Voces

Ştefan Apostoaie, Alexandru Chica, Marina Ghiucă, Irina Roznovăţ

Computer Science Faculty, Iaşi

Abstract.
This paper is documentation for a project that creates an extension for the Eclipse IDE which allows editing of XML/XHTML documents with embedded metadata vocabularies. The editor supports XML/XHTML syntax and is able to dynamically import vocabularies like FOAF, DOAP. RDF parsing of vocabularies is done with Jena. The editor consumes web services which expose vocabulary syntax. All information about this project can be taken from its site: https://code.google.com/p/voces/.

Keywords: metadata, vocabulary, Xtext, eclipse, DSL

1 Introduction

Voces is a project that combines many components in order to create a editor support for XML/XHTML documents. The main sub-systems that interact in Voces are:

- plug-in for Eclipse IDE
- web service based on RESTful
- editor based on Xtext
- RDF parser that uses Jena for RDF triplets' parsing

At the beginning of Voces's development, a components naming convention was established. The **"ro.fii.wade.voces,<component>"** template was strictly respected during development phases. In this way, someone that opens Voces for the first time can deduce that ro.fii.wade.voces.metadataParser refers to RDF parser.

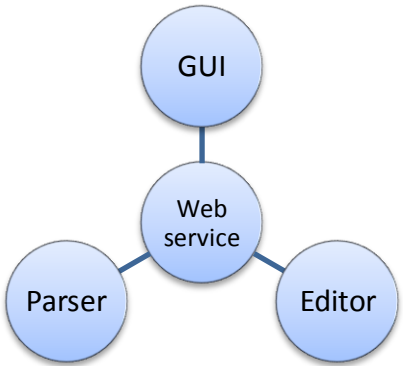

Fig. 1. Interaction between systems' components

As a synthesis for the system's components, web service interacts with all the others: it receives notification from user through a button to search for available vocabularies, it sends them to parser to extract only properly information needed by editor, and then, it passes the processed information to editor. The final step will be done by editor that updates dynamically its grammars in order to help user to edit XML/XHTML documents. An editor feature that helps user a lot is "Content highlighting extension".

2 Editor

2.1 Xtext

Xtext is a framework for development of domain specific languages (mainly programming languages). Xtext allows the user to define a language grammar using EBNF syntax. Starting from this grammar, the framework generates a parser and an Abstract Syntax Tree meta-model, together with a fully featured default editor for Eclipse. Xtext integrates with Eclipse Modeling technologies such as EMF, GMF, M2T, and parts of EMFT, so that adding new features for a DSL becomes an easy task.

2.2 Editor grammar and meta-model

For the Voces project, the following grammar was created:

```
grammar ro.fii.wade.voces.Metavoc with
org.eclipse.xtext.common.Terminals

generate metavoc
"http://www.fii.ro/wade/voces/Metavoc"
```

```
XMLModel :
        (contents+=XMLValidElement)*
        ;

XMLValidElement returns XMLValidElement :
       (startelement=XMLStartElement)
       ( (children += XMLValidElement)* | name=ID
| content=INT )
       (endelement=XMLEndElement)        ;

XMLStartElement returns XMLStartElement :
        "<" XMLElementText ">" ;

XMLEndElement returns XMLEndElement  :
        "</" XMLElementText ">" ;

XMLElementText returns XMLElementText :
        namespace=VocNS
        ":"
        nselements=NSElement
        (attributes=XMLElementAttributes)*
        ;

XMLElementAttributes returns XMLElementAttributes
:
    namespace=AttrNS
    ":"
    nselements=NSElement
    "="
    elementValue='"'name=ID'"'
    ;

AttrNS returns AttrNS :
    name=ID;

VocNS returns VocNS :
    name=ID;

NSElement returns NSElement :
     name=ID;
```

Basically this grammar defines the RDF/XML format. XMLModel is the root node of the syntax tree, which can contain zero or more XMLValidElement rules.

An XMLValidElement rule can contain a start element rule (<element>), other XMLValidElement rules or plain text (name=ID) and an end element rule (</element>).

The rules XMLStartElement and XMLEndElement contain the required XML brackets (‘<’, ‘>’, ‘</’) and a rule called XMLElementText, which contain the syntax of the XML element text. A RDF/XML element may have a namespace name, followed by a ‘:’ character and then the namespace element; as an option, the element might have one or more XMLElementAttributes rules, which describe the syntax of an XML attribute (namespace:element=”value”).

In the grammar specification, you can see the line “name=ID” referenced many times. This specifies that the rule accepts any character the user types in. the ID terminal is defined as:

```
terminal ID     : '^'?('a'..'z'|'A'..'Z'|'_')
('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

These are default terminals provided in org.eclipse.xtext.common.Terminals grammar.

From the grammar specification above, the meta-model in Fig.2 is generated.

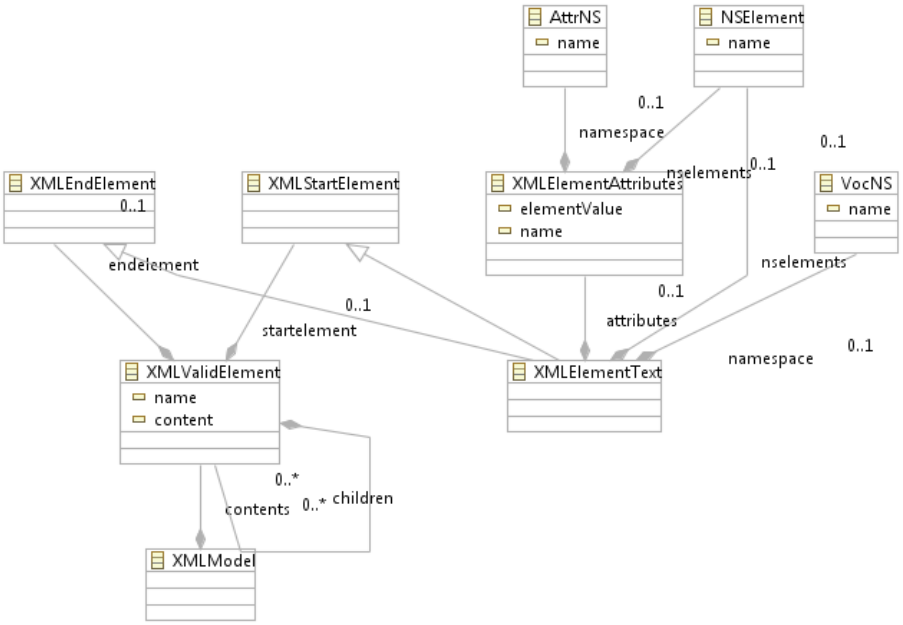

Fig. 2**.** Meta-model generated from grammar

The meta-model is used to generate the other classes needed in the application: default content assist, highlighting and verification classes.

2.3 Content highlighting extension

To customize the content highlighting in the editor, the package ro.fii.wade.voces.highlighting was added in ro.fii.wade.voces.metavoc.ui/src. This package contains the two classes that are needed to implement semantic highlighting for the model: `MetavocSemanticHighlightingConfiguration` and `MetavocSemanticHighlightingCalculator`. The first class specifies the highlighting configuration: the style and appearance of the elements; this class has to implement the `ISemanticHighlightingConfiguration` interface.

To add a specific highlighting configuration, the user has to override the configure method from the interface. It's implementation is as follows:

```
@Override
  public void
configure(IHighlightingConfigurationAcceptor
acceptor) {
    // TODO Auto-generated method stub

  acceptor.acceptDefaultHighlighting(NAMESPACE_ID,
"Namespace", namespaceType());

  acceptor.acceptDefaultHighlighting(NAMESPACE_ELE
MENT_ID, "NamespaceElement",
namespaceElementType());

  acceptor.acceptDefaultHighlighting(NAMESPACE_SEP
ARATOR_ID, "NamespaceSeparator",
namespaceSeparatorType());
  }
```

In order to register the components within the IDE, in the class MetavocUIModule, the following methods have to be added:

```
  //bind the calculator in the existing UI
  public Class<? extends
org.eclipse.xtext.ui.common.editor.syntaxcoloring.
ISemanticHighlightingCalculator>
bindSemanticHighlightingCalculator()
{
```

```
    return
MetavocSemanticHighlightingCalculator.class;
  }

  //bind the configuration in the existing UI
  public Class<? extends
org.eclipse.xtext.ui.common.editor.syntaxcoloring.
ISemanticHighlightingConfiguration>
bindSemanticConfiguration() {
    return
MetavocSemanticHighlightingConfiguration.class;
  }
```

The result is showed in Fig.3.

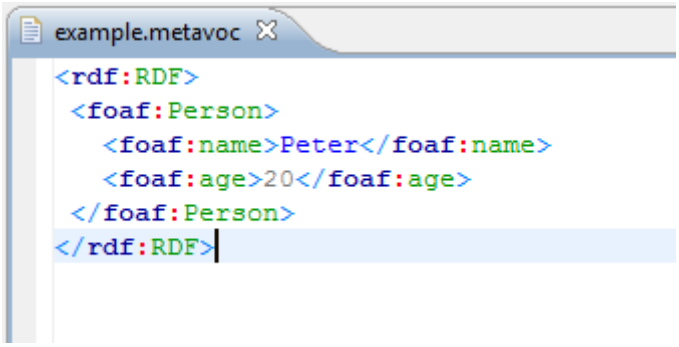

Fig. 3. Semantic highlighting

2.4 Content assist extension

To extend the default content assist, one has to extend the `AbstractMetavocProposalProvider` class in order to override the default content assist proposals.In the ro.fii.wade.voces.metavoc.ui project, the ro.fii.wade.voces.contentassist package, the above class is extended by the `MetavocProposalProvider` class, which overrides some base methods, in order to hide the default content assist suggestions, and provides new suggestions for the namespace name and namespace elements of the editor. The method `completeVocNS_Name` provides content assist proposals for the namespace name, and the method `completeNSElement_Name` provides suggestions for the namespace elements.

In order to keep the code modular, the list of suggestions is handled by the `CompletionProposalHelper` class.

The result of the content assist extension is shown below, in Fig.3.

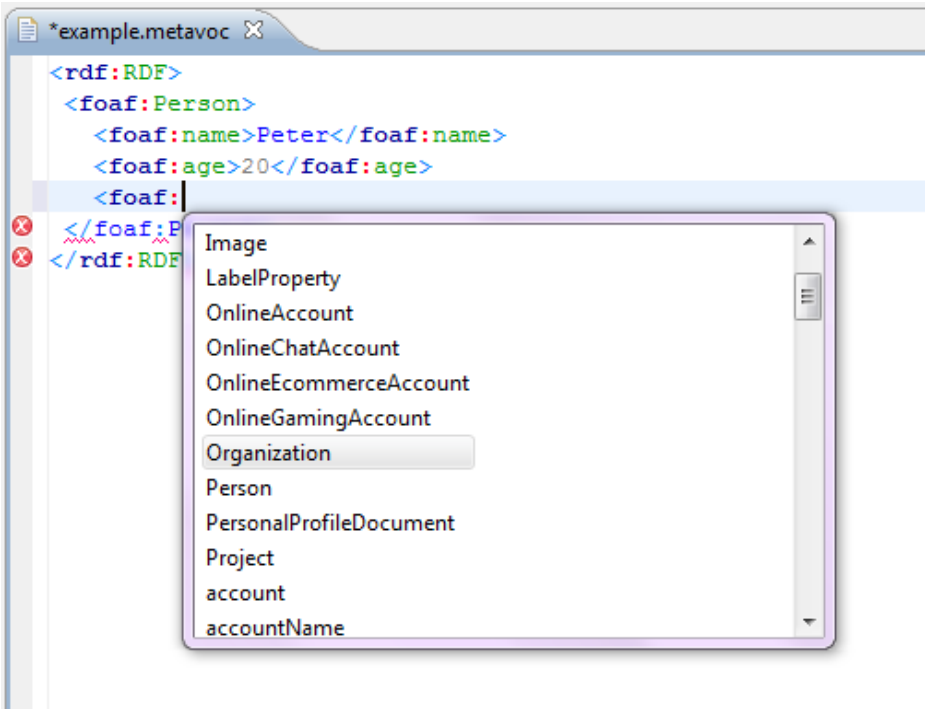

Fig. 4. Content assist example

3 Web service

This web service was created using the JAX-RS API for RESTful web services and its most popular implementation Jersey. For the web service development it was used NetBeans 6.8, which has support for the RESTful web services, meaning it has wizards for creating and tools to test them.

The JAX-RS API uses annotations to ease the RESTful Web Service creation.

@GET specifies that the method it precedes is returning something (usually a string). The format of the result is specified by the

@Produces annotation. There are several formats available: application/json, application/xml, text/xml, text/plain, text/html, etc.

@PUT is used for methods that are used to get some information from the web service user. The method that uses this annotation usually receives a String parameter with a specified format.

@Consumes is used to specify the format of the @PUT method parameter. It is used together with @PUT annotation.

@QueryParam(“”) is used to specify the name of the input parameter used by a method. It is useful for querying the service: here <query_param> is the value of the @QueryParam annotation).

The plugin uses a RESTful web service for providing the list of terminals that were extracted from the vocabularies. The web service offers two methods:

```
@GET
@Produces("application/xml")
public String getVocabulary (@QueryParam("name") String name);

@GET
@Produces("text/plain")
public String getVocabularyList();
```

The getVocabularyList service returns the list of vocabularies used in the application. Currently the list of vocabularies is limited to a number of three, but in the future another service will be provided for registering other vocabularies. More vocabularies can be found searching the internet either manually or automatic (using crawlers).

The getVocabulary returns the parsed content of the vocabulary specified by the name parameter. This method uses the Voces Parser to extract from the RDF files the relevant elements. The Voces Parser receives an URI as input and returns elements into XML format. Its role is to get the RDF from the internet and parse it for the relevant information.

```
@GET
@Produces("application/xml")
public String getVocabulary(@QueryParam("name")
String name)
{
   String vocabularyURI = ocabularyList.get(name);
   if (vocabularyURI != null)
   {
        return
ParserRDF.ParseRDFFile(vocabularyURI, fileId++);
   }
        return "";
}
```

These services are called by the editor to update the content assist and validate the files. The results can also be cached and updated periodically to

prevent frustrating lags in the editor. This can be done since there is a high possibility for the RDF vocabularies to remain stable, hence no need to reparse them very often.

By using a web service for this kind of functionality, Voces provides the plug-in extra flexibility. The user need not update the plug-in often since the main job is done on a remote server, and the update of the web service is hidden from the user (he will only see that the plug-in does more).

It's clear that the editor needs some sort of mechanism to extract the relevant information from the vocabularies. There were some options for doing that including embedding this functionality to the editor and using some remote services for that. The first option had one major disadvantage: the update of the parser implied an update of the whole plug-in. Another disadvantage was that mechanism for increasing performance (like caching) was simply excluded since the user does not want to store huge amount of data on his computer. So that is why Voces uses a RESTful web service.

4 UI Plug-in

Voces provides a user-interface plug-in for Eclipse.
From the beginning, user is assisted by Voces in order to create new XML/XHTML documents. When the user starts Voces, he can use a button to see all vocabularies that are available in that moment.

Fig. 5. Toolbar button

Accessing the button will open a new window with a list box that displays all vocabularies that were received through the web service until that moment. In this way, user is informed about the vocabularies' availability.

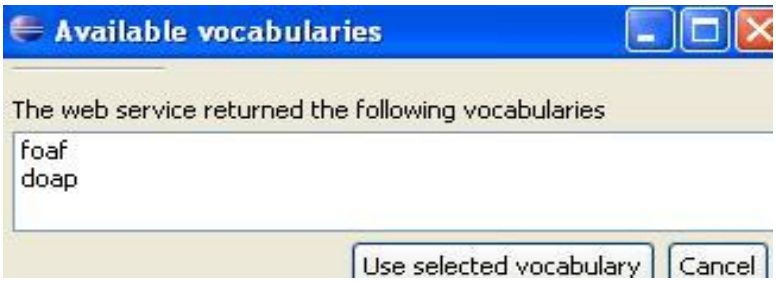

Fig. 6. Vocabularies' avalability

5 Voces Parser

In **Voces**, vocabularies are parsed using Jena. Jena is a Java API for semantic web applications. It was developed by Brian McBride into Hewlett-Packard laboratories and it is derived from SiRPAC. It can be used to create and manipulate RDF models. Jena provides some interfaces for RDF triplets' manipulation.

The Jena "Statement" interface provides methods for getting and/or setting the subject, predicate and object of a statement. The object of a statement can be either a resource or a literal and the getObject() method returns an object typed as RDFNode, which is a common superclass of both Resource and Literal. An object type can be determine using instanceOf() method.

RDFNode interface provides a common base for all the elements that can be part from a RDF triple. Literal interface refers to literals and strings that are <object> from the RDF triple. The objects that implements Container, Alt, Bag or Seq interfaces can be also seen as <object> into RDF triples. Objects that implements Property interface can be RDF triples' predicates.

To create a simple RDF/XML document, Model interface have to be used. ModelMem class creates a RDF model into memory. This class extends ModelCom class that contains all the methods for models' usage. The ModelRDB class is used to manipulate those RDF models that are stored into relational databases as MySQL, Oracle or PostgreSQL. Against ModelMem models, the ModelsRDB models are persistent.

RDF models are directly accessed through iterators: NodeIterator (for generic nodes – also resources and literals), ResIterator and StmtIterator.

Another way to access information from a RDF model is using SPARQL – query language for RDF. Results are RDF triples or sets of RDF graphs.

Voces Parser uses Jena to parse vocabularies that are received through its WEB service. Only "namespace" and "subjects" properties are extracted from vocabularies and stored into xml files. These output xml files are provided by Voces Parser to web service and later processed by it.

RDF packages used by Voces Parser are:

```
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.ResIterator;
import com.hp.hpl.jena.rdf.model.Resource;
```

The following code shows exactly how the RDF model is created with the information from the vocabularies, how its content is processed and only "subjects" properties are extracted from the model.

```
// create an empty model
Model model = ModelFactory.createDefaultModel();
…
// read the RDF/XML file
model.read(in, "");
…
//take all subjects from a RDF-triplet
ResIterator iter = model.listSubjects();

//process all the subjects
while (iter.hasNext())
{
  Resource  subject   = iter.nextResource();
  String resourceInfo = subject.toString();
  …
}
```

For example, being given as input the **FOAF** vocabulary (*http://www.foaf-project.org/*), the output will be in xml format as it is shown downstairs:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Vocabulary>
 <Namespace>foaf</Namespace>
 <Terminal>knows</Terminal>
 <Terminal>firstName</Terminal>
 <Terminal>icqChatID</Terminal>
…
</Vocabulary>
```

6 Deployment

In order to use the editor one has to export the following plug-ins:
ro.fii.wade.voces.metavoc
ro.fii.wade.voces.editorOptions

To export the plug-ins one has to right click the project, choose Export -> Plug-In Development and choose Deployable plug-ins and fragments (Fig.7).

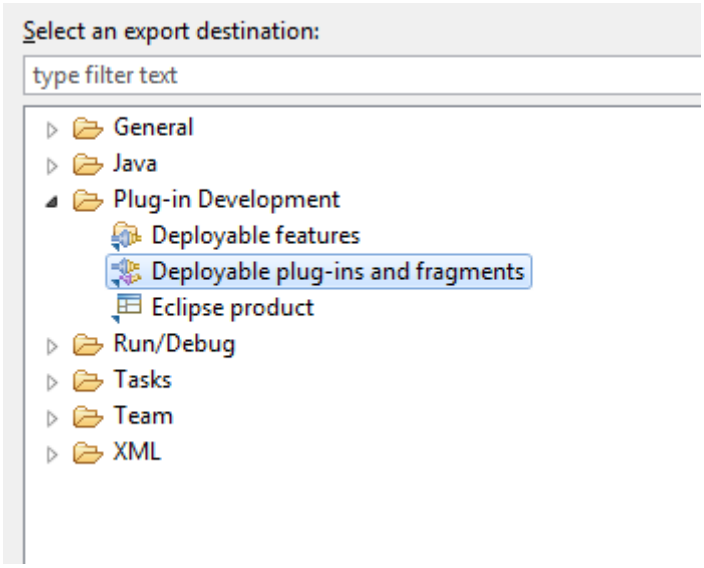

Fig. 7. Exporting the plug-in

The resulting jars have to be copied in the Eclipse plug-in directory.

After these steps, the editor plug-in will be available in the IDE. To use it one has to create a file with the '.metavoc' extension. All the features of the editor are available only when editing .metavoc files.

Fig. 8 represents the deployed application in Eclipse.

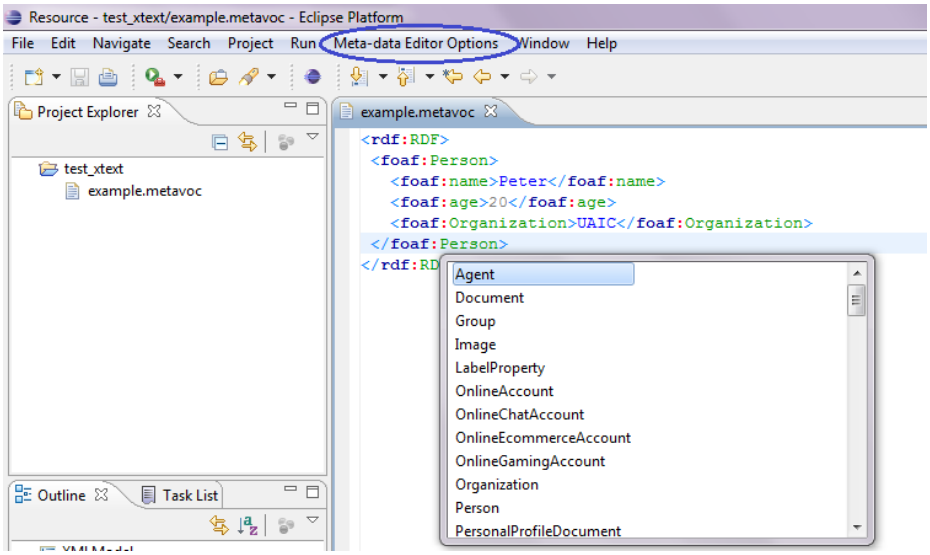

Fig. 8. Editor deployed

7 Conclusion

Voces is a project that creates an extension for the Eclipse IDE which allows editing of XML/XHTML documents with embedded metadata vocabularies. It contains a plug-in through that user demands web service to search new vocabularies. Another Voces component is a web service that interacts with parser and also with editor. The Voces parser uses Jena to manipulate RDF information. The editor updates dynamically its content and in order to help user to edit XML/XHTML documents, providing useful features like content highlighting, content assist and syntax verification.

8 References

[1] http://www.eclipsepluginsite.com/
[2] http://www.vogella.de/articles/EclipsePlugIn/article.html
[3] http://agile.csc.ncsu.edu/SEMaterials/tutorials/plugin_dev/
[4] http://jena.sourceforge.net/tutorial/RDF_API/index.html
[5] http://jena.sourceforge.net/IO/iohowto.html
[6] http://www.devx.com/semantic/Article/35906
[7] http://willdaniels.co.uk/reference/rdf-vocabulary
[8] http://www.foaf-project.org/
[9] https://trac.usefulinc.com/doap
[10] http://dublincore.org/specifications/
[11] http://www.java-forums.org/swt/9751-swt-list-example-demonstration.html
[12] http://www.eclipse.org/Xtext/documentation/
[13] http://www.peej.co.uk/articles/rest.html
[14] http://netbeans.org/kb/docs/websvc/rest.html
[15] http://www.eclipse.org/Xtext/
[16] http://www.eclipse.org/Xtext/documentation/0_7_2/xtext.html