Introduction to Jena

Use semantic web technologies in your Java applications

Majid Sazvar
sazvar@stu-mail.um.ac.ir

Knowledge Engineering Research Group (KERG)
Ferdowsi University of Mashhad

2011

What is Jena?

- Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine.

- Jena is open source and grown out of work with the HP Labs Semantic Web Program.

- Jena Homepage:

http://jena.sourceforge.net

What is Jena?

- The Jena Framework includes:
  - A RDF API
    - Reading and writing RDF in RDF/XML, N3 and N-Triples.
  - ARQ Engine
    - ARQ is a query engine for Jena that supports the SPARQL RDF Query language.
  - TDB Engine
    - It provides for large scale storage and query of RDF datasets using a pure Java engine.
  - SDB Engine
    - It provides for scalable storage and query of RDF datasets using conventional relational databases for use in standalone applications, J2EE and other application frameworks.

What is Jena?

- The Jena Framework includes:
  - Joseki
    - Joseki is an HTTP engine that supports the SPARQL Protocol and the SPARQL RDF Query language.
  - Eyeball
    - Eyeball is a Jena contrib. for checking RDF/OWL models for common issues such as illegal URIs, missing property values, and incorrect prefix mappings.

Why Jena?

Create a RDF Model

- The *ModelFactory* class enables the creation of models:
  - *ModelFactory.createDefaultModel(),* allows the creation of an in-memory model
    - Returns a Model instance over which you can create Resources
  - *Model.createProperty()* allow relationship creation
  - To add statements for a model use: *Resource.addProperty()* or *Model.createStatement()* and *Model.add()*

- In Jena a statement is composed by:
  - A subject in the form of a *Resource*
  - A predicate represented by a *Property* class
  - An object, either a *Literal* or *Resource*

- *Resource*, *Property* and *Literal* inherit from *RDFNode*

Example

```
// URI declarations
String familyUri = "http://family/";
String relationshipUri = "http://purl.org/vocab/relationship/";

// Create an empty Model
Model model = ModelFactory.createDefaultModel();

// Create a Resource for each family member, identified by their URI
Resource adam = model.createResource(familyUri+"adam");
Resource beth = model.createResource(familyUri+"beth");
Resource chuck = model.createResource(familyUri+"chuck");
Resource dotty = model.createResource(familyUri+"dotty");
// and so on for other family members

// Create properties for the different types of relationship to represent
Property childOf = model.createProperty(relationshipUri,"childOf");
Property parentOf = model.createProperty(relationshipUri,"parentOf");
Property siblingOf = model.createProperty(relationshipUri,"siblingOf");

// Add properties to adam describing relationships to other family members
adam.addProperty(siblingOf,beth);
adam.addProperty(parentOf,edward);

// Can also create statements directly . . .
Statement statement = model.createStatement(adam,parentOf,fran);
// but remember to add the created statement to the model
model.add(statement);
```

Interrogating an RDF Model

- By means of *listXXX()* method in *Model* and *Resource* interfaces
  - It returns specializations of java.util.Iterator
  - *Model.listStatements(Resource s, Property p, RDFNode o)*

- Examples:
  - *ResIterator parents = model.listSubjectsWithProperty(parentOf);*
    - *Resource person = parents.nextResource();*
  - *NodeIterator moreParents = model.listObjectsOfProperty(childOf);*
  - *StmtIterator moreSiblings = edward.listProperties(siblingOf);*
  - *model.listStatements(adam,null,null);*

Importing and Persisting Models

- So far we have worked with in-memory models
  - Necessary to persist and retrieve models
- Easiest solution:
  - *Model.read()*
  - *Model.write()*
- More sophisticated over RDBMS:
  - MySQL

RDF Data Query Language (RDQL)

- RDQL is a query language for RDF
  - Allows complex queries to be expressed concisely
    - A query engine performing the hard work of accessing the data model

- Example:

```
SELECT      ?definition
WHERE
            (?concept, <wn:wordForm>, "domestic dog"),
            (?concept, <wn:glossaryEntry>, ?definition)
USING
            wn FOR http://www.cogsci.princeton.edu/~wn/schema/
```

Using RDQL

- Need to import *com.hp.hpl.jena.rdql*
- To create a query instantiate *Query* and pass as a *String* the query
- Create *QueryEngine* and invoke *QueryEngine.exec(Query)*
- Variables can be bound to values through a *ResultBinding* object

Example

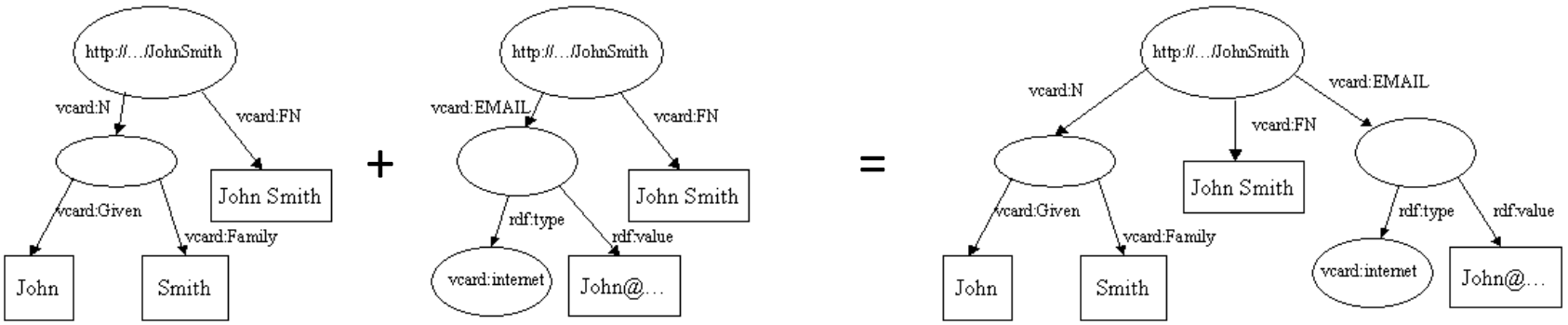
```
// Create a new query passing a String containing the RDQL to execute, containing variables x and y
Query query = new Query(queryString);

// Set the model to run the query against
query.setSource(model);

// A ResultBinding specifies mappings between query variables and values
ResultBindingImpl initialBinding = new ResultBindingImpl() ;

// Bind the query's first variable to a resource
Resource someResource = getSomeResource();
initialBinding.add("x", someResource);

// Bind the query's second variable to a literal value
RDFNode foo = model.createLiteral("bar");
initialBinding.add("y", foo);

// Use the query to create a query engine
QueryEngine qe = new QueryEngine(query);

// Use the query engine to execute the query
QueryResults results = qe.exec();
```

Operations on Models

- The common set operations:
  - Union (*.union(Model)*) , intersection (*.intersection(Model)*) and difference (*.difference(Model)*)

- Example: union

```
// read the RDF/XML files
model1.read(new InputStreamReader(in1), "");
model2.read(new InputStreamReader(in2), "");
// merge the Models
Model model = model1.union(model2);
// print the Model as RDF/XML
model.write(system.out, "RDF/XML-ABBREV");
```



SPARQL

- Builds on previously existing query languages such as rdfDB, RDQL, SeRQL

SPARQL Syntax

- SPARQL query to find the URL of a contributor's blog:

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
SELECT      ?url
FROM        <bloggers.rdf>
WHERE  {
            ?contributor foaf:name “Dave Beckett" .
            ?contributor foaf:weblog ?url .
}
```

  - PREFIX indicates prefix for FOAF namespace
  - SELECT indicates what the query should return
  - FROM optional clause indicating the URI of the dataset to use
  - WHERE triple patterns expressed in Turtle syntax (graph pattern)

Using SPARQL with Jena

- SPARQL is supported in Jena via ARQ module
  - It also understands RDQL queries

- Need to import package: *com.hp.hpl.jena.query*
- *QueryFactory.create()* returns a *Query* object from a file or String
- *QueryExecutionFactory.create(query, model)* returns a *QueryExecution* object
- *QueryExecution* supports various methods:
  - *execSelect()* returns a *ResultSet*
  - Apart from SELECT, you can apply the following types of queries:
    - ASK, DESCRIBE, CONSTRUCT

Example

```
// Open the bloggers RDF graph from the filesystem
InputStream in = new FileInputStream(new File("bloggers.rdf"));

// Create an empty in-memory model and populate it from the graph
Model model = ModelFactory.createMemModelMaker().createModel();
model.read(in,null); // null base URI, since model URIs are absolute
in.close();

// Create a new query
String queryString =
        "PREFIX foaf: <http://xmlns.com/foaf/0.1/> " +
        "SELECT ?url " +
        "WHERE {" +
        "      ?contributor foaf:name \"Jon Foobar\" . " +
        "      ?contributor foaf:weblog ?url . " +
        "      }";

Query query = QueryFactory.create(queryString);

// Execute the query and obtain results
QueryExecution qe = QueryExecutionFactory.create(query, model);
ResultSet results = qe.execSelect();

// Output query results
ResultSetFormatter.out(System.out, results, query);

// Important - free up resources used running the query
qe.close();
```

Refining SPARQL Queries

- DISTINCT used with SELECT

  SELECT DISTINCT

- Used with SELECT clause:
  - LIMIT n → shows upto n results
  - OFFSET n → ignores first n results
  - ORDER BY var → sorts results by normal ordering
    - ASC(?var) and DESC(?var)

More complex queries

- RDF is often used to represent *semi-structured* data. This means that two nodes of the same type in a model may have different sets of properties.

- Optional matches

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
SELECT      ?name ?depiction
WHERE {
            ?person foaf:name ?name .
            OPTIONAL {
                        ?person foaf:depiction ?depiction .
            } .
}
```

- Alternative matches

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
PREFIX      rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT      ?name ?mbox
WHERE {
            ?person foaf:name ?name .
      {
            { ?person foaf:mbox ?mbox } UNION { ?person foaf:mbox_sha1sum ?mbox }
      }
}
```

More complex queries

- Using filters

```
PREFIX      rss:  <http://purl.org/rss/1.0/>
PREFIX      xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX      dc:  <http://purl.org/dc/elements/1.1/>
SELECT      ?item_title ?pub_date
WHERE {
            ?item rss:title ?item_title .
            ?item dc:date ?pub_date .
            FILTER xsd:dateTime(?pub_date) >= "2005-04-01T00:00:00Z"^^xsd:dateTime
            &&
            xsd:dateTime(?pub_date) < "2005-05-01T00:00:00Z"^^xsd:dateTime
}
```

Working with Multiple Graphs

- The model after the *FROM* clause is the background graph
- Several graphs can be specified after the *FROM NAMED <URI>* clause
- Named graphs are used within a SPARQL query with the *GRAPH* keyword

- Example: find people found in two named FOAF graphs

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
PREFIX      rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT      ?name
FROM NAMED  <jon-foaf.rdf>
FROM NAMED  <liz-foaf.rdf>
WHERE {
            GRAPH <jon-foaf.rdf> {
                        ?x rdf:type foaf:Person .
                        ?x foaf:name ?name .
            } .
            GRAPH <liz-foaf.rdf> {
                        ?y rdf:type foaf:Person .
                        ?y foaf:name ?name .
            } .
}
```

Working with Multiple Graphs

- Example: determining which graph describes different people

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
PREFIX      rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT      ?name ?graph_uri
FROM NAMED <jon-foaf.rdf>
FROM NAMED <liz-foaf.rdf>
WHERE {
            GRAPH ?graph_uri {
                        ?x rdf:type foaf:Person .
                        ?x foaf:name ?name .
            }
}
```

Combining Background Data and Named Graphs

- Example: getting a personalized live PlanetRDF feed

```
PREFIX      foaf: <http://xmlns.com/foaf/0.1/>
PREFIX      rss:  <http://purl.org/rss/1.0/>
PREFIX      dc:   <http://purl.org/dc/elements/1.1/>
SELECT      ?title ?known_name ?link
FROM        <http://planetrdf.com/index.rdf>
FROM NAMED  <phil-foaf.rdf>
WHERE {
            GRAPH <phil-foaf.rdf> {
                        ?me foaf:name "Phil McCarthy" .
                        ?me foaf:knows ?known_person .
                        ?known_person foaf:name ?known_name .
            } .

            ?item dc:creator ?known_name .
            ?item rss:title ?title .
            ?item rss:link ?link .
            ?item dc:date ?date.
}
ORDER BY DESC(?date) LIMIT 10
```

Ontologies in Jena

- They are treated a special type of RDF model, *OntModel*
  - This interface allows to manipulate programmatically an ontology:
    - Create classes, property restrictions

- Alternatively:
  - Statements meaning semantic restrictions can be added to an RDF model
  - Merge an ontology model with a data model with *Model.union()*

- Examples:

```
// Make a new model to act as an OWL ontology for WordNet
OntModel wnOntology = ModelFactory.createOntologyModel();

// Use OntModel's convenience method to describe
// WordNet's hyponymOf property as transitive
wnOntology.createTransitiveProperty(WordnetVocab.hyponymOf.getURI());

// Alternatively, just add a statement to the underlying model to express that hyponymOf is of type
    TransitiveProperty
wnOntology.add(WordnetVocab.hyponymOf, RDF.type, OWL.TransitiveProperty);
```

Inference in Jena

- Given an ontology and a model Jena can inference statements not explicitly expressed
- OWLReasoner applies OWL ontologies over a model to reason
- Example:

```
// Make a new model to act as an OWL ontology for WordNet
OntModel wnOntology = ModelFactory.createOntologyModel();
...
// Get a reference to the WordNet plants model
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
Model model = maker.openModel("wordnet-plants",true);

// Create an OWL reasoner
Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();

// Bind the reasoner to the WordNet ontology model
Reasoner wnReasoner = owlReasoner.bindSchema(wnOntology);

// Use the reasoner to create an inference model
InfModel infModel = ModelFactory.createInfModel(wnReasoner, model);

// Set the inference model as the source of the query
query.setSource(infModel);

// Execute the query as normal
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```

Jena Generic Rule Engine

- Jena Rule Engine supports rule-based inference over RDF graphs using:
  - Forward-chaining
  - Backward-chaining
  - Hybrid execution engine

- Implemented as class: *com.hp.hpl.jena.reasoner.rulesys.GenericRuleReasoner*
  - Requires a RuleSet to define its behaviour
    - A set of *com.hp.hpl.jena.reasoner.rulesys.Rule*