Solving Problems with the Resource Description Framework

Practical RDF

O'REILLY®

Shelley Powers

Practical RDF

Practical RDF

Shelley Powers

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 8

Jena: RDF in Java

Hewlett-Packard's Semantic Web team has been quietly working on Jena—a full-featured Java API for RDF—about as long as work has been progressing on RDF itself. In fact, the cochair of the RDF Working Group is Brian McBride, one of the creators of Jena.

Jena is an open source API and toolkit, accessible at Source Forge (*http://sourceforge.net/projects/jena*) or at *http://www.hpl.hp.com/semweb/jena.htm*. In addition, there's a Jena developers' discussion forum at *http://groups.yahoo.com/group/jena-dev/*.

Overview of the Classes

Included with the Jena toolkit are the dependencies and installation instructions, which I won't repeat here. I have worked with Jena on Linux (Red Hat), FreeBSD, and Windows; the examples included with Jena and the examples in this chapter work equally well in all environments. The only requirement is that you use JRE 1.2 or above.

A description of the many Java classes included with Jena is included with the installation (as Javadocs). I won't cover all of them here, only those most critical to understanding the underlying architecture in Jena.

I used Jena 1.6.1 in this chapter, but by the time this book is out, Jena 2.0 should be available. The Jena developers are refactoring many of the classes, changing class structure as well as making modifications to the API itself. These changes will break these examples, unfortunately. However, the concepts behind the examples should stay the same, and the book support site will have updated example source.

The Underlying Parser

Included within the Jena toolset is an RDF parser, ARP (an acronym for Another RDF Parser), accessible as a standalone product. You had a chance to look at and work with ARP in Chapter 7, so I won't go into additional detail here, since it works in the background with no further intervention necessary on our part. Our work begins once the RDF data is loaded into a model.

Though not covered in this book, Jena also includes an N3 (Notation3) parser.

The Model

Jena's API architecture focuses on the RDF model, the set of statements that comprises an RDF document, graph, or instantiation of a vocabulary. A basic RDF/XML document is created by instantiating one of the model classes and adding at least one statement (triple) to it. To view the RDF/XML, read it into a model and then access the individual elements, either through the API or through the query engine.

The `ModelMem` class creates an RDF model in memory. It extends `ModelCom`—the class incorporating common model methods used by all models—and implements the key interface, `Model`. In addition, the DAML class, `DAMLModelImpl`, subclasses `ModelMem`.

The `ModelRDB` class is an implementation of the model used to manipulate RDF stored within a relational database such as MySQL or Oracle. Unlike the memory model, `ModelRDB` persists the RDF data for later access, and the basic functionality between it and `ModelMem` is opening and maintaining a connection to a relational database in addition to managing the data. An interesting additional aspect of this implementation, as we'll see later in the section "In-Memory Versus Persistent Model Storage," is that you can also specify how the RDF model is stored within a relational database—as a flat table of statements, as a hash, or through stored procedures.

Once data is stored in a model, the next step is querying it.

One major change with Jena 2.0 is the addition of the `ModelFactory` to create new instances of models.

The Query

You can access data in a stored RDF model directly using specific API function calls, or via RDQL—an RDF query language. As will be demonstrated in Chapter 10, querying data using an SQL-like syntax is a very effective way of pulling data from an RDF model, whether that model is stored in memory or in a relational database.

Jena's RDQL is implemented as an object called Query. Once instantiated, it can then be passed to a query engine (`QueryEngine`) and the results stored in a query result (`QueryResult` and various implementations: `QueryResultsFormatter`, QueryResultsMem, and QueryResultsStream). To access specific returned values, program variables are bound to the result sets using the ResultBinding class.

Once data is retrieved from the RDF/XML, you can iterate through it using any number of iterators. Once you query data using the Query object, or if you access all RDF/XML elements of a specific class, you can assign the results to an iterator object and iterate through the set, displaying the results or looking for a specific value. Each of several different iterator classes within Jena is focused on specific RDF/XML classes, such as NodeIterator for general RDF nodes (literal or resource values), ResIterator, and StmtIterator.

DAML+OIL

Starting with later versions of Jena, support for DAML+OIL was added to the tool suite. DAML+OIL is a language for describing ontologies, a way of describing constraints and refinements for a given vocabulary that are beyond the sophistication of RDFS. Much of the effort on behalf of the Semantic Web is based on the Web Ontology Language at the W3C, which owes much of its effort to DAML+OIL. The principle DAML+OIL class within Jena, outside of the DAMLModel, is the DAMLOntology class. I won't be covering the DAML+OIL classes in this chapter, but the creators of Jena provide a tutorial that demonstrates them and is included in the documents you get when you download Jena.

Ontologies, DAML+OIL, and the W3C ontology language effort, OWL, are described in Chapter 12.

Creating and Serializing an RDF Model

Automating the process of creating an RDF/XML document is actually a fairly simple process, but you have to understand first how your RDF triples relate to one another. One approach to using Jena to generate RDF/XML for a particular vocabulary is to create a prototype document of the vocabulary and run it/them through the RDF Validator. Once the RDF/XML validates, parse it into N-Triples, and use these to build an application that can generate instances of a model of a given vocabulary, each using different data.

For the purposes of this chapter, I'm using Example 6-6 from Chapter 6 for a demonstration. This particular document, duplicated in this chapter's source, records the history and status of an article from one of my web sites. It makes a good example because it demonstrates the relationships that can appear within the PostCon vocabulary, and therefore makes a fine prototype for building an application that will build new versions of PostCon RDF/XML documents.

The examples in this chapter are, for the most part, working with the in-memory model from Jena. This model doesn't require the reader to have Berkeley DB, MySQL, or any other database installed.

Very Quick Simple Look

At its simplest, you can create an RDF model, create a single resource, add a couple of properties and then serialize it, all with just a few lines of code. So to get started, we'll do just that.

In Example 8-1, a new model is created, with the resource and one predicate repeated with two different objects. To create this model, an in-memory memory model is instantiated first, then an instance of an RDF resource using the Jena `Resource` class. Two instances of `Property` are created and attached to the module using `addProperty`, forming two complete RDF statements. The first parameter in the `addProperty` method is the `Property` instance, the second the actual property value. Once the model is built, it's printed out to standard output using the Jena `PrintWriter` class. For now, the values used within the model are all hardcoded into the application.

Example 8-1. Creating an RDF model with two statements, serialized to RDF/XML

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.common.PropertyImpl;
import java.io.FileOutputStream;
import java.io.PrintWriter;


public class pracRdfFirst extends Object {

    public static void main (String args[]) {
        String sURI = "http://burningbird.net/articles/monsters1.htm";
        String sPostcon = "http://www.burningbird.net/postcon/elements/1.0/";
        String sRelated = "related";
      try {
          // Create an empty graph
          Model model = new ModelMem( );

          // Create the resource
          Resource postcon = model.createResource(sURI);

          // Create the predicate (property)
          Property related = model.createProperty(sPostcon, sRelated);

          // Add the properties with associated values (objects)
          postcon.addProperty(related,
                              "http://burningbird.net/articles/monsters3.htm");
          postcon.addProperty(related,
                              "http://burningbird.net/articles/monsters2.htm");
```

Example 8-1. Creating an RDF model with two statements, serialized to RDF/XML (continued)

```
          // Print RDF/XML of model to system output
          model.write(new PrintWriter(System.out));

        } catch (Exception e) {
           System.out.println("Failed: " + e);
        }
      }
}
```

Once compiled, running the application results in the following output:

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:NS0='http://www.burningbird.net/postcon/elements/1.0/'
 >
  <rdf:Description rdf:about='http://burningbird.net/articles/monsters1.htm'>
    <NS0:related>http://burningbird.net/articles/monsters3.htm</NS0:related>
    <NS0:related>http://burningbird.net/articles/monsters2.htm</NS0:related>
  </rdf:Description>
</rdf:RDF>
```

The generated RDF validates within the RDF Validator, producing the graph shown in Figure 8-1.

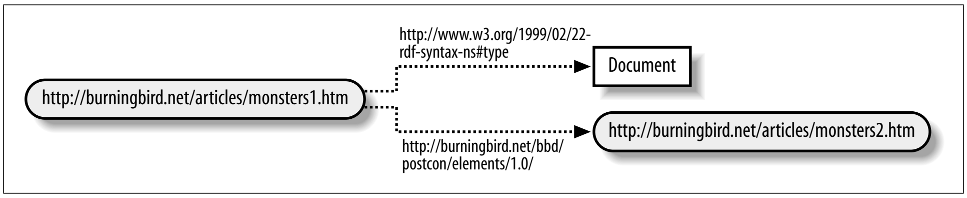
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
Document
http://burningbird.net/articles/monsters1.htm
http://burningbird.net/articles/monsters2.htm
http://burningbird.net/bbd/postcon/elements/1.0/

Figure 8-1. RDF model with one resource and two statements

At this point, we can continue creating and adding properties to the model directly in the application. However, the problem with creating the Property and Resource objects directly in the application that builds the models is that you have to duplicate this functionality across all applications that want to use the vocabulary. Not only is this inefficient, it adds to the overall size and complexity of an application. A better approach would be one the Jena developers demonstrated when they built their vocabulary objects: using a Java wrapper class.

Though omitted in Example 8-1 and other examples, you should close the memory model and free the resources using the `model.close()` method.

Encapsulating the Vocabulary in a Java Wrapper Class

If you look at your Jena installation, in the directory source code directory under the following path, you'll find several Java classes in the vocabulary directory, */com/hp/hpl/mesa/rdf/jena/vocabulary*.

The classes included wrap Dublin Core (DC) RDF, VCARD RDF, and so on. By using a wrapper class for the properties and resources of your RDF vocabulary, you have a way of defining all aspects of the RDF vocabulary in one spot, an approach that simplifies both implementation and maintenance.

The location of the vocabulary classes will change in Version 2.0.

In this section, we'll create a vocabulary class for PostCon, using the existing Jena vocabulary wrapper classes as a template, The PostCon wrapper class consists of a set of static strings holding property or resource labels and a set of associated RDF properties, as shown in Example 8-2. As complex as the example RDF file is, you may be surprised by how few entries there are in this class; PostCon makes extensive use of other RDF vocabularies for much of its data collection, including Dublin Core, which has a predefined vocabulary wrapper class included with Jena (`DC.java`).

Example 8-2. POSTCON vocabulary wrapper class

```
package com.burningbird.postcon.vocabulary;

import com.hp.hpl.mesa.rdf.jena.common.ErrorHelper;
import com.hp.hpl.mesa.rdf.jena.common.PropertyImpl;
import com.hp.hpl.mesa.rdf.jena.common.ResourceImpl;
import com.hp.hpl.mesa.rdf.jena.model.Model;
import com.hp.hpl.mesa.rdf.jena.model.Property;
import com.hp.hpl.mesa.rdf.jena.model.Resource;
import com.hp.hpl.mesa.rdf.jena.model.RDFException;

public class POSTCON extends Object {

    // URI for vocabulary elements
    protected static final String uri = "http://burningbird.net/postcon/elements/1.0/";

    // Return URI for vocabulary elements
    public static String getURI( )
    {
        return uri;
    }

    // Define the property labels and objects
           static final String   nbio = "bio";
    public static       Property bio = null;
           static final String   nrelevancy = "relevancy";
    public static       Property relevancy = null;
           static final String   npresentation = "presentation";
    public static       Resource presentation = null;
           static final String   nhistory = "history";
    public static       Property history = null;
           static final String   nmovementtype = "movementType";
```

Example 8-2. POSTCON vocabulary wrapper class (continued)

```
    public static       Property movementtype = null;
           static final String   nreason = "reason";
    public static       Property reason = null;
           static final String   nstatus = "currentStatus";
    public static       Property status = null;
           static final String   nrelated = "related";
    public static       Property related = null;
           static final String   ntype = "type";
    public static       Property type = null;
           static final String   nrequires = "requires";
    public static       Property requires = null;


    // Instantiate the properties and the resource
    static {
        try {

            // Instantiate the properties
            bio          = new PropertyImpl(uri, nbio);
            relevancy    = new PropertyImpl(uri, nrelevancy);
            presentation = new PropertyImpl(uri, npresentation);
            history      = new PropertyImpl(uri, nhistory);
            related      = new PropertyImpl(uri, nrelated);
            type         = new PropertyImpl(uri, ntype);
            requires     = new PropertyImpl(uri, nrequires);
            movementtype = new PropertyImpl(uri, nmovementtype);
            reason       = new PropertyImpl(uri, nreason);
            status       = new PropertyImpl(uri, nstatus);

        } catch (RDFException e) {
            ErrorHelper.logInternalError("POSTCON", 1, e);
        }
    }

}
```

At the top of the example code, after the declarations, is a static string holding the URI of the PostCon element vocabulary and a method to return it. Following these is a list of declarations for each property, including a `Property` element and the associated label for each.

Note that the two PostCon RDF classes Resource and Movement are not included. The reason is that I'm using the Jena Resource class to define them and then adding `rdf:type` to define the type of the resource. The resulting RDF graph is the same—only the syntax is different.

Once the properties are defined in the code, they are instantiated, and the file is saved and compiled. To import this class, use the following in your Java applications:

```
import com.burningbird.postcon.vocabulary.POSTCON;
```

At this point, the PostCon vocabulary wrapper class is ready for use. We rewrite the application in Example 8-1, except this time we'll use the POSTCON wrapper class, as shown in Example 8-3. In addition, we'll cascade the `addProperty` calls directly in the function call to create the resource (`createResource`), to keep the code compact, as well as to show a more direct connection between the two.

Example 8-3. Using wrapper class to add properties to resource

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.vocabulary.*;
import com.burningbird.postcon.vocabulary.POSTCON;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class pracRDFSecond extends Object {


    public static void main (String args[]) {

        // Resource names
        String sResource = "http://burningbird.net/articles/monsters1.htm";
        String sRelResource1 = "http://burningbird.net/articles/monsters2.htm";
        String sRelResource2 = "http://burningbird.net/articles/monsters3.htm";

        try {
            // Create an empty graph
            Model model = new ModelMem( );

            // Create the resource
            //   and add the properties cascading style
            Resource article
              = model.createResource(sResource)
               .addProperty(POSTCON.related, model.createResource(sRelResource1))
               .addProperty(POSTCON.related, model.createResource(sRelResource2));

            // Print RDF/XML of model to system output
            model.write(new PrintWriter(System.out));

        } catch (Exception e) {
            System.out.println("Failed: " + e);
        }
    }
}
```

As you can see, using the wrapper class simplified the code considerably. The new application is saved, compiled, and run. The output from this application is shown in Example 8-4. Again, running it through the RDF Validator confirms that the serialized RDF/XML represents the model correctly and validly.

Example 8-4. Generated RDF/XML from serialized PostCon submodel

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:NS0='http://burningbird.net/postcon/elements/1.0/'
 >
  <rdf:Description rdf:about='http://burningbird.net/articles/monsters1.htm'>
    <NS0:related rdf:resource='http://burningbird.net/articles/monsters2.htm'/>
    <NS0:related rdf:resource='http://burningbird.net/articles/monsters3.htm'/>
  </rdf:Description>
</rdf:RDF>
```

You've probably noted by now that Jena generates namespace prefixes for the vocabulary elements. As you'll see later, you can change the prefix used for namespaces. However, the specific prefix used is unimportant, except perhaps for readability across models when the same vocabulary is used in multiple places, such as the Dublin Core vocabulary.

Adding More Complex Structures

As has been demonstrated, adding literal or simple resource properties for a specific RDF resource in a model is quite uncomplicated with Jena. However, many RDF models make use of more complex structures, including nesting resources following the RDF node-edge-node pattern. In this section, we'll demonstrate how Jena can just as easily handle more complex RDF model structures and their associated RDF/XML.

Much of the code shown in this chapter came about through development of the PostCon application (RDF Web Content Information System), discussed throughout the book. You can download the source for the Java-based implementation of PostCon at SourceForge (*http://rdfcontent.sourceforge.net/*).

The `pstcn:bio` property is, itself, a resource that does not have a specific URI—a blank node, or *bnode*. Though not a literal, it's still added as a property using `addProperty`.

In Example 8-5, a new resource representing the article is created and the two related resource properties are added. In addition, a new resource is created for bio, and several properties are added to it; these properties are defined within the DC vocabulary, and I used the DC wrapper class to create them. Once the resource is implemented, I attach it to a higher-level resource using `addProperty`.

Example 8-5. Adding a blank node to a model

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.vocabulary.*;
import com.burningbird.postcon.vocabulary.POSTCON;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class pracRDFThird extends Object {

    public static void main (String args[]) {

// Resource names
String sResource = "http://burningbird.net/articles/monsters1.htm";
String sRelResource1 = "http://burningbird.net/articles/monsters2.htm";
String sRelResource2 = "http://burningbird.net/articles/monsters3.htm";
String sType = "http://burningbird.net/postcon/elements/1.0/Resource";

try {
 // Create an empty graph
 Model model = new ModelMem( );

 // Create the resource
 // and add the properties cascading style
 Resource article
 = model.createResource(sResource)
 .addProperty(POSTCON.related, model.createResource(sRelResource1))
 .addProperty(POSTCON.related, model.createResource(sRelResource2));

 // Create the bio bnode resource
 // and add properties
 Resource bio
 = model.createResource( )
 .addProperty(DC.creator, "Shelley Powers")
 .addProperty(DC.publisher, "Burningbird")
 .addProperty(DC.title, model.createLiteral("Tale of Two Monsters: Legends", "en"));

 // Attach to main resource
 article.addProperty(POSTCON.bio, bio);

 // Print RDF/XML of model to system output
  model.write(new PrintWriter(System.out));

        } catch (Exception e) {
            System.out.println("Failed: " + e);
        }
    }
}
String sResource = "http://burningbird.net/articles/monsters1.htm";
String sRelResource1 = "http://burningbird.net/articles/monsters2.htm";
```

I could have used the cascade approach to add the bio directly to the document resource as it was being created. However, creating bio separately and then adding it to the top-level resource is, in my opinion, easier to read, and the resulting RDF model and serialized RDF/XML is identical. The results of the application are shown in Example 8-6. As you can see, Jena uses `rdf:nodeID` and separates out the resource, rather than nesting it. This is nothing more than convenience and syntactic sugar—the resulting RDF graph is still equivalent in meaning.

Example 8-6. Generated RDF/XML demonstrating more complex structures

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:NS0='http://burningbird.net/postcon/elements/1.0/'
  xmlns:dc='http://purl.org/dc/elements/1.0/'
 >
  <rdf:Description rdf:nodeID='A0'>
    <dc:creator>Shelley Powers</dc:creator>
    <dc:publisher>Burningbird</dc:publisher>
    <dc:title xml:lang='en'>Tale of Two Monsters: Legends</dc:title>
  </rdf:Description>
  <rdf:Description rdf:about='http://burningbird.net/articles/monsters1.htm'>
    <NS0:related rdf:resource='http://burningbird.net/articles/monsters2.htm'/>
    <NS0:related rdf:resource='http://burningbird.net/articles/monsters3.htm'/>
    <NS0:bio rdf:nodeID='A0'/>
  </rdf:Description>
</rdf:RDF>
```

The example demonstrates how to implement the striped XML quality of RDF, which has a node-edge-node-edge pattern of nesting. Another RDF pattern that PostCon supports is a container holding the resource's history, which is implemented in the later section titled "Creating a Container."

Creating a Typed Node

The RDF model created to this point shows the top-level resource as a basic `rdf:Description` node, with a given URI. However, in the actual RDF/XML, the top-level node is what is known as a typed node, which means it is defined with a specific `rdf:type` property.

Implementing a typed node in Jena is actually quite simple, by the numbers.

First, the POSTCON wrapper class needs to be modified to add the new resource implementation. To support this, two new Jena classes are imported into the POSTCON Java code:

```
import com.hp.hpl.mesa.rdf.jena.common.ResourceImpl;
import com.hp.hpl.mesa.rdf.jena.model.Resource;
```

Next, the document resource definition is added:

```
// add the one resource
      static final String   nresource = "resource";
public static       Resource resource = null;
```

Finally, the resource is instantiated:

```
resource = new ResourceImpl(uri+nresource);
```

Once the wrapper class is modified, the typed node information is implemented within the Jena code, as shown in Example 8-7.

Example 8-7. Adding an rdf:type for the top-level document resource

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.vocabulary.*;
import com.burningbird.postcon.vocabulary.POSTCON;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class chap1005 extends Object {


    public static void main (String args[]) {

        // Resource names
        String sResource = "http://burningbird.net/articles/monsters1.htm";

        try {
            // Create an empty graph
            Model model = new ModelMem( );

            // Create the resource
            //   and add the properties cascading style
            Resource article
              = model.createResource(sResource)
                     .addProperty(RDF.type, POSTCON.resource);

            // Print RDF/XML of model to system output
            model.write(new PrintWriter(System.out));

        } catch (Exception e) {
            System.out.println("Failed: " + e);
        }
    }
}
```

The resulting RDF/XML:

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
 >
  <rdf:Description rdf:about='http://burningbird.net/articles/monsters1.htm'>
    <rdf:type rdf:resource='http://burningbird.net/postcon/elements/1.0/Resource'/>
  </rdf:Description>
</rdf:RDF>
```

is equivalent to the same RDF/XML used in the sample document:

```
<pstcn:Resource rdf:about="monsters1.htm">
...
</pstcn:Resource>
```

Both result in the exact same RDF model, shown in Figure 8-2.

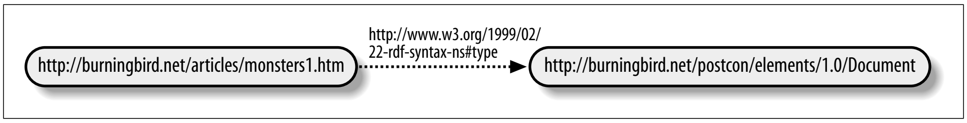
http://burningbird.net/articles/monsters1.htm
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://burningbird.net/postcon/elements/1.0/Document

Figure 8-2. RDF model of typed (document) node

Creating a Container

As discussed earlier in the book, an RDF container is a grouping of related items. There are no formalized semantics for a container other than this, though tools and applications may add additional semantics based on type of container: `Alt`, `Seq`, or `Bag`.

The PostCon vocabulary uses an `rdf:Seq` container to group the resource history, with the application-specific implication that if tools support this concept, the contained items are sequenced in order, from top to bottom, within the container:

```
<pstcn:history>
   <rdf:Seq>
      <rdf:_1 rdf:resource="http://www.yasd.com/dynaearth/monsters1.htm" />
      <rdf:_2 rdf:resource="http://www.dynamicearth.com/articles/monsters1.htm" />
      <rdf:_3 rdf:resource="http://burningbird.net/articles/monsters1.htm" />
   </rdf:Seq>
</pstcn:history>
```

For tools that don't support my additional container semantics, the items can be sequenced by whatever properties are associated with each contained resource—the date, URI, movement type, or even random sequencing:

```
<rdf:Description rdf:about="http://www.yasd.com/dynaearth/monsters1.htm">
    <pstcn:movementType>Add</pstcn:movementType>
    <pstcn:reason>New Article</pstcn:reason>
    <dc:date>1998-01-01T00:00:00-05:00</dc:date>
</rdf:Description>
```

RDF containers are just a variation of typed node and can be implemented directly just by using the same code shown to this point. After all, a container is nothing more than a blank node with a given `rdf:type` (such as `http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq`) acting as the subject for several statements, all with the same predicate and all pointing to objects that are resources. You could emulate containers directly given previous code. However, it's a lot simpler just to use the APIs.

In Example 8-8, an RDF container, an `rdf:Seq`, is created and three resources are added to it. Each of the resources has properties of its own, including `pstcn:movementType`, `reason` (both of which are from POSTCON), and `date` (from DC). Once completed, the `rdf:Seq` is then added to the document resource.

Example 8-8. Adding the history container to the model

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.vocabulary.*;
import com.burningbird.postcon.vocabulary.POSTCON;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class pracRDFFifth extends Object {


    public static void main (String args[]) {

        // Resource names
        String sResource = "http://burningbird.net/articles/monsters1.htm";
        String sHistory1 = "http://www.yasd.com/dynaearth/monsters1.htm";
        String sHistory2 = "http://www.dynamicearth.com/articles/monsters1.htm";
        String sHistory3 = "http://www.burningbird.net/articles/monsters1.htm";

        try {
            // Create an empty graph
            Model model = new ModelMem( );

            // Create Seq
            Seq hist = model.createSeq( )
               .add (1, model.createResource(sHistory1)
               .addProperty(POSTCON.movementtype, model.createLiteral("Add"))
               .addProperty(POSTCON.reason, model.createLiteral("New Article"))
               .addProperty(DC.date, model.createLiteral("1998-01-01T00:00:00-05:00")))
               .add (2, model.createResource(sHistory2)
               .addProperty(POSTCON.movementtype, model.createLiteral("Move"))
               .addProperty(POSTCON.reason, model.createLiteral("Moved to separate
                    dynamicearth.com domain"))
               .addProperty(DC.date, model.createLiteral("1999-10-31:T00:00:00-05:00")))
               .add (3, model.createResource(sHistory3)
               .addProperty(POSTCON.movementtype, model.createLiteral("Move"))
               .addProperty(POSTCON.reason, model.createLiteral("Collapsed
                    into Burningbird"))
               .addProperty(DC.date, model.createLiteral("2002-11-01:T00:00:00-5:00")));

            // Create the resource
            //   and add the properties cascading style
            Resource article
              = model.createResource(sResource)
              .addProperty(POSTCON.history, hist);
```

Example 8-8. Adding the history container to the model (continued)

```
            // Print RDF/XML of model to system output
            RDFWriter writer = model.getWriter( );
            writer.setNsPrefix("pstcn", "http://burningbird.net/postcon/elements/1.0/");
            writer.write(model, new PrintWriter(System.out),
               "http://burningbird.net/articles" );

        } catch (Exception e) {
            System.out.println("Failed: " + e);
        }
    }
}
```

Another new item added with this code is the `RDFWriter.setNsPrefix` method, which defines the prefix so that it shows as `pstcn` rather than the default of `NS0`. This isn't necessarily important—whatever abbreviation used is resolved to the namespace within the model—but it does make the models easier to read if you use the same QName all the time.

As described in Chapter 4, a container is a grouping of like items, and there are no additional formal semantics attached to the concept of container. Now, the fact that I used `rdf:Seq` could imply that the items within the container should be processed in order, from first to last. However, this is up to the implementation to determine exactly how an `rdf:Seq` container is processed outside of the formal semantics within the RDF specifications.

What's interesting is that, within Jena, a container is treated exactly as the typed node that I described earlier—which means that the generated RDF/XML, as shown in Example 8-9, shows the `rdf:Seq` as its typed node equivalent, rather than in the container-like syntax shown in the example source.

Example 8-9. Generated RDF/XML showing container defined as typed node

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:pstcn='http://burningbird.net/postcon/elements/1.0/'
  xmlns:dc='http://purl.org/dc/elements/1.0/'
 >
  <rdf:Description rdf:about='http://burningbird.net/articles/monsters1.htm'>
    <pstcn:history rdf:nodeID='A0'/>
  </rdf:Description>
  <rdf:Description rdf:about='http://www.dynamicearth.com/articles/monsters1.htm'>
    <pstcn:movementType>Move</pstcn:movementType>
    <pstcn:reason>Moved to separate dynamicearth.com domain</pstcn:reason>
    <dc:date>1999-10-31:T00:00:00-05:00</dc:date>
  </rdf:Description>
  <rdf:Description rdf:about='http://www.burningbird.net/articles/monsters1.htm'>
    <pstcn:movementType>Move</pstcn:movementType>
    <pstcn:reason>Collapsed into Burningbird</pstcn:reason>
    <dc:date>2002-11-01:T00:00:00-5:00</dc:date>
  </rdf:Description>
```

Example 8-9. Generated RDF/XML showing container defined as typed node (continued)

```
  <rdf:Description rdf:nodeID='A0'>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
    <rdf:_1 rdf:resource='http://www.yasd.com/dynaearth/monsters1.htm'/>
    <rdf:_2 rdf:resource='http://www.dynamicearth.com/articles/monsters1.htm'/>
    <rdf:_3 rdf:resource='http://www.burningbird.net/articles/monsters1.htm'/>
  </rdf:Description>
  <rdf:Description rdf:about='http://www.yasd.com/dynaearth/monsters1.htm'>
    <pstcn:movementType>Add</pstcn:movementType>
    <pstcn:reason>New Article</pstcn:reason>
    <dc:date>1998-01-01T00:00:00-05:00</dc:date>
  </rdf:Description>
</rdf:RDF>
```

I prefer the Jena implementation of the container because it implies nothing about container-like behavior that doesn't exist within the RDF specifications. The generated RDF/XML provides a clearer picture of a set of like resources, grouped for some reason, and then added as a property to another resource. No more, no less.

Now that we've had a chance to build RDF models and view the serialized RDF/XML from them, we'll take a look at parsing and accessing data in existing RDF/XML documents.

One type of RDF statement I haven't demonstrated is a reified statement, primarily because I don't use reified statements within my applications. However, if you need reification for your own effort, you can find a couple of example Java applications that build reified statements within the Jena Toolkit.

Parsing and Querying an RDF Document

Once an RDF/XML document is created, it serves no useful purpose unless the data in the document can be parsed and queried. In many ways, the advantage to something like RDF/XML is that the data is structured in specific ways, making it easier to access different data with the same code.

This section will take a look at opening an existing RDF/XML document, both within the filesystem and through the Internet, and accessing the data contained within the documents.

Just Doing a Basic Dump

When accessing the data within an RDF/XML document, you'll want to access the data in two different ways—accessing specific pieces of data or accessing all of it for alternative presentation. For instance, most of the tools discussed in Chapters 14 and 15 are interested in all the data within an RDF/XML document, data that is then transformed in one way or another.

One of the most common ways of "dumping" the data within an RDF/XML document (outputting all the data in a new format) is to print it out in N-Triples format. This was demonstrated with the parser attached with the Jena Toolkit, ARP. However, another way of looking at the data is to dump out a listing of objects of one type or another.

In Example 8-10, the PostCon RDF file for the demonstration article is accessed and opened into a memory model using the `read` method; this method takes the URL of the file as its parameter. Once the model is loaded, the `listObjects` method is called on the model object and assigned to a `nodeIterator`. This object is just one of the many different iterators that Jena provides: `nodeIterator`, `stmtIterator`, `ResIterator`, and so on. Each of these is specialized to provide access to specific Jena object types. In the example, once the `nodeIterator` is populated, it's traversed, and all of the RDF objects—the property "values"—are printed out using the simple `toString` base method.

Example 8-10. Basic dump of objects, printing out object values

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;

public class pracRDFSixth extends Object {

public static void main (String args[]) {

String sUri = args[0];

try {

   // Create memory model, read in RDF/XML document
   ModelMem model = new ModelMem( );
   model.read(sUri);

   // Print out objects in model using toString
   NodeIterator iter = model.listObjects( );
   while (iter.hasNext( )) {
        System.out.println("  " + iter.next( ).toString( ));
   }

   } catch (Exception e) {
            System.out.println("Failed: " + e);
   }
 }
}
```

The application is run against the *monsters1.rdf* example file:

```
java pracRDFSixth http://burningbird.net/articles/monsters1.rdf
```

This is probably one of the simplest Jena applications you can write and test to make sure that a model is loaded correctly. Instead of objects, you could also dump out the subjects (ResIterator and listSubjects) or even the entire statement (StmtIterator and listStatements). The functionality is relatively the same, except for the iterator and the fetch method called.

Accessing Specific Values

Instead of listing all statements or all objects, you can fine-tune the code to list only subjects, statements, or objects matching specific properties, using the property implementations created within the wrapper classes, such as POSTCON.

To access all objects that have the PostCon related property, the POSTCON wrapper class is added to the import section:

```
import com.burningbird.postcon.vocabulary.POSTCON;
```

Next, the listObjectsOfProperty method is used instead of listObjects:

```
NodeIterator iter = model.listObjectsOfProperty(POSTCON.related);
```

That's it to access all objects given a specific property. As you can see, the wrapper class is handy for more than just creating a model.

To access all the statements for a given resource, first access the resource from the model and then list all the properties associated with that resource. In Example 8-11, all of the statements are accessed for the top-level resource contained within the document. Traversing the list of statements, the subject is accessed and printed out (both namespace and local name), followed by the predicate (again, namespace and local name), and finally the object.

Example 8-11. Printing out each statement triple for a given RDF/XML document

```
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.burningbird.postcon.vocabulary.POSTCON;

public class pracRDFSeventh extends Object {

public static void main (String args[]) {

String sUri = args[0];
String sResource = args[1];

try {

   // Create memory model, read in RDF/XML document
   ModelMem model = new ModelMem( );
   model.read(sUri);

   // Find resource
   Resource res = model.getResource(sResource);
```

Example 8-11. Printing out each statement triple for a given RDF/XML document (continued)

```
   // Find properties
   StmtIterator iter = res.listProperties( );

   // Print out triple - subject | property | object
   while (iter.hasNext( )) {
        // Next statement in queue
        Statement stmt = iter.next( );

        // Get subject, print
        Resource res2 = stmt.getSubject( );
        System.out.print(res2.getNameSpace( ) + res2.getLocalName( ));

        // Get predicate, print
        Property prop = stmt.getPredicate( );
        System.out.print(" " + prop.getNameSpace( ) + prop.getLocalName( ));

        // Get object, print
        RDFNode node = stmt.getObject( );
        System.out.println(" " + node.toString( ) + "\n");
   }

   } catch (Exception e) {
            System.out.println("Failed: " + e);
   }
 }
}
```

Running this application outputs the triple for each statement for the document, including application-generated object values for blank nodes:

```
http://burningbird.net/articles/monsters1.htm
http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://burningbird.net/postcon/
elements/1.0/Resource

http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/bio
anon:a9ae05:f2ecfdc9db:-7fff

http://burningbird.net/articles/monsters1.htm http://burningbird.net/postcon/
elements/1.0/relevancy
anon:a9ae05:f2ecfdc9db:-7ff7

http://burningbird.net/articles/monsters1.htm http://burningbird.net/postcon/
elements/1.0/presentation
anon:a9ae05:f2ecfdc9db:-7fec

http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/history
anon:a9ae05:f2ecfdc9db:-7fde

http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/related
http://burningbird.net/articles/monsters2.htm
```

```
http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/related
http://burningbird.net/articles/monsters3.htm

http://burningbird.net/articles/monsters1.htm
http://burningbird.net/postcon/elements/1.0/related
http://burningbird.net/articles/monsters4.htm
```

Note in the code that the variation of `getObject` used is the one returning an `RDFNode` object. The reason is that other variations work only if the object is a literal and throw exceptions if a nonliteral is found. Since some of the objects in this document are resources, the `RDFNode` method works best.

As can be seen from the examples, querying the data in an RDF/XML document doesn't have to be difficult—you just have to remember the triple nature of the statements in RDF/XML.

One of the most powerful aspects of Jena is the ability to use a query language—RDQL—to query an RDF model to data that matches given patterns. This is explored in Chapter 10.

In-Memory Versus Persistent Model Storage

All the examples to this point have used the memory model, but Jena also provides the capability to persist data to relational database storage. The databases supported are MySQL, PostgreSQL, Interbase, and Oracle. Within each database system, Jena also supports differing storage layouts:

Generic
: All statements are stored in a single table, and resources and literals are indexed using integer identifiers generated by the database.

GenericProc
: Similar to generic, but data access is through stored procedures.

MMGeneric
: Similar to generic but can store multiple models.

Hash
: Similar to generic but uses MD5 hashes to generate the identifiers.

MMHash
: Similar to hash but can store multiple models.

The first step of storing a model in a database is to create the structure to store the data. The tables must be created in an already existing database, which has been formatted and had tables added. This code needs to be run once. After the database structure is created, it can then be opened directly in another application or used within the same application.

In Example 8-12, I'm storing two models in the database using a different name for each. In addition, I'm also creating the JDBC connection directly rather than having `DBConnection` create it for me. The model used is based on a MySQL database, using the `MMGeneric` layout. I'm not using the slightly more efficient hash method (`MMHash`), primarily because the generic layout is the better one to take if you're thinking of accessing the data directly through JDBC rather than through Jena.

At the time of this writing, using `DBConnection` to make the JDBC connection is failing in the second application to access the same database. Creating an instance of the JDBC connection and passing it in as a parameter to `DBConnection` averts this failure.

Once the database is formatted, two RDF/XML documents are opened and stored in two separate models within the database.

Example 8-12. Persisting two RDF/XML models to a MySQL database

```
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.rdb.ModelRDB;
import com.hp.hpl.mesa.rdf.jena.rdb.DBConnection;
import java.sql.*;

public class pracRDFEighth extends Object {


public static void main (String args[]) {

// Pass two RDF documents, connection string,
String sUri = args[0];
String sUri2 = args[1];
String sConn = args[2];
String sUser = args[3];
String sPass = args[4];

try {
  // Load driver class
  Class.forName("com.mysql.jdbc.Driver").newInstance( );

   // Establish connection - replace with your own conn info
   Connection con = DriverManager.getConnection(sConn, "user", "pass");
   DBConnection dbcon = new DBConnection(con);

   // Format database
   ModelRDB.create(dbcon, "MMGeneric", "Mysql");

   // Create and read first model
   ModelRDB model1 = ModelRDB.createModel(dbcon, "one");
   model1.read(sUri);
```

Example 8-12. Persisting two RDF/XML models to a MySQL database (continued)

```
   // Create and read second model
   ModelRDB model2 = ModelRDB.createModel(dbcon, "two");
   model2.read(sUri2);


   } catch (Exception e) {
            System.out.println("Failed: " + e);
   }
 }
}
```

The application expects the following command line:

```
    java pracRDFEighth firstrdffile secondrdffile connect_string username password
```

You'll need to adjust the database connection string, username, and password to fit your environment. In the example, instead of reading the two models into separate databases, I could also have read them into the same database.

Once the model data is persisted, any number of applications can then access it. In Example 8-13, I'm accessing both models, dumping all of the objects in the first and writing out triples from the second.

Example 8-13. Accessing RDF models stored in MySQL database

```
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.rdb.ModelRDB;
import com.hp.hpl.mesa.rdf.jena.rdb.DBConnection;
import java.sql.*;

public class pracRDFNinth extends Object {


public static void main (String args[]) {

String sConn = args[0];
String sUser = args[1];
String sPass = args[2];

try {
  // load driver class
  Class.forName("com.mysql.jdbc.Driver").newInstance( );

   // Establish connection - replace with your own conn info
     Connection con = DriverManager.getConnection(sConn, sUser, sPass);
   DBConnection dbcon = new DBConnection(con);

   // Open two existing models
   ModelRDB model1 = ModelRDB.open(dbcon, "one");
   ModelRDB model2 = ModelRDB.open(dbcon, "two");
```

Example 8-13. Accessing RDF models stored in MySQL database (continued)

```
   // Print out objects in first model using toString
   NodeIterator iter = model1.listObjects( );
   while (iter.hasNext( )) {
        System.out.println("  " + iter.next( ).toString( ));
   }

   // Print out triples in second model - find resource
   Resource res = model2.getResource("http://burningbird.net/articles/monsters1.htm");

   // Find properties
   StmtIterator sIter = res.listProperties( );

   // Print out triple - subject | property | object
   while (sIter.hasNext( )) {
        // Next statement in queue
        com.hp.hpl.mesa.rdf.jena.model.Statement stmt = sIter.next( );

        // Get subject, print
        Resource res2 = stmt.getSubject( );
        System.out.print(res2.getNameSpace( ) + res2.getLocalName( ));

        // Get predicate, print
        Property prop = stmt.getPredicate( );
        System.out.print(" " + prop.getNameSpace( ) + prop.getLocalName( ));

        // Get object, print
        RDFNode node = stmt.getObject( );
        System.out.println(" " + node.toString( ) + "\n");
   }

   } catch (Exception e) {
            System.out.println("Failed: " + e);
   }
 }
}
```

Jena uses a highly normalized data model for the RDF statements. In addition to accessing the data through the Jena API, you can also access it directly using whatever database connectivity you prefer. However, I recommend that you access the data for read-only purposes and leave updates to the Jena API.