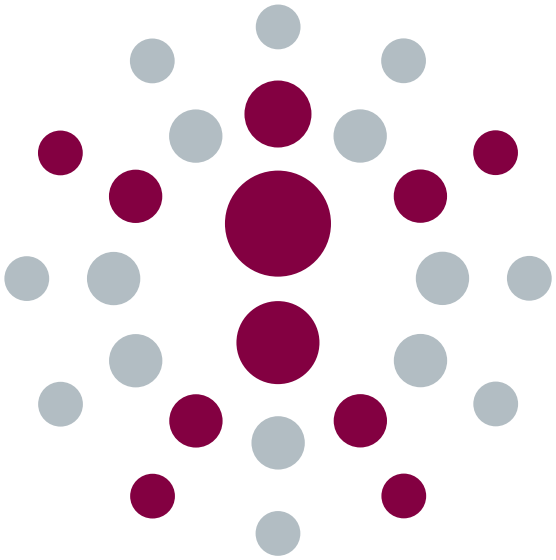

sedri

User Manual

12/2013

Contents

1 Introduction

SEDRI (formerly for *Semantic Drug Interface*) is a 3-tier framework for aggregating several knowledge bases based on SPARQL queries and serving the aggregated sources as a simple HTTP interface. SEDRI is controlled by a XML configuration file which defines the HTTP endpoints and sources that will be queried. See section 2 for the explanation of the possible config elements. SEDRI also has the goal to allow user-defined actions before the queries are send and after the results are returned. These actions are defined by Pre- and Postprocessors. See section 3 for more information about these extensions.

Figure 1.1 gives a brief overview of the three layers of SEDRI. After reading in the configuration file and recognizing a HTTP request on a given endpoint, at first the defined preprocessors are executed. Afterwards the `QueryProcessor` queries all sources and aggregates them in one RDF model. On top of this model the postprocessors are executed. Imaginable use cases for these postprocessors are Deduplication tools, Link generators or similar tools.

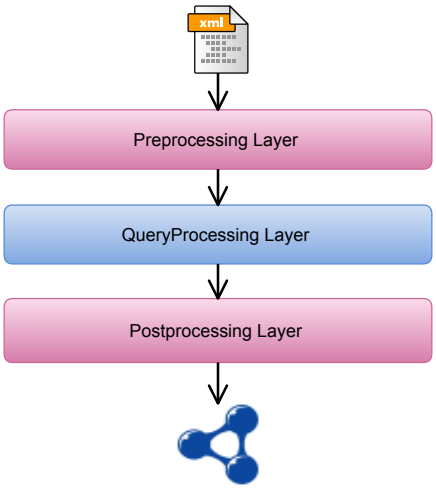

Figure 1.1: Layer Overview of SEDRI

The QueryProcessor supports `CONSTRUCT`, `DESCRIBE` and `SELECT` queries. If multiple sources are given it will be checked that only applicable query types are given. This means CONSTRUCT and DESCRIBE can be aggregated because both return resource descriptions but none of them can be aggregated with a SELECT query.

1.1 Internal architecture

The configuration syntax is defined by a XML Schema file. Based on this schema several classes are generated by `xjc` a tool of the *Java Architecture for XML Binding* (JAXB). The given configuration is parsed by the `Main` class and validated against the schema. The Main class also starts the web server which will then listen for requests. This is handled by the `Webservice` class which checks if an incoming request matches any of the given endpoint urls of the configuration. If this is the case the given parameters will be parsed and the preprocessors executed. Afterwards the QueryProcessor queries every given source with the query including all substituted variables. See section 2.1.2 and 2.1.3 for an explanation of the substitution approach. The results are stored in an RDF model which is the input for postprocessors. If the postprocessing is finished the results will be returned in an appropriate format.

Figure 1.2 shows the UML class diagram of SEDRI. The methods and attributes of the auto-generated classes of `xjc` are obmitted for readability reasons.

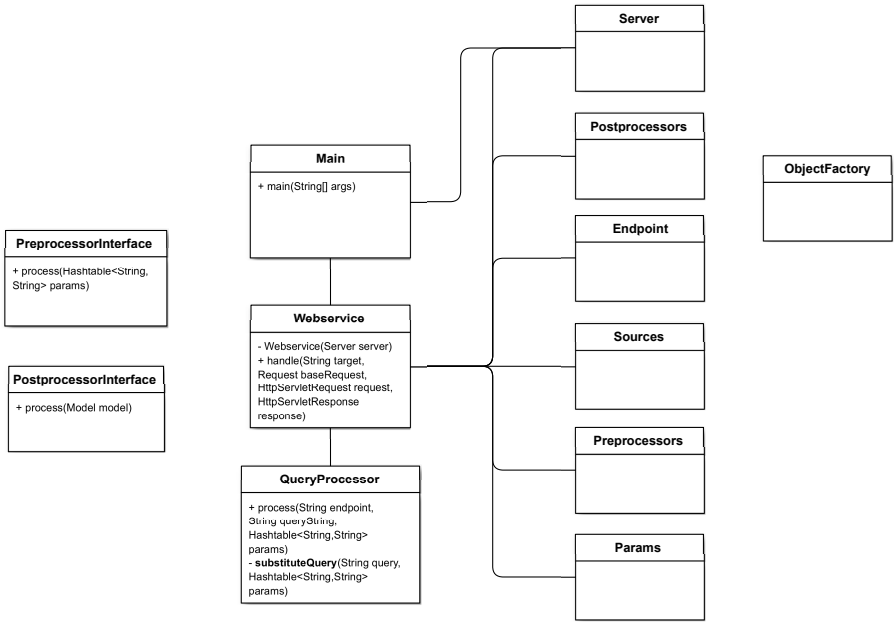

Figure 1.2: UML class diagram of SEDRI

2 Configuration

SEDRI is controlled by a XML configuration file. This configuration contains information about the server endpoints that should be opened and what sources these endpoints should aggregate. The following sections will describe the available elements.

2.1 Elements

2.1.1 Server

The main element of the configuration is `Server`. The Server element takes a parameter `port` which defines on which port the server should be started. Besides this the element takes *1..n* child elements `Endpoint`.

2.1.2 Endpoint

The Endpoint elements define several server endpoints which can be accessed through HTTP. An Endpoint element consists of a simple `url` element containing the respective url where the endpoint should listen for requests. In Addition it contains the `Params`, `Preprocessors`, `Postprocessors`, `Format` and `Sources` elements.

Params

The `Params` element is the parent element for several `Param` elements which contains the name of the expected GET parameters of the endpoint. This parameters can substitute placeholders in the configured queries so that these queries are not fixed.

Preprocessors

The `Preprocessors` element is the parent element for several `Preprocessor` elements which contains the full class name of a specific preprocessor. See section 3 for details about this extension mechanism.

Postprocessors

The `Postprocessors` element is the parent element for several `Postprocessor` elements which contains the full class name of a specific postprocessor. See section 3 for details about this extension mechanism.

Format

The `Format` element is a simple element which contains the default return format of the endpoint. This value is used if the given request doesn't provide an `Accept:` header information. Possible values are:

- RDF/XML
- RDF/XML-ABBREV
- TURTLE
- N-TRIPLE
- N3

2.1.3 Sources

Each `Endpoint` element contains a `Sources` element which contains *1..n* `Source` elements. These Source elements define which data sources should be queried.

Url

The `Url` child element of each Source element defines the url of the SPARQL endpoint which will be queried.

Query

The `Query` element contains the SPARQL query that will be send to the `Url`. By convention the variables of the given query are interpreted in different ways. Variables prefixed with a *$* expect names which are defined in the `Params` section. These variable keys will be substituted by their value at query time. So if a Param `foo` is defined then each occurence of `$foo` in the query will be substituted by their value. Variables prefixed with a *?* are interpreted as usual SPARQL variable names.

2.2 Example

```
<server port="8080">
  <endpoint>
    <url>/procure</url>
    <params>
      <param>drug</param>
    </params>
    <preprocessors>
      <preprocessor>edu.leipzig.restsparql.TestPreprocessor</preprocessor>
    </preprocessors>
    <postprocessors>
      <postprocessor>edu.leipzig.restsparql.TestPostprocessor</postprocessor>
    </postprocessors>
    <sources>
      <source>
        <url>http://drugbank.bio2rdf.org/sparql</url>
        <query>
        construct {
          &lt;http://bio2rdf.org/drugbank:$drug&gt;
          &lt;http://bio2rdf.org/drugbank_vocabulary:packager&gt;
          ?packager.

          ?packager ?p ?o.

          &lt;http://bio2rdf.org/drugbank:$drug&gt;
          &lt;http://bio2rdf.org/drugbank_vocabulary:manufacturer&gt;
          ?manufacturer.
        } where {
          &lt;http://bio2rdf.org/drugbank:$drug&gt;
          &lt;http://bio2rdf.org/drugbank_vocabulary:packager&gt;
          ?packager.

          ?packager ?p ?o.

          &lt;http://bio2rdf.org/drugbank:$drug&gt;
          &lt;http://bio2rdf.org/drugbank_vocabulary:manufacturer&gt;
          ?manufacturer.
        }
        </query>
      </source>
    </sources>
    <format>TURTLE</format>
```

```
42    </endpoint>
43    <endpoint>
44      <url>/test</url>
45      <sources>
46        <source>
47          <url>http://dbpedia.org/sparql</url>
48          <query>
49          select ?s
50          where {
51            ?s a &lt;http://dbpedia.org/ontology/Drug&gt;
52          }
53          limit 10
54          </query>
55        </source>
56        <source>
57          <url>http://drugbank.bio2rdf.org/sparql</url>
58          <query>
59          select ?s
60          where {
61            ?s a &lt;http://bio2rdf.org/drugbank_vocabulary:Drug&gt;
62          }
63          limit 10
64          </query>
65        </source>
66      </sources>
67      <format>TURTLE</format>
68    </endpoint>
69  </server>
```

3 Extensibility

3.1 Preprocessors

Preprocessors are classes that implement the `PreprocessorInterface`. This interface simply defines a `process` method that gets a Hashtable of a key value String pair and returns the same. Whatever happens inside of the method is up to the user. Usually the preprocessors will be used to convert the given parameters, e.g. in the medical domain from an ATC code to a Drugbank ID. The following listing gives a simple – and useless – example how to implement a preprocessor. In this case every request gets an additional parameter `foo` with the value *bar*.

```
1  import java.util.Hashtable;
2  import edu.leipzig.restsparql.PreprocessorInterface;
3
4  public class TestPreprocessor implements PreprocessorInterface{
5
6    public Hashtable<String, String>
7      process(Hashtable<String, String> params){
8        params.put("foo","bar");
9        return params;
10   }
11 }
```

3.2 Postprocessors

Postprocessors are classes that implement the `PostprocessorInterface`. Like the PreprocessorInterface, this interface simply defines a `process` method that gets a Jena[1] Model as input and returns the same type. Whatever happens inside of the method is up to the user. Possible use cases are the application of Deduplication tools ontop of the RDF model or Link generators. The following listing gives a simple – and useless – example how to implement a preprocessor. In this case all statements of the given model are removed and the empty model is returned.

```
1  import com.hp.hpl.jena.rdf.model.Model;
2  import edu.leipzig.restsparql.PostprocessorInterface;
3
4  public class TestPostprocessor
```

[1] `https://jena.apache.org/`

```
5       implements PostprocessorInterface{
6
7       public Model process(Model model){
8           model.removeAll();
9           return model;
10      }
11  }
```