ORAC: An RDF/OWL metadata store using MySQL for persistence, Java servlets for interface, and Jena for modeling

Steven E. Matuszek

A paper submitted to the
Computer Science Electrical Engineering Department
in partial fulfillment of the requirements for the M.S. degree at
University of Maryland Baltimore County

December 7, 2003

some revisions March 7, 2004

Abstract

While much of the promise of the Semantic Web lies in creators describing their own content, much of it surely also lies in the ability to describe other people's content. Since these descriptions cannot be attached to the content, a centralized server for metadata is needed. Further, a Web-based front end is needed to allow the largest number of people to contribute metadata. Jena is an open-source toolkit from HP Labs that provides a Java API for RDF models, and that implements persistence through a relational database. This project undertakes to create Java servlets that can present a familiar Web-based interface to users, accept arbitrary RDF schemas, store corresponding RDF metadata, expose the metadata as HTML or as RDF/XML, and perform simple queries and inference.

Contents

Introduction to the problem domain

The easiest way for the Semantic Web to come to fruition would be for everyone to just learn RDF and start annotating their pages. After all, the author of a page knows best what it represents. Then everyone could go back and annotate old pages, and images.

But pages get deep-linked to, causing startled Old Economy companies to file lawsuits. Copyrighted images get found, and stolen, using Google Image Search.

Worst of all may be creators that *do* want their content to be found. These creators are likely to lie in their metadata. How long was AltaVista an effective search engine before people were putting hundreds of keywords in their META tags?

Thus, we cannot exclusively rely on content creators to provide metadata.

A combination of creator-supplied metadata and community-supplied metadata is probably the answer. Google's PageRank algorithm is widely known to consider both (the community-supplied metadata being how many people link to a resource, in what contexts). [18]

If you collect enough metadata, and run inference rules over them, you might even get intelligent-seeming conclusions. Paul Ford writes in a news article—a news article set in 2009—

> So the guess has always been that you need a whole lot of syntactically stable statements in order to come up with anything interesting. In fact, you need a whole brain's worth—millions. Now, no one has proved this approach works at all, and the #1 advocate for this approach was a man named Doug Lenat of the CYC corporation, who somehow ended up on President Ashcroft's post-coup blacklist as a dangerous intellectual and hasn't been seen since. [5]

AI in-jokes aside, we agree that a critical mass of collaborative filtering is crucial to having worthwhile metadata. Ford cites eBay (user feedback), Amazon (user reviews), and Google (zeitgeist authority designation).

RDF, or Resource Description Framework, appears to be an excellent starting point for this. It is a knowledge description language, but with the added wrinkles that anyone with a Web server can contribute their own ontologies of properties using RDF Schema, and that all subjects of statements must be represented by a URI (Uniform Resource Identifier). This doesn't necessarily mean a computer-retrievable UR*L*, but every URL is a URI, so it's easy to make statements about a Web page or image, and only a short leap to people, places, and so forth.

Introduction to the architecture

The original Orac was an artificial intelligence that appeared in the BBC science fiction series *Blake's 7*. It had a unique ability to retrieve data from any computer that utilized the "Tarial cell" architecture, which was every computer in the Federation [15]. This would make it very powerful, but it would also raise serious privacy concerns.

My Orac is a collection of Java servlets that interact with the Jena API, which is backed by a MySQL relational database management system, on a Red Hat Linux 9.0 server running the Apache Tomcat Web server and servlet engine. I will discuss each of these components.

Linux, Apache, and MySQL

All this software is free, and all of it except Java is open source. Has there been a grassroots content revolution since desktop publishing in which people had to pay for the software?

Clark [4] points at RSS and FOAF, two applications of RDF, as vehicles that will help to get the open source community involved in the Semantic Web, despite its lack of interest in artificial intelligence or information retrieval *per se*.

Java servlets

Servlets are a server-side technology. Once invoked by the Web server's servlet container, they stay in memory, which makes them very efficient for maintaining connections to databases, or to other back-end resources such as an RDF store. And because they are written in Java, they have full-featured control logic, abstract data structures, strong typing, and can utilize any Java API, be it academic, enterprise, or open-source. Naturally, the Servlet API provides special functions for handling form submissions and other HTTP events, and it is all hidden from the user, who sees only pure generated HTML.

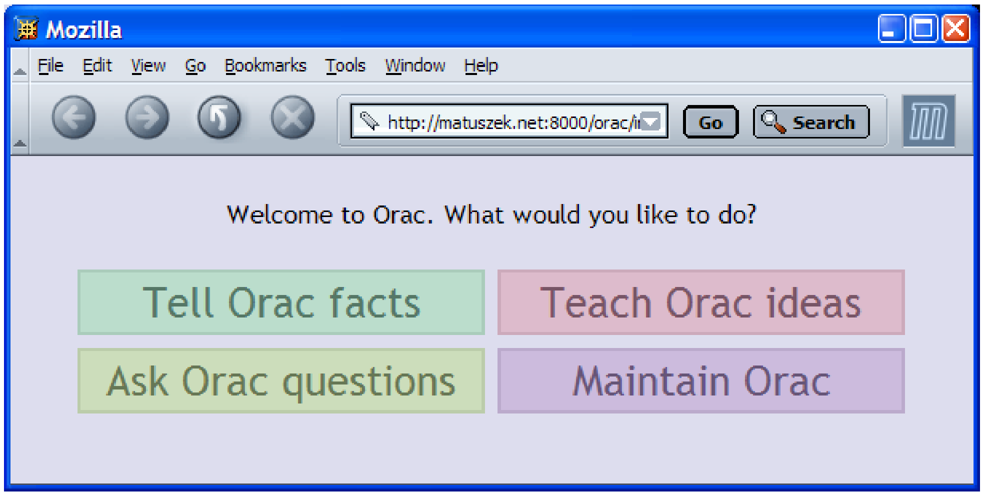
Why is this important? The pure Web is the most effective means of collecting information. No Java applets, no downloaded toolbars, just input fields and clickable links. The higher the startup cost of your process, the less it is taking advantage of an immeasurable resource: other people's time.

Jena

Jena is an open-source RDF toolkit written in Java. As such, it can be combined with a Web-based front end easily. All of the building blocks of RDF are first-class citizens in Jena: classes include **`Model`**, **`Resource`**, **`Property`**, **`Statement`**, and **`Literal`**. More, they are related in all the appropriate ways: **`Model.listStatements()`** returns a collection of **`Statement`**s, **`Statement.getSubject()`** returns a **`Resource`**, which has methods like **`getURI()`**, **`Statement.getObject()`** returns an **`RDFNode`** which can be either a **`Resource`** or a **`Literal`**, and so forth.

Orac's four-story architecture

The front page of Orac presents the user with four *user stories* [17]:

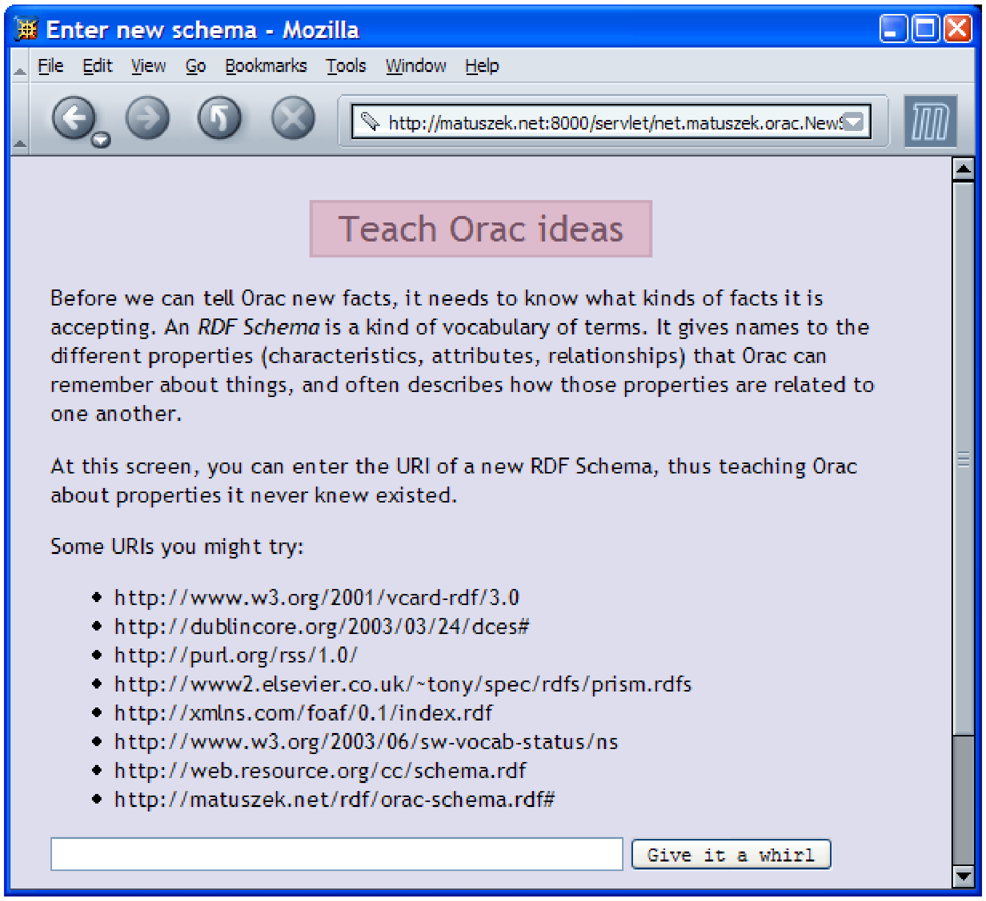

Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/orac/i
Go
Search

Welcome to Orac. What would you like to do?

Tell Orac facts
Teach Orac ideas
Ask Orac questions
Maintain Orac

Teach Orac ideas

We'll begin here, although it is unlikely to be the most comfortable choice for the beginning user. When we speak of teaching Orac ideas, we refer to making it aware of new RDF properties that it can store. This is done by providing it with an RDF Schema.


Enter new schema - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.NewS
Give it a whirl

Teach Orac ideas

Before we can tell Orac new facts, it needs to know what kinds of facts it is accepting. An *RDF Schema* is a kind of vocabulary of terms. It gives names to the different properties (characteristics, attributes, relationships) that Orac can remember about things, and often describes how those properties are related to one another.

At this screen, you can enter the URI of a new RDF Schema, thus teaching Orac about properties it never knew existed.

Some URIs you might try:

- http://www.w3.org/2001/vcard-rdf/3.0
- http://dublincore.org/2003/03/24/dces#
- http://purl.org/rss/1.0/
- http://www2.elsevier.co.uk/~tony/spec/rdfs/prism.rdfs
- http://xmlns.com/foaf/0.1/index.rdf
- http://www.w3.org/2003/06/sw-vocab-status/ns
- http://web.resource.org/cc/schema.rdf
- http://matuszek.net/rdf/orac-schema.rdf#

One might think that interpreting an RDF Schema would be relatively straightforward, but in fact it can be quite difficult. I took as my model the vCard schema, which is a common example domain for RDF. Some of its properties are simple:

```
<rdf:Description ID="NICKNAME">
    <rdf:type
     rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <rdfs:label>Nickname</rdfs:label>
</rdf:Description>
```

quite naturally results in triples of the form

```
<JohnSmith   vCard:NICKNAME   “J-Dog”>,
```

but the canonical instantiation [8],[13] of the **`vCard:N`** property is that its substructure is preserved by having an anonymous node as its argument, which in turn has the properties **`vCard:Given`**, **`vCard:Family`**, etc:

```
<JohnSmith   vCard:N   anon0>
<anon0   vCard:Given   "John">
<anon0   vCard:Family  "Smith">
```

This is represented in the schema by **vCard:N** having a range of **#NPROPERTIES**, of which **vcard:Given**, **vCard:Family**, etc. declare themselves to be subclasses.

The range of the telephone number property, **TEL**, is **TELTYPES**. (This is in fact an unqualified URI, being neither fully qualified nor **#**local. The model that we store is a cleaned-up version of the original schema; I have a method that goes through and fixes the unqualified URIs.) **TELTYPES** does not have subclasses as does **NPROPERTIES**; instead it has several instances:

```
<rdf:Description ID="TEL">
   <rdf:type
    rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
   <rdfs:label>Telephone</rdfs:label>
   <rdfs:range rdf:resource="#TELTYPES"/>
</rdf:Description>

<rdfs:Class rdf:ID="TELTYPES"/>
   <TELTYPES rdf:ID="home"/>      <TELTYPES rdf:ID="msg"/>
   <TELTYPES rdf:ID="work"/>      <TELTYPES rdf:ID="fax"/>
   <TELTYPES rdf:ID="cell"/>      <TELTYPES rdf:ID="video"/>
   <TELTYPES rdf:ID="pager"/>     <TELTYPES rdf:ID="bbs"/>
   <TELTYPES rdf:ID="modem"/>     <TELTYPES rdf:ID="car"/>
   <TELTYPES rdf:ID="isdn"/>      <TELTYPES rdf:ID="pcs"/>
```

resulting in these sample triples, which have a **TELTYPE** as their **rdf:type** and a literal for an **rdf:value**:

```
<JohnSmith  vCard:TEL  anon1>
 <anon1  rdf:value  "615-788-4467">
 <anon1  rdf:type   vCard:work>
<JohnSmith  vCard:TEL  anon2>
 <anon2  rdf:value  "301-232-3232">
 <anon2  rdf:type   vCard:home>
```

The **ADR** property is the most entangled of all, having a range of **ADRTYPES**, which like **TELTYPES** has several instances, and which has a subclass **ADRPROPERTIES**, which like **NPROPERTIES** has subclasses **Pobox**, **Street**, **Country**, and so forth, which we presume to take as component parts of an address, just as family name, given name *et al.* are component parts of a name.

```
<JohnSmith  vCard:ADR  anon3>
 <anon3  rdf:type  vCard:home>
 <anon3  vCard:Street  "111 Lake Drive">
 <anon3  vCard:Locality  "Malvern">

<JohnSmith  vCard:ADR  anon4>
 <anon4  rdf:type  vCard:work>
 <anon4  vCard:Street  "1000 Hilltop Circle">
 <anon4  vCard:Locality  "Catonsville">
```

The conclusion is that if a property's range has subclasses, we should assume them to be component parts of the property (such as Street-Locality-Region-Country), and if a property's range has instances, we should assume them to be different instances of the property (cell phone, work phone, home phone, etc.)

This is of interest because we would like to automatically generate forms for the end user to fill out, and he is not going to want to understand all about anonymous nodes. A hand-crafted form would naturally combine subclasses (given name, family name) into a grouping, and make multiple copies of an input for multiple instances of a property (cell phone, work phone). The hand-crafter would also be likely to intuit much of the semantics of the schema, such as that most people do not have more than one first name.

Thus, we need an algorithm for generating the fields, correctly grouped and/or duplicated.

The problem is that an HTML form is simply a bag of inputs, with no hierarchical structure. The author of the form could put related fields together for the user, but the servlet would be getting all the objects, half the predicates, and only one of the subjects.

The necessary insight turned out to be to record the notional anonymous nodes that were our spirit guides as we grouped inputs together. Along with every visible input, I put two hidden inputs defining the subject (whether it be the protagonist of the data entry or an anonymous node) and the property (whether it be a property from the local schema or a property from RDF/RDFS). Two integer variables allow us to sequentially number the statements and the anonymous nodes.

The final algorithm:

```
X = 0;      //  for naming statements
Y = 0;      //  for naming nodes

Read in all the properties ?property from the RDF Schema

--- start table (a)

For each ?property
{
   --- start row (a)

   if ?property is a subclass of something else
   {
      Skip it
      // it presumably is a component of another property
   }
   else if ?property has no range
   {
      make a label ?property.label

      make a hidden-input [subject_X   = "root"]
      make a hidden-input [predicate_X = ?property.uri]
      make an input       [object_X    = _____________]
      X++
   }
   else
   {
      get ?property's range as rangeResource
```

```
find all Resources ?instance such that
    <?instance rdf:type rangeResource>

find all Resources ?subprop such that
    <?subprop rdfs:subClassOf rangeResource>
        OR (<?subprop rdfs:subClassOf ?dummy> AND
            <?dummy   rdfs:subClassOf rangeResource>)

if (rangeResource has no ?subprops  AND
    rangeResource has no ?instances)
{
   // <subject  PZ  something>
   //    <something  rdf:type  RZ>

   make a label ?property.label
   make a notice "Please enter an RZ."

   make a hidden-input [subject_X   = "root"]
   make a hidden-input [predicate_X = ?property.uri]
   make an input       [object_X    = _____________]
   X++
}
else if (rangeResource has    ?subprops  AND
         rangeResource has no ?instances)
{
   //  <subject  P0  anon1>
   //    <anon1  SP0a  value>
   //    <anon1  SP0b  value>
   //    <anon1  SP0v  value>

   make a label ?property.label

   make a hidden-input [subject_X   = "root"]
   make a hidden-input [predicate_X = ?property.uri]
   make a hidden-input [object_X    = "anon_Y"]
   X++

   --- start table (b)

   for each ?subprop
   {
      --- start row (b)

      make a label ?subprop.name

      make a hidden-input [subject_X   = "anon_Y"]
      make a hidden-input [predicate_X = ?subprop.uri]
      make an input       [object_X    = _____________]
      X++

      --- end row (b)
   }
   Y++

   --- end table (b)
}
```

```
else if (rangeResource has no ?subprops  AND
         rangeResource has    ?instances)
{
   // <root  P1  anon2>
   //    <anon2  rdf:type   T1a>
   //    <anon2  rdf:value  value>
   // <root  P1  anon3>
   //    <anon3  rdf:type   T1b>
   //    <anon3  rdf:value  value>
   // <root  P1  anon4>
   //    <anon4  rdf:type   T1c>
   //    <anon4  rdf:value  value>

   make a label ?property.label

   --- start table (b)

   for each ?instance
   {
      --- start row (b)

      make a hidden-input [subject_X   = "root"]
      make a hidden-input [predicate_X = ?property.uri]
      make a hidden-input [object_X    = "anon_Y"]
      X++

      make a label ?instance.name

      make a hidden-input [subject_X   = "anon_Y"]
      make a hidden-input [predicate_X = rdf:type]
      make a hidden-input [object_X    = ?instance.uri]
      X++

      make a hidden-input [subject_X   = "anon_Y"]
      make a hidden-input [predicate_X = rdf:value]
      make an input       [object_X    = _____________]
      X++

      Y++

      --- end row (b)
   }

   --- end table (b)
}
else if (rangeResource has ?subprops  AND
         rangeResource has ?instances)
{
   // <root  P2  anon5>
   //    <anon5  rdf:type  T2a>
   //       <anon5  SP2a  value>
   //       <anon5  SP2b  value>
   //       <anon5  SP2c  value>
   // <root  P2  anon6>
   //    <anon6  rdf:type  T2b>
   //       <anon6  SP2a  value>
   //       <anon6  SP2b  value>
```

```
         //       <anon6  SP2c  value>
         // <root  P2  anon7>
         //    <anon7  rdf:type  T2c>
         //       <anon7  SP2a  value>
         //       <anon7  SP2b  value>
         //       <anon7  SP2c  value>

         make a label ?property.label

         --- start table (b)

         for each ?instance
         {
            --- start row (b)

            make a hidden-input [subject_X   = "root"]
            make a hidden-input [predicate_X = ?property.uri]
            make a hidden-input [object_X    = "anon_Y"]
            X++

            make a label ?instance.label

            make a hidden-input [subject_X   = "anon_Y"]
            make a hidden-input [predicate_X = rdf:type]
            make a hidden-input [object_X    = ?instance.uri]
            X++

            --- start table (c)

            for each ?subprop
            {
               --- start row (c)

               make a label ?subprop.label

               make a hidden-input [subject_X   = "anon_Y"]
               make a hidden-input [predicate_X = ?subprop.uri]
               make an input       [object_X    = _____________]
               X++

               --- end row (c)
            }
            --- end table (c)
            Y++
            --- end row (b)
         }
         --- end table (b)
      }
   }
   --- end row (a)
}
--- end table (a)
make a hidden-input [total_statements = X]
make a hidden-input [total_nodes = Y]
```

The servlet at the other end has merely to create and store all the statements, starting at **<subject_0 predicate_0 object_0>**. In fact, any form that supplies these as input names can successfully submit to the StoreData servlet.

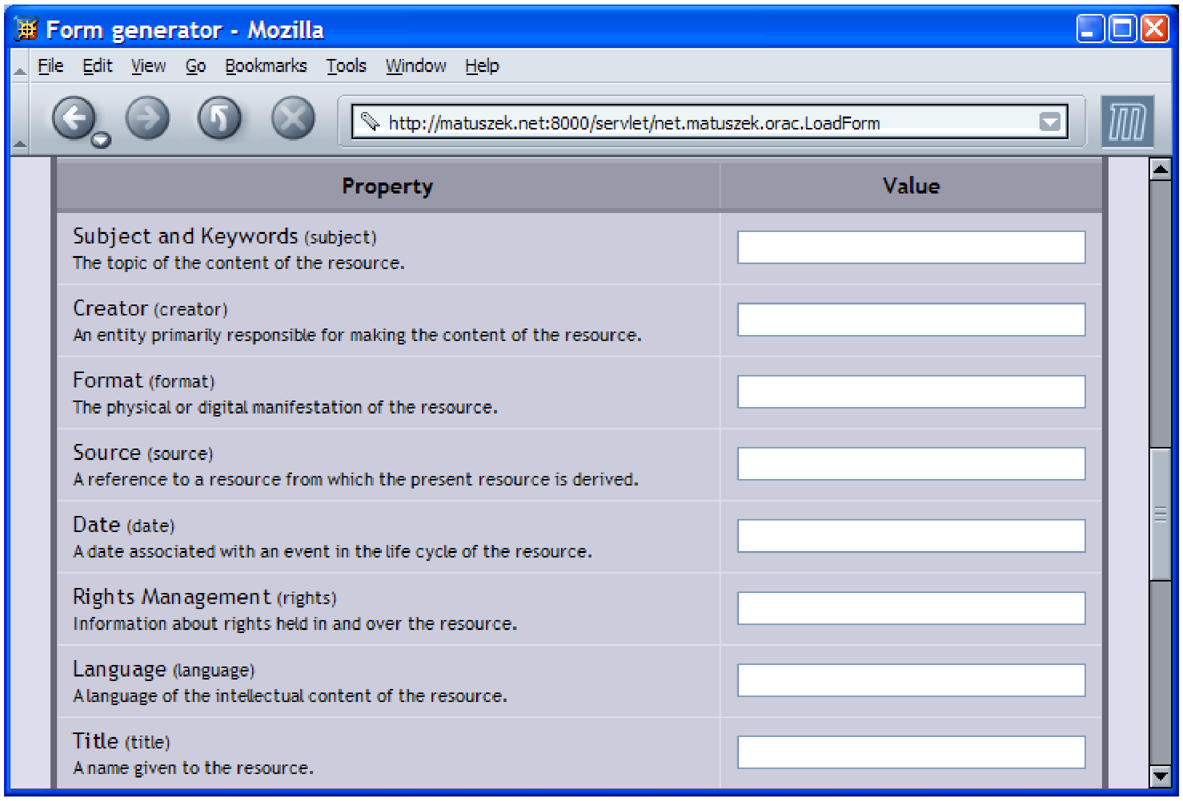
Further reflection reveals that that would still leave a lot of unneeded statements. What if the user didn't actually enter any telephone numbers? The model would include **<root vCard:TEL anon67>** without **anon67** having any useful properties. So the next step is to prune these useless entities.

That algorithm can be summed up simply:

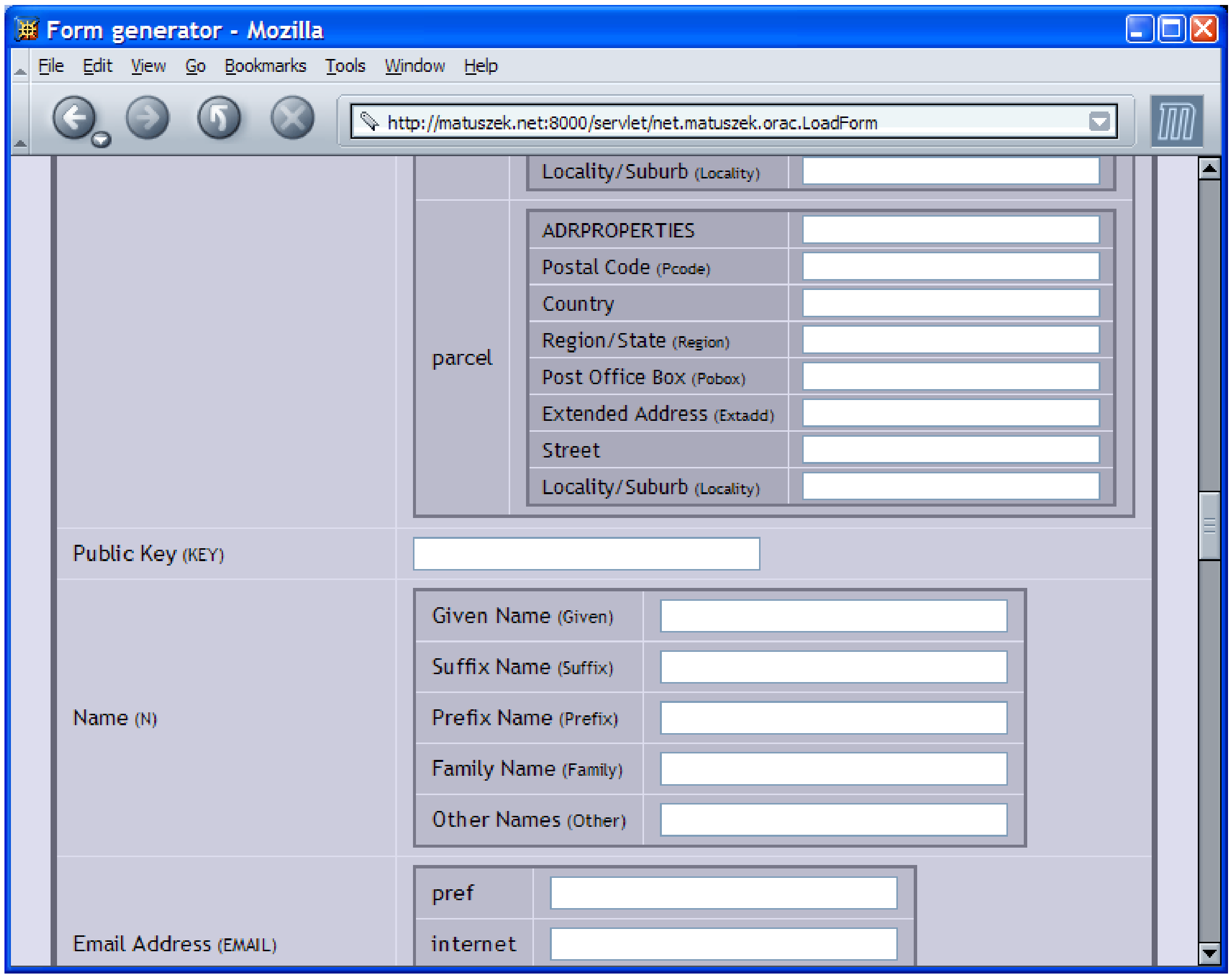
```
for each anonymous node ?a
    assume ?a is useless
    for each statement that has ?a as a subject
        if the property is not rdf:type
            ?a is not useless
            exit loop
    if ?a is still useless
        remove all statements with subject ?a
        remove all statements with object ?a
```

Tell Orac facts

Of course, we won't get there until the user enters the data. The generated form could be simple if generated from a straightforward schema like the Dublin Core:


Form generator - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.LoadForm

Property	Value
Subject and Keywords (subject) The topic of the content of the resource.	
Creator (creator) An entity primarily responsible for making the content of the resource.	
Format (format) The physical or digital manifestation of the resource.	
Source (source) A reference to a resource from which the present resource is derived.	
Date (date) A date associated with an event in the life cycle of the resource.	
Rights Management (rights) Information about rights held in and over the resource.	
Language (language) A language of the intellectual content of the resource.	
Title (title) A name given to the resource.	



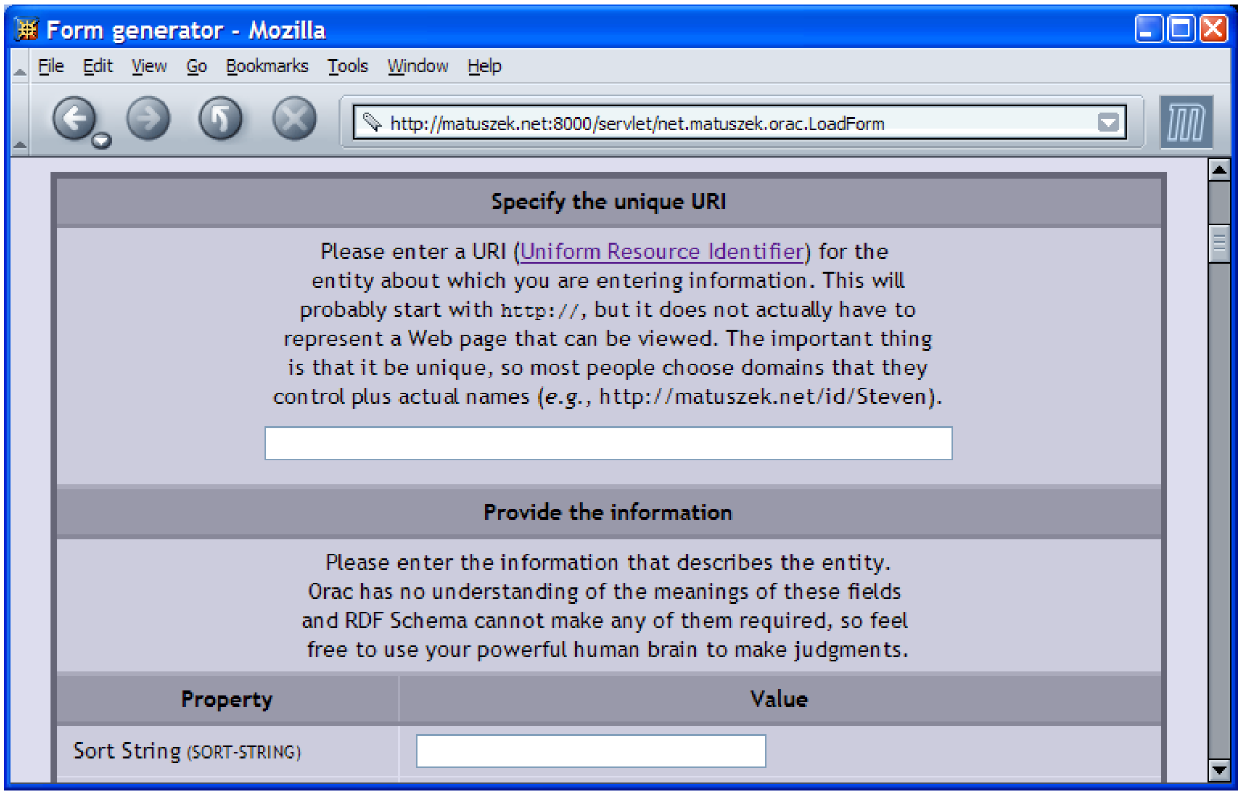
http://matuszek.net:8000/servlet/net.matuszek.orac.LoadForm

		Locality/Suburb (Locality)	
	parcel	ADRPROPERTIES	
		Postal Code (Pcode)	
		Country	
		Region/State (Region)	
		Post Office Box (Pobox)	
		Extended Address (Extadd)	
		Street	
		Locality/Suburb (Locality)	
Public Key (KEY)			
Name (N)	Given Name (Given)		
	Suffix Name (Suffix)		
	Prefix Name (Prefix)		
	Family Name (Family)		
	Other Names (Other)		
Email Address (EMAIL)	pref		
	internet		

or Byzantine, if generated from vCard or another complicated schema.

If the user clicks on **Tell Orac facts**, she gets her choice of the schemas that have already been supplied to Orac through the **Teach Orac ideas** story. Saving the models saves a little processing time, but not as much as saving the entire form would. It's too long to save as a Literal, and RDF Statements are the only way that we are maintaining any persistent state.

Orac explains to the user as much as possible about the nature of the information she is entering. Only one Resource can be described in a single form, but she can always come back and pick a different root URI.


Form generator - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.LoadForm

Specify the unique URI

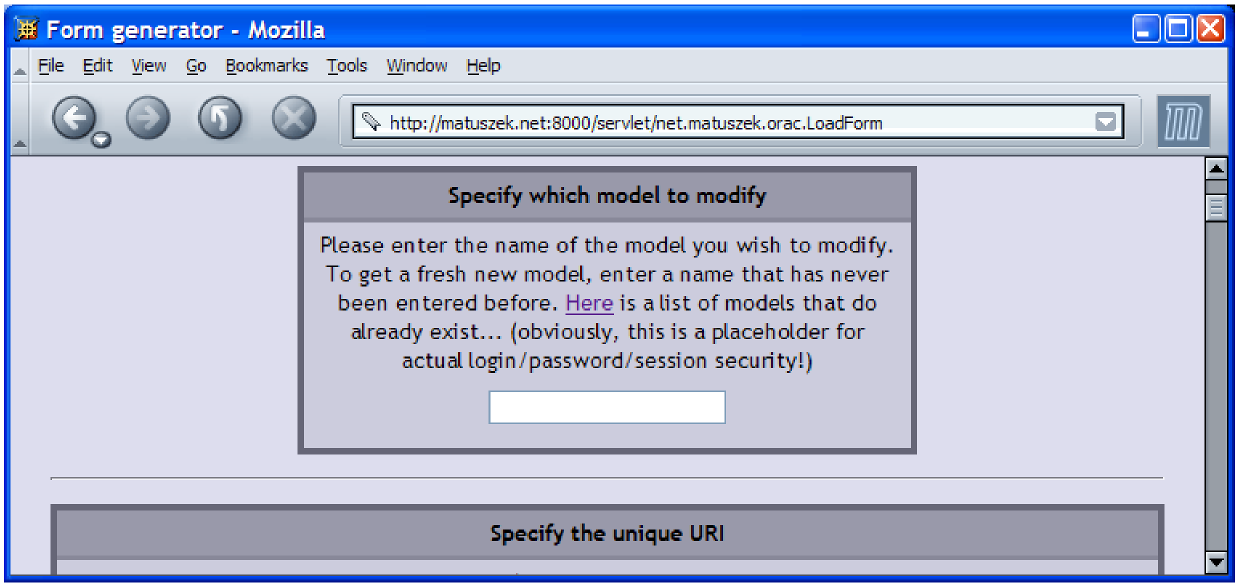
Please enter a URI (Uniform Resource Identifier) for the entity about which you are entering information. This will probably start with `http://`, but it does not actually have to represent a Web page that can be viewed. The important thing is that it be unique, so most people choose domains that they control plus actual names (*e.g.*, http://matuszek.net/id/Steven).

Provide the information

Please enter the information that describes the entity. Orac has no understanding of the meanings of these fields and RDF Schema cannot make any of them required, so feel free to use your powerful human brain to make judgments.

Property	Value
Sort String (SORT-STRING)	

Note in the screenshots that we allow the user to specify the name of the Model into which he would like to store.


Form generator - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.LoadForm

Specify which model to modify

Please enter the name of the model you wish to modify. To get a fresh new model, enter a name that has never been entered before. Here is a list of models that do already exist... (obviously, this is a placeholder for actual login/password/session security!)

Specify the unique URI

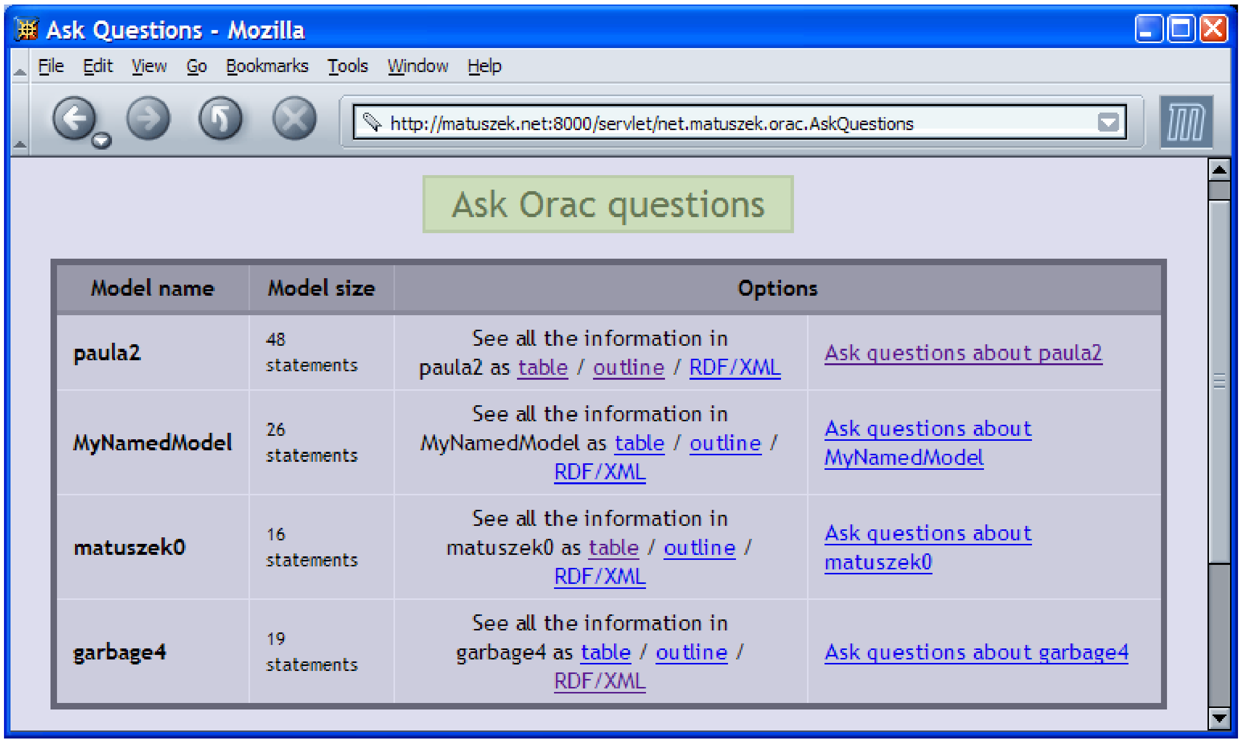
This allows each user to create his own experimental models, or to contribute to a larger, more community-oriented share of metadata. Obviously, you would want some kind of real security on this in a production setting.

Even a password would be a very binary form of security: you either can modify or you can’t. A common theory for how to incorporate the ideas of others is a *Web of Trust*, in which you can assign degrees of trust to other people or entities, and use those values to decide whether to draw conclusions from their metadata [14]. Of course, *trust* is a loaded term: a similar objective would be to track on a person-by-person basis how likely you are to agree with their explicitly subjective tastes [1].

Another limitation on storing data is that it is not entirely idempotent. If you tell Orac that **`<Eric loves candy>`** twice, it will not keep two copies of the statement. But if you tell Orac a statement for which we had to generate an anonymous node, the second time you do so, the anonymous node will be a different anonymous node. Even if the two anonymous nodes have the exact same properties, they will be considered discrete.

Ask Orac a question

The user can take a look at the existing models that Jena has stored in the MySQL database:


Ask Questions - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.AskQuestions

Ask Orac questions

Model name	Model size	Options	
paula2	48 statements	See all the information in paula2 as table / outline / RDF/XML	Ask questions about paula2
MyNamedModel	26 statements	See all the information in MyNamedModel as table / outline / RDF/XML	Ask questions about MyNamedModel
matuszek0	16 statements	See all the information in matuszek0 as table / outline / RDF/XML	Ask questions about matuszek0
garbage4	19 statements	See all the information in garbage4 as table / outline / RDF/XML	Ask questions about garbage4

Each model can be viewed in three different ways. The RDF/XML view returns text/plain, and can be accessed with a persistent(ish) URI, i.e. http://matuszek.net:8000/servlet/net.matuszek.orac.ViewRDFModel?which_model=paula2, so that automated tools can access it purely as data.

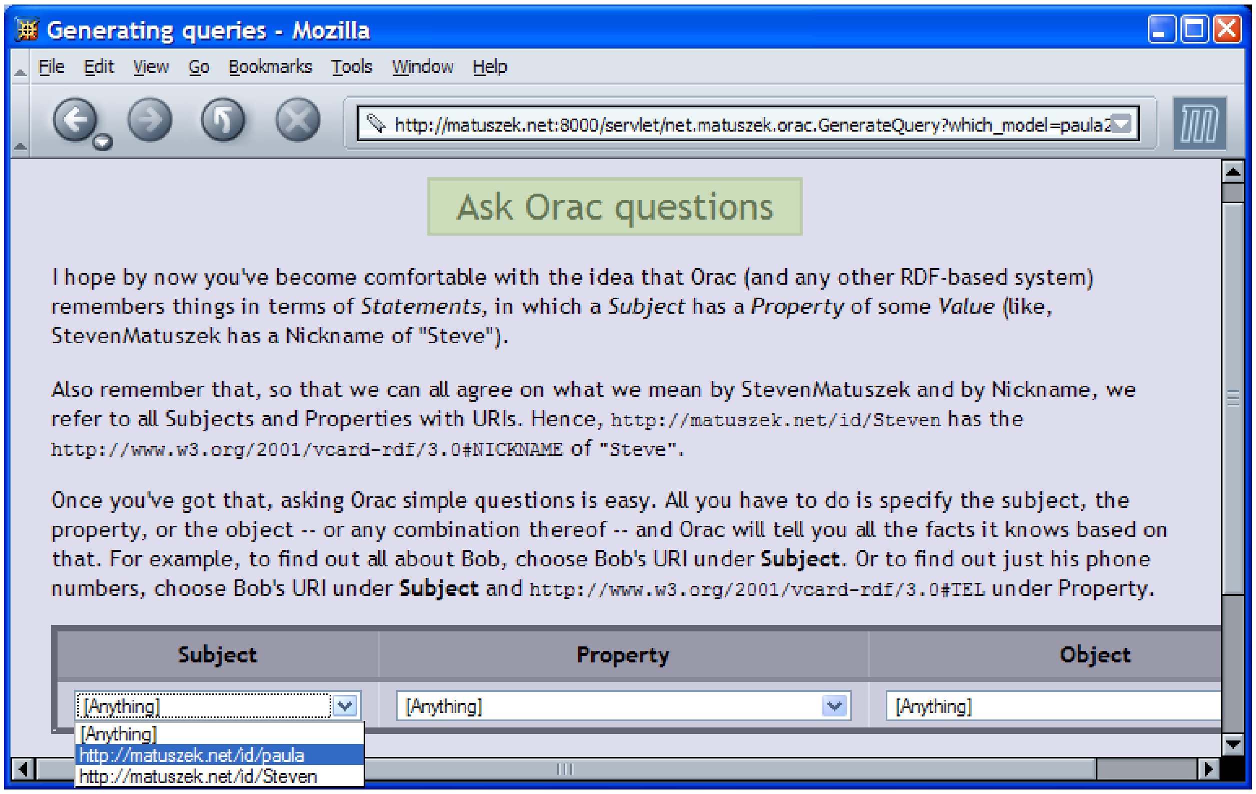
Models can also be viewed in a table of Subject → Property → Object triples, sorted by Subject, or in an outline view that folds in the anonymous nodes:

http://matuszek.net/id/paula

1. (BDAY): "Nov 17, 1946"
2. (FN): "Dr. Paula Andre Matuszek"
3. (LABEL):

        1. (type): http://www.w3.org/2001/vcard-rdf/3.0#home
        2. (Country): "USA"
        3. (Locality): "Malvern"
        4. (Pcode): "19355"
        5. (Region): "PA"
        6. (Street): "205 Paoli Pike"
    4. (N):
        1. (Family): "Matuszek"
        2. (Given): "Paula"
        3. (Other): "Marie Andre"
        4. (Prefix): "Dr."
    5. (NICKNAME): "Paula"
    6. (TEL):
        1. (type): http://www.w3.org/2001/vcard-rdf/3.0#home
        2. (value): "555-647-5555"

There is also a front end to a generic query of the statements in the model. The user can match any statement in the model on Subject, Property, Object, or any combination thereof. The servlet also goes through the model ahead of time and extracts all the Subjects, Properties, and Objects that actually appear (and which are thus the only entries that could result in a match), and puts them into pop-up menus (HTML <SELECT> inputs). This might become inefficient when the model gets very large, but at this point it is not a problem.


Generating queries - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.GenerateQuery?which_model=paula2

Ask Orac questions

I hope by now you've become comfortable with the idea that Orac (and any other RDF-based system) remembers things in terms of *Statements*, in which a *Subject* has a *Property* of some *Value* (like, StevenMatuszek has a Nickname of "Steve").

Also remember that, so that we can all agree on what we mean by StevenMatuszek and by Nickname, we refer to all Subjects and Properties with URIs. Hence, `http://matuszek.net/id/Steven` has the `http://www.w3.org/2001/vcard-rdf/3.0#NICKNAME` of "Steve".

Once you've got that, asking Orac simple questions is easy. All you have to do is specify the subject, the property, or the object -- or any combination thereof -- and Orac will tell you all the facts it knows based on that. For example, to find out all about Bob, choose Bob's URI under **Subject**. Or to find out just his phone numbers, choose Bob's URI under **Subject** and `http://www.w3.org/2001/vcard-rdf/3.0#TEL` under Property.

Subject	Property	Object
[Anything]	[Anything]	[Anything]

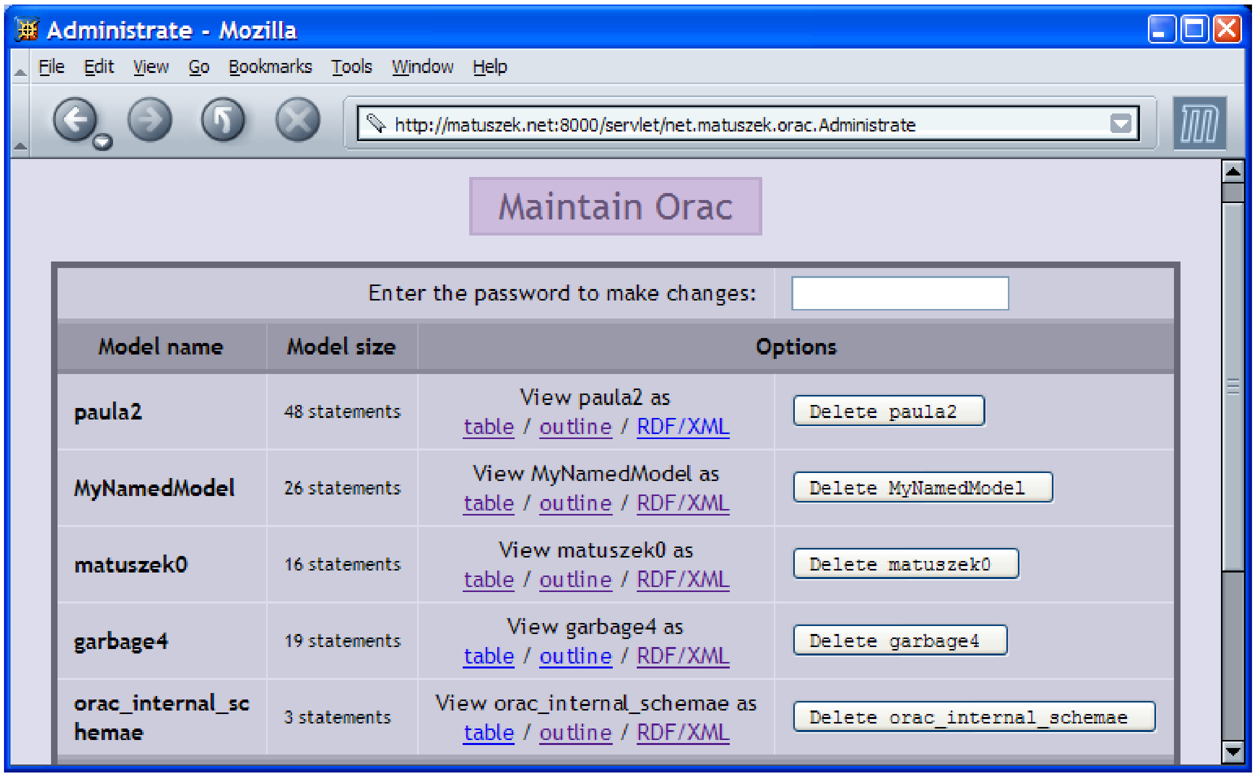
The result is displayed as an outline, and if necessary, more Statements are fetched from the original model to flesh it out. It doesn't do much good to learn that someone's telephone number is an anonymous node.

Clearly, this is the aspect in which Orac has the furthest to advance. Conjunctions and disjunctions of criteria would be simple to add.

OWL is expressed in RDF, and as such can be natively added to a Jena store. This would permit us to use more advanced semantics such as

- transitive properties: given triples like **`<A RelatedTo B>`**, **`<B Relative C>`**, correctly return both **`B`** and **`C`** when asked "**`A`** is **`RelatedTo`** ____?"
- inverse properties: given **`<w:Michelle vCard:PHOTO w:michelle.jpg>`**, match on **`<w:michelle.jpg photo:Depicts w:Michelle>`**
- disjoint classes: know that statements **`<A rdf:type YankeesFan>`** and **`<A rdf:type OriolesFan>`** cannot both obtain. [16]

Maintain Orac


Administrate - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://matuszek.net:8000/servlet/net.matuszek.orac.Administrate

Maintain Orac

Enter the password to make changes:			
Model name	**Model size**	**Options**	
paula2	48 statements	View paula2 as table / outline / RDF/XML	Delete paula2
MyNamedModel	26 statements	View MyNamedModel as table / outline / RDF/XML	Delete MyNamedModel
matuszek0	16 statements	View matuszek0 as table / outline / RDF/XML	Delete matuszek0
garbage4	19 statements	View garbage4 as table / outline / RDF/XML	Delete garbage4
orac_internal_schemae	3 statements	View orac_internal_schemae as table / outline / RDF/XML	Delete orac_internal_schemae

Currently this story allows the user to see all the models (which he can also do from **Ask Orac Questions**), and also to delete models, given a password.

Other administrative tasks slated to be added here are actions to merge models, remove individual statements by hand, add individual statements by hand, and so forth.

Summary of important Java classes

- Tell
  - EnterData
    - This servlet checks what schemas are already present in the Jena store, and allows the user to choose among them for data entry.
  - LoadForm
    - This servlet loads up a schema from the Jena store and displays the form generated from it.
  - StoreData
    - This servlet takes the statements (in the form of inputs) passed to it by LoadForm (or FormGenerator) and stores them in the specified persistent model.
- Teach
  - NewSchema
    - This servlet explains RDF Schema to the user and accepts the URI of a schema.
  - FormGenerator
    - This servlet accepts a schema URI from NewSchema, loads it into memory, cleans it up, and displays the generated form.
  - OracEcho
    - This servlet accepts a schema URI and displays the generated model for debugging purposes.
- Ask
  - AskQuestions
    - This servlet provides the user the choice of viewing or querying the existing models.
  - GenerateQuery
    - This servlet allows the user to assemble a matching query against the chosen model.
  - AnswerQuery
    - This servlet accepts the input from GenerateQuery, builds and runs the requested query, and displays the results.
  - ViewModel
    - This servlet views the statements in a model in a Subject → Property → Object table.
  - ViewOutlineModel
    - This servlet views a model in a tree structure, starting with non-anonymous subjects and descending through both simple properties and those that involve anonymous nodes.
  - ViewRDFModel
    - This servlet renders the selected model in machine-readable RDF/XML.

- Maintain
  - Administrate
    - This servlet allows the user to view or delete existing models. They need a password to delete.
  - DeleteModel
    - This servlet attempts to delete the specified model, and displays the return code from ModelPersistence.

- Utility
    - ModelPersistence
        - This utility class is a wrapper for all of the database backing of the stored RDF models. It maintains the database connection, and its static methods return existing models, create new models, delete existing models, and save and return special system models such as stored RDF schemas.
    - OracParser
        - This class encapsulates most of our Jena utility methods. It fixes unqualified URIs, generates HTML for many individual elements such as Statements or Namespaces, and returns the RDFS metadata that don't already have methods in Jena, such as comments, subclasses, and labels.
    - PrettyPrinter
        - This class provides shortcuts for several HTML patterns that are used in different places, and also the methods that generate outlines from models.
    - SchemaToForm
        - This is the class that takes in a schema and returns a form. It needed a class all by itself.

Conclusions

Observations on the domain

I like RDF for describing other people's resources, which is what this project was all about.

I found RDF Schema to be somewhat haphazard, vague and occasionally contradictory. Different schema designers seemed to have little in common when it got more complicated than simple properties. Even the designer of vCard acknowledges its kludginess:

> "Even though the **`LABEL`** property has the same substructure defined by **`N`** and **`ADR`**, we do not use them in specifying its value. This is because the value of **`LABEL`** is formatted text that is not intended to be interpreted." [8]

W3C's RDF Semantics working draft [6] couches it in terms of existential variables and entailment, formal proof that is important, but that I don't think helps in the trenches. The RDF Primer [11] more helpfully notes that RDFS is not prescriptive as far as data typing, which is probably what gave this Java programmer trouble.

Observations on the architecture

The longer you program in Java, the more any new technology—servlets, JDBC databases, cryptography—becomes Just Another API.

You decide what object you need, you look at what objects you have, and you read the javadocs to see what methods will get you there from here.

Of course, that is predicated on the API and the javadocs being complete and well written. Jena 2.0 is very complete, its documentation well written for the most part, and it has good redundancy. I learned more about RDF from programming in Jena than I did from reading papers about it. McBride writes in *An Introduction to RDF and the Jena RDF API:*

> Implementing too quickly, without first understanding the RDF data model, leads to frustration and disappointment. Yet studying the data model alone is dry stuff and often leads to tortuous metaphysical conundrums. [10]

Amen!

Future work

Of course, the questions being asked are simplistic at this stage. The original intent was to allow inference-backed selection of images. OWL Lite would appear to be the right language for this next step. Jena includes a plug-in reasoning layer, with prototype reasoners that can, for example, generate corpuses of new statements through forward-chaining.

I'd like to refine form creation to allow the user to choose which of the possible inputs they want. Rearranging the groupings would be hard to do intuitively in HTML, but one enormous improvement would be to allow the user to specify how many of each input he even wants. For example, ten copies of **`foaf:Knows`**, since everyone knows a lot of people, but zero copies of **`vCard:LABEL`**, the property even its creators disavow.

We'd want to save the generated forms in some way. That much HTML is too large to save as a Literal, alas. So it's either talk to the MySQL server directly, or save onto the local filesystem.

Final conclusion

RDF/RDFS has its limitations, but it seems to be a move in the right direction for the Semantic Web. Jena is an outstanding tool, and I predict it will become the standard for RDF application programming. Orac can't yet look at pictures and tell me which ones are of mammals, but it can read in arbitrary schemas, so any given ontology should not be too hard to add. And it is persistently storing data. While it might not set the atmosphere on fire, and I'd want to improve the security before letting in the general public, it already permits a set of trusted people to share data and metadata without having to set up a Yahoo! Group, which must surely be a worthy goal in itself.

References

[1] Joshua Allen, *Making a Semantic Web*: http://www.netcrucible.com/semantic.html

[2] Dan Brickley and R.V. Guha, *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft*, http://www.w3.org/TR/rdf-schema/

[3] Dave Beckett, *RDF/XML Syntax Specification (Revised), W3C Working Draft*: http://www.w3.org/TR/rdf-syntax-grammar/

[4] Kendall Grant Clark, *A Web of Rules*: http://www.xml.com/pub/a/2003/10/23/iswc.html

[5] Paul Ford, *How Google beat Amazon and eBay to the Semantic Web*: http://ftrain.com/google_takes_all.html

[6] Patrick Hayes, *RDF Semantics, W3C Working Draft*: http://www.w3.org/TR/2003/WD-rdf-mt-20031010/

[7] Kevin Hemenway, *The Semantic Web 1-2-3*, http://www.disobey.com/detergent/2002/sw123/

[8] Renato Ianella, *Representing vCard Objects in RDF/XML, W3C Note*: http://www.w3.org/TR/2001/NOTE-vcard-rdf-20010222/

[9] The Jena javadoc: http://jena.sourceforge.net/javadoc/index.html

[10] Brian McBride, *An Introduction to RDF and the Jena RDF API*: http://jena.sourceforge.net/tutorial/RDF_API/index.html

[11] Frank Manola and Eric Miller, *RDF Primer, W3C Working Draft*: http://www.w3.org/TR/rdf-primer/

[12] Andy Seaborne, *A Programmer's Introduction to RDQL*: http://jena.sourceforge.net/tutorial/RDQL/index.html

[13] Ken Taylor, *Address Book Samples*: http://mobile.act.cmis.csiro.au/sms2/xml/Address-Book-Samples.asp

[14] Aaron Swartz, *The Semantic Web in Breadth*: http://logicerror.com/semanticWeb-long

[15] *The Computers of Blake's 7*, DIVERSE UNIVERSE issue 13, 9/2002: http://www.spacedoutinc.org/DU-13/ComputersBlake.html

[16] *The Jena 2 Ontology API*, http://jena.sourceforge.net/ontology/index.html

[17] Extreme Programming concept of User Stories, http://www.extremeprogramming.org/rules/userstories.html

[18] Sergey Brin and Lawrence Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, http://www-db.stanford.edu/~backrub/google.html