's nothing that says you have to use XML Schema datatypes, but they're handy and XML applications generally understand them already, so there's no reason not to use them.

You can also represent type in N3 notation, as in (see Listing 21):

Listing 21. Specifying data types in N3 notation

```
<http://www.technorati.com/tag/Movies>
       <http://www.technorati.com/tag#Inboundlinks>
            "321580"^^http://www.w3.org/2001/XMLSchema#integer .
```

Note that RDF itself does nothing to enforce or validate these data types. There's nothing to stop you from entering a string for a date or a number for a URL. RDF simply provides a way to denote the information; the RDF application is responsible for making sure the data conforms.

Referencing existing resources

Sometimes you not only describe existing resources, you use them to describe other resources. For example, you can take the feed that you describe and add an Image property to it (see Listing 22):

Listing 22. Referencing a resource as a property value

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
```

```
      <tapi:PubDate
            rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks
          rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>

      <tapi:Image rdf:resource
             "http://static.technorati.com/pix/logos/logo_sm.gif"/>

   </rdf:Description>
</rdf:RDF>
```

Notice that you specify the value as an `rdf:resource`, and use its URI as the actual property value. In some cases, however, the resource you want to reference isn't defined outside of the current namespace. For example, what if the image you reference isn't just a file you can reference with a single URL, but rather a collection of properties? You need a way to point to a definition of that resource.

To do that, create a second `Description` and assign it a `nodeID` (see Listing 23):

Listing 23. Using anonymous nodes

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:nodeID="image1"/>

   </rdf:Description>

   <rdf:Description rdf:nodeID="image1">
      <tapi:Url>
          http://static.technorati.com/pix/logos/logo_sm.gif
      </tapi:Url>
      <tapi:Title>Technorati logo</tapi:Title>
      <tapi:Link>http://www.technorati.com</tapi:Link>
   </rdf:Description>

</rdf:RDF>
```

Notice that you create a second `Description` element and assign it a `nodeID`. This enables you to create a resource that you can reference from the `tapi:Image` property. It's called a `nodeID`, because if you were to draw out the RDF graph for the data, the image1 node in Figure 1 above would be an empty oval node with its own properties.

Creating new resources

Using the `nodeID` creates a way to reference this node, but only within the context of this document and namespace. Sometimes, however, you want to create a resource that you can reference from anywhere, including other documents and other namespaces.

You can do that through the use of the `rdf:ID` attribute (see Listing 24):

Listing 24. Creating a new resource

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
              rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks
           rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>

   </rdf:Description>

   <rdf:Description rdf:ID="image1">
      <tapi:Url>
          http://static.technorati.com/pix/logos/logo_sm.gif
      </tapi:Url>
      <tapi:Title>Technorati logo</tapi:Title>
      <tapi:Link>http://www.technorati.com</tapi:Link>
   </rdf:Description>

</rdf:RDF>
```

Remember I talked about how, in an XML document, identifying an element with an ID attribute enables you to create a fragment by which to reference it? Well, here you see it in action. You created a new resource and gave it an ID of `image1`, which enables you to reference it with the relative URI `#image1`.

Now, note that I said it's a relative URI. The actual full URI for this resource is the base URI for the document followed by the fragment. For example, when I ran this document through the RDF validator, I got a URI of http://www.w3.org/RDF/Validator/run/1154295344338#image1.

Of course, that's not terribly predictable, so you're better off to specify the base URI directly using the `xml:base` attribute (see Listing 25):

Listing 25. Specifying a base URI

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">

      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
             rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
      </tapi:PubDate>
      <tapi:Inboundlinks
           rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>

   </rdf:Description>

   <rdf:Description rdf:ID="image1">
      <tapi:Url>
           http://static.technorati.com/pix/logos/logo_sm.gif
      </tapi:Url>
      <tapi:Title>Technorati logo</tapi:Title>
      <tapi:Link>http://www.technorati.com</tapi:Link>

   </rdf:Description>

</rdf:RDF>
```

Now you can say with certainty that no matter where the document is located, the full URI for the image resource is going to be http://www.technorati.com/tag#image1. This consistency enables you to reference this resource from anywhere, including other documents and other namespaces.

Note that the value of the `rdf:ID` attribute must be unique in a namespace, so you must define a specific resource in one element.

Using multiple descriptions

So far, every time you've created or described a resource, it's been within a single `Description` element, but that's by no means required. You can split out the description of your original feed into several different `Description` elements (see Listing 26):

Listing 26. Creating a resource with multiple Description elements

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
```

```
      <tapi:Image rdf:resource="#image1"/>
   </rdf:Description>

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:PubDate
             rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
        Sun, 30 Jul 2006 11:38:42 PST
       </tapi:PubDate>
   </rdf:Description>

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:Inboundlinks
          rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
        321580
      </tapi:Inboundlinks>
   </rdf:Description>

   <rdf:Description rdf:ID="image1">
      <tapi:Url>
          http://static.technorati.com/pix/logos/logo_sm.gif
      </tapi:Url>
      <tapi:Title>Technorati logo</tapi:Title>
      <tapi:Link>http://www.technorati.com</tapi:Link>
   </rdf:Description>

</rdf:RDF>
```

What's particularly interesting about this form of description is that there's no limit to where these `Description` elements must be. You could create a resource that includes information from all over the world, as long as everybody involved had the appropriate information.

Designating types

Once you start to build ontologies, you'll start to create types of resources. For example, you might create a Service type, a Price type, an Owner type, and so on, so that you can classify information from different services. Classifying this information involves setting a "type" for the data (see Listing 27):

Listing 27. Setting a date type

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#">

   <rdf:Description rdf:about="http://www.technorati.com/tag/Movies">
      <rdf:type rdf:resource="http://www.example.com/mashup/Feed" />
      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
            rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
        Sun, 30 Jul 2006 11:38:42 PST
       </tapi:PubDate>
      <tapi:Inboundlinks
          rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
        321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>
```

```
   </rdf:Description>
...
```

For the moment, this is just an arbitrary type; you'll look at defining types shortly, when you look at RDF Schema.

Alternate forms: types as element names

Once you determine that data is of a particular type, you have the option to express it in a way that's a bit cleaner than setting the `rdf:type` attribute. Instead, you can use the name of the type in place of the `Description` element (see Listing 28):

Listing 28. Using the type name as the element name

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#"
   xmlns:mashup="http://www.example.com/mashup/">

   <mashup:Feed rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
             rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks
          rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>
   </mashup:Feed>

   <rdf:Description rdf:ID="image1">
...
   </rdf:Description>

</rdf:RDF>
```

Here the code defines the `mashup:` prefix to represent the `http://www.example.com/mashup/` namespace. So ultimately, you've set the type for the data as http://www.example.com/mashup/Feed, just as in the previous example.

This form of expressing data types becomes crucial, and almost universal, once you start talking about OWL, and to a lesser extent, RDF Schema.

Bags, sequences, and alternatives

So far you've dealt with properties that have a single, distinct value, but if you've ever dealt with data in the real world, you know that's not always true. Most systems

of any usefulness have at the very least one or two one-to-many relationships. For example, the feed you're looking at has information about itself, but it also has a dozen or so item entities associated with it.

In RDF, you can express these relationships in a number of different ways. The simplest is to use a Bag (see Listing 29):

Listing 29. Expressing a one-to-many relationship as a Bag

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#"
   xmlns:mashup="http://www.example.com/mashup/">

   <mashup:Feed rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate
            rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks
           rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>

      <tapi:Items>
         <rdf:Bag>
            <rdf:li rdf:resource="http://www.robotjohnny.com/2006/07/30/haiku-
                  review-lady-in-the-water/"/>
            <rdf:li
                rdf:resource="http://riesje.livejournal.com/5359.html"/>
            <rdf:li
                rdf:resource="http://www.vanguardreport.com/phpnuke/index.php"/>
         </rdf:Bag>
      </tapi:Items>

   </mashup:Feed>
...
```

Internally, RDF numbers each one of the items inside the bag, but for convenience, it also provides the `rdf:li` element (modeled on HTML's list item).

An `rdf:Bag` is exactly what it sounds like, when you come right down to it. It's a bunch of items grouped together, in no particular order. Every item in the bag has the same importance.

Of course, sometimes that's not quite what you want. For example, Items are typically arranged in reverse chronological order; order matters. To express that, you can create a sequence instead (see Listing 30):

Listing 30. Creating a sequence

```
<rdf:RDF
```

```
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:tapi="http://www.technorati.com/tag#"
   xml:base="http://www.technorati.com/tag#"
   xmlns:mashup="http://www.example.com/mashup/">

   <mashup:Feed rdf:about="http://www.technorati.com/tag/Movies">
      <tapi:Title>[Technorati] Tag results for Movies</tapi:Title>
      <tapi:PubDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
         Sun, 30 Jul 2006 11:38:42 PST
        </tapi:PubDate>
      <tapi:Inboundlinks rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
         321580
      </tapi:Inboundlinks>
      <tapi:Image rdf:resource="#image1"/>

      <tapi:Items>
       <rdf:Seq>
         <rdf:li rdf:resource="http://www.robotjohnny.com/2006/07/30/haiku-
                   review-lady-in-the-water/"/>
         <rdf:li
             rdf:resource="http://riesje.livejournal.com/5359.html"/>
         <rdf:li
             rdf:resource="http://www.vanguardreport.com/phpnuke/index.php"/>
       </rdf:Seq>
      </tapi:Items>

   </mashup:Feed>
...
```

Functionally, an `rdf:Seq` is the same as an `rdf:Bag`, but the use of it implies that the order is important. Note that there's nothing about RDF itself that makes this distinction. RDF is simply an agreed-upon way to mark up information; the application itself needs to enforce the preserved order.

One more wrinkle that pops up with one-to-many relationships is a situation in which one and only one of a group applies. For example, you might have a set of featured items from which your application chooses one to display (see Listing 31):

Listing 31. Using rdf:Alt

```
...
            <rdf:li rdf:resource="http://www.vanguardreport.com/phpnuke/index.php"/>
         </rdf:Seq>
      </tapi:Items>


      <mashup:FeaturedItem>
         <rdf:Alt>
            <rdf:li rdf:resource="http://www.robotjohnny.com/2006/07/30/haiku-
                   review-lady-in-the-water/"/>
            <rdf:li
                rdf:resource="http://riesje.livejournal.com/5359.html"/>
            <rdf:li
                rdf:resource="http://www.vanguardreport.com/phpnuke/index.php"/>
         </rdf:Alt>
      </mashup:FeaturedItem>

   </mashup:Feed>
...
```

The syntax is the same as `rdf:Bag` and `rdf:Alt`, but in this case the application -- remember, it's the application's responsibility! -- must choose one and only one of the included items.

Collections

Now, remember how I said that you can describe a single resource using multiple `rdf:Description` elements? In that situation, all of the information gets aggregated together into a single resource. That means that if we define a Bag (or a Seq or an Alt) a second `rdf:Description` can add items to it. To prevent that, you can define a group as a Collection instead (see Listing 32):

Listing 32. Using Collections

```
...
         </rdf:Seq>
      </tapi:Items>


      <mashup:FeaturedItems rdf:parseType="Collection">
            <rdf:Description rdf:about="http://www.robotjohnny.com/2006/07/30/haiku-
                  review-lady-in-the-water/"/>
            <rdf:Description rdf:about=
                     "http://riesje.livejournal.com/5359.html"/>
            <rdf:Description rdf:about=
             "http://www.vanguardreport.com/phpnuke/index.php"/>
       </mashup:FeaturedItems>


   </mashup:Feed>
...
```

In this case, you define the specific resources that belong to the Collection and mark it with the `rdf:parseType` attribute.

RDF defines two other values for `parseType`. You can use the `Resource` value as another way to create blank nodes, just as you used the `NodeID` earlier, and you can use the `Literal` value to create an element that includes arbitrary XML information. For example, you might use it to enable a Summary to include XHTML for display, or to include the XML templates you used in Part 1 of this series (see Resources).

Section 5. RDF Schema

RDF describes how to specify information about a resource, but the information itself

has no meaning. To define types of items and their relationships -- such as classes and subclasses -- you need to make some additions to your vocabulary. Those additions are encapsulated in RDF Schema. RDF Schema (otherwise known as RDFs) has its own namespace and a number of predefined elements and attributes. In this section, I take a look at what you need to create classes and objects and their relationships.

Creating classes

The first step in defining a vocabulary is to create classes. Like classes in an object oriented language, RDFs classes provide a type of template from which you can create objects, or as they're called in RDF parlance, individuals. To start, create some of your service-related classes (see Listing 33):

Listing 33. Basic classes

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xml:base="http://www.example.com/mashup/">

  <rdf:Description rdf:ID="Service">
    <rdf:type
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:ID="SOAPService">
    <rdf:type
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:ID="RESTService">
    <rdf:type
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:ID="RSSFeed">
    <rdf:type
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

</rdf:RDF>
```

In this case, you created four classes. You know they're classes because you see they are marked with a type of `http://www.w3.org/2000/01/rdf-schema#Class`. (So actually you could use `rdfs:Class` rather than `rdf:Description`.)

Creating subclasses

Once you define classes, you can create their relationships. You can do that with the

`rdfs:subClassOf` property (see Listing 34):

Listing 34. Creating subclasses

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.example.com/mashup/">

      <rdf:Description rdf:ID="Service">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
      </rdf:Description>

      <rdf:Description rdf:ID="SOAPService">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RESTService">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RSSFeed">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#RESTService" />
      </rdf:Description>

</rdf:RDF>
```

Here you defined the `SOAPService` and `RESTService` classes as subclasses of `Service`, and `RSSFeed` as a subclass of `RESTService`. That means if you define an individual as an `RSSFeed`, you know that it's also a `RESTService` and a `Service`.

Creating an instance

All right, now you have the classes, so how do you create individuals, the equivalent of objects in an object-oriented language?

Do it the same way you created resources from scratch before, the difference here being that you specify types based on the classes you've already created (see Listing 35):

Listing 35. Creating individuals

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:mashup="http://www.example.com/mashup/"
   xml:base="http://www.example.com/mashup/services">
```

```
        <rdf:Description rdf:ID="Google">
         <rdf:type rdf:resource=
            "http://www.example.com/mashup/#SOAPService"/>
      </rdf:Description>

        <rdf:Description rdf:ID="Amazon">
         <rdf:type rdf:resource=
            "http://www.example.com/mashup/#RESTService"/>
         <rdf:type rdf:resource=
             "http://www.example.com/mashup/#SOAPService"/>
      </rdf:Description>

</rdf:RDF>
```

In this case, take note of the namespaces. You defined the `mashup:` prefix to match the base namespace used for the definitions, and defined a new namespace for the base.

You created two individuals, Google and Amazon, and assigned them types. Notice that the Amazon service, which accepts both REST and SOAP requests, actually has two types. That means it will have properties assigned to both classes. Note also that you can go ahead and simplify this code by using the alternate format, where you use the type name as the element name (see Listing 36):

Listing 36. Simplifying presentation

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:mashup="http://www.example.com/mashup/"
   xml:base="http://www.example.com/mashup/services">

      <mashup:SOAPService rdf:ID="Google">
      </mashup:SOAPService>

      <mashup:RESTService rdf:ID="Amazon">
         <rdf:type rdf:resource=
               "http://www.example.com/mashup/#SOAPService"/>
      </mashup:RESTService

</rdf:RDF>
```

Notice that you can still specify multiple classes, you just have to note the second class explicitly. Now look at how to add properties to these classes.

Creating properties

In an object-oriented language, you typically create properties in the context of the class. In RDFs, however, you create the properties on their own and assign them to the classes (see Listing 37):

Listing 37. Creating properties

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.example.com/mashup/">

      <rdf:Description rdf:ID="Service">
         <rdf:type rdf:resource=
            "http://www.w3.org/2000/01/rdf-schema#Class"/>
      </rdf:Description>

      <rdf:Description rdf:ID="SOAPService">
         <rdf:type rdf:resource=
             "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RESTService">
         <rdf:type rdf:resource=
             "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RSSFeed">
         <rdf:type rdf:resource=
            "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#RESTService" />
      </rdf:Description>

      <rdf:Property rdf:ID="baseUrl">
          <rdfs:domain rdf:resource="#RESTService" />
          <rdfs:range rdf:resource=
               "http://www.w3.org/2001/XMLSchema#anyURI"/>
          </rdf:Property>

      <rdf:Property rdf:ID="endpoint">
          <rdfs:domain rdf:resource="#SOAPService" />
          <rdfs:range rdf:resource=
              "http://www.w3.org/2001/XMLSchema#anyURI"/>
       </rdf:Property>

</rdf:RDF>
```

Here you created two properties, `baseUrl` and `endpoint`, and specified their domain and range. The domain is the type of resource on which the property can appear, and the range is the type of data it can hold.

The domain is more important than you might realize, because you can use it in reasoning. For example, you know that an endpoint property can only appear on a SOAPService, so if an individual has an endpoint, you know it's a SOAPService, even if it hasn't been explicitly marked that way. Later, when you start to look at ontologies and assign services and their data to specific classes, and make decisions, this ability to reason will become crucial.

Using subproperties

Just as you can create subclasses to create specializations of a class, you can create subproperties to create specializations of a property. For example (see Listing

38):

Listing 38. Creating subproperties

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.example.com/mashup/">

      <rdf:Description rdf:ID="Service">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
      </rdf:Description>

      <rdf:Description rdf:ID="SOAPService">
         <rdf:type rdf:resource=
               "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RESTService">
         <rdf:type rdf:resource=
                "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Service" />
      </rdf:Description>

      <rdf:Description rdf:ID="RSSFeed">
         <rdf:type rdf:resource=
                 "http://www.w3.org/2000/01/rdf-schema#Class"/>
         <rdfs:subClassOf rdf:resource="#RESTService" />
      </rdf:Description>

      <rdf:Property rdf:ID="accessUrl">
         <rdfs:domain rdf:resource="#Service" />
      </rdf:Property>

      <rdf:Property rdf:ID="baseUrl">
          <rdfs:subPropertyOf rdf:resource="#accessUrl" />
          <rdfs:domain rdf:resource="#RESTService" />
          <rdfs:range rdf:resource=
               "http://www.w3.org/2001/XMLSchema#anyURI"/>
          </rdf:Property>

      <rdf:Property rdf:ID="endpoint">
          <rdfs:subPropertyOf rdf:resource="#accessUrl" />
          <rdfs:domain rdf:resource="#SOAPService" />
          <rdfs:range rdf:resource=
                "http://www.w3.org/2001/XMLSchema#anyURI"/>
          </rdf:Property>

</rdf:RDF>
```

In Listing 38, you created an overall property, `accessUrl`, and defined your baseUrl and endpoint properties as specializations of that property. This means that if you create `accessUrl` as a resource, any properties you give to it, those properties will also apply to its subproperties.

Section 6. Summary

Wrap up and looking ahead

At this point I've discussed the Resource Description Framework (RDF), which provides a standard way to describe properties for various resources. These properties can themselves be resources, and can themselves have properties. You looked at how to create new resources and reference them from other documents and namespaces. You also looked at RDF Schema, which enables you to create classes and their relationships to each other.

This means you're ready to develop an ontology for your mashup system, and to specify it using Web Ontology Language (OWL).

Resources

Learn

- The ultimate mashup -- Web services and the semantic Web tutorial series: Take the other tutorials and create a custom mashup.
- Programmable Web: Stay up to date with the latest on mashups and the new Web 2.0 APIs.
- DB2 for Linux™, UNIX™ and Windows™ on developerWorks: Find current learning resources, downloads, product information, and support for IBM DB2, IBM's relational database management system, for information on demand.
- The W3C Semantic Web Activity site: Read about the Semantic Web.
- W3C RDF Activity: Visit this site for the latest on Resource Description Framework.
- Introduction to XML (Doug Tidwell, developerWorks, August 2002): Take this tutorial for a good grounding in XML.
- Read the following W3C documents that cover a wide and important range of RDF topics:
  - RDF Primer
  - RDF: Concepts and Abstract Syntax
  - RDF Semantics
  - RDF/XML Syntax Specification (revised)
  - RDF Vocabulary Description Language 1.0: RDF Schema
  - RDF Test Cases
- W3C's RDF-powered Validation Service: Check and visualize your RDF documents.
- Thinking XML: Introducing N-Triples (Uche Ogbuji, developerWorks, April 2003): Find more details on the non-XML formats discussed in this tutorial.
- Search RDF data with SPARQL (Philip McCarthy, developerWorks, May 2005): Learn to query and find RDF data find the data you need. You'll see more on that in Part 5 of this series.
- Thinking XML: XML meets semantics (Uche Ogbuji, developerWorks, February 2001 - June 2002): Explore Uche Ogbuji's series on XML, RDF, and semantics.
- Supercharging WSDL with RDF (Uche Ogbuji, developerWorks, November 2000): Add metadata to your service information.

- Using RDF with SOAP (Uche Ogbuji, developerWorks, February 2002): Examine ways to use SOAP to communicate information in RDF models.
- XML Watch: Tracking provenance of RDF data (Edd Dumbill, developerWorks, July 2003): Gather RDF data from anywhere, as long as you keep track of where it originally came from.
- developerWorks Java zone: Learn more about programming in Java with articles, tutorials, forums, and more.
- developerWorks XML zone: Find a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks about XML.
- IBM XML 1.1 certification: Find out how you can become an IBM Certified Developer in XML 1.1 and related technologies.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.

Get products and technologies

- Rational Web Developer: Download a trial version and make your development easier.
- DB2 Enterprise 9: Download a trial version of DB2 9 or DB2 Express-C 9, a no-charge version of DB2 Express 9 data server.

Discuss

- developerWorks XML forums: Communicate with other XML developers trying to solve the same problems you are.
- developerWorks blogs: Participate and get involved in the developerWorks community.

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).

Trademarks

IBM, DB2, and pureXML are trademarks of IBM Corporation in the United States, other countries, or both.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other company, product, or service names may be trademarks or service marks of others.