

Integrating Semantic Search Tools into the SEALS Platform

Khadija Elbedweihi, Stuart Wrigley

October 12, 2011

Abstract

This tutorial explains how to prepare, package and zip a semantic search tool to be integrated into the SEALS platform. It explains how to implement the wrapper which acts as a bridge between the functionality of the tool and the functionality required by SEALS.

1 Introduction

Wrapping a semantic search tool consists of mainly two steps. (1) You have to place all required libraries in a certain folder structure and (2) you need a simple java class that acts as a bridge between the functionality of your tool and that expected by the SEALS platform. The expected folder structure looks like this:

```
descriptor.xml  * describes package content
bin/
  lib/
    * tool libs and other libs required by your tool
    * bridge class zipped as .jar
    * (X) some *.bat or *.sh scripts
conf/
  * files required by your tool to be accessed at runtime
lib/ (X)
  * (keep empty)
```

The SEALS platform is currently under development. To be compliant with future versions the package has to contain some elements that are currently not used. They are marked with an (X). The descriptor file is an XML file that describes the content of the package in simple and self-explaining way. You can download a full example `semanticsearch-demo-win.zip`(windows) or `semanticsearch-demo-linux.zip`(linux). In the following we show how to generate such a package and its bridge.

2 Packaging a Semantic-Search tool

In the following we assume that you are using eclipse. Note that this is not a requirement, but just helps to keep the explanation brief. These instructions require that you have downloaded `semanticsearch-demo-win.zip` or `semanticsearch-demo-linux.zip` as well as `sst-client.jar`. Our example is described briefly in Appendix A.

1. Create a new eclipse project and ensure that all libraries required by your tool are on the build path (for our example it was `json.jar` and `commons-io-1.4.jar`).
2. Add also the jar file `sst-client.jar` to your build path. It contains besides some other classes also the interface you have to implement.
3. Create some package and a class that extends `AbstractPlugin` and implements either “`ISemanticSearchToolBridge`” (for automated evaluation only) or “`UtilISemanticSearchToolBridge`” (for user-in-the-loop or the two-phases evaluation) as shown in Appendix A. (download `SSWrapper.java` as an example). This class acts as a bridge between the functionality of your tool and the functionality required by SEALS. Do not change the methods `canExecute` and `getType`. The other methods are specific for your tool.
4. Export your tool wrapper as a jar file. You do not need to include any of the libraries on your build path into that jar, only the class you have created in step 3.
5. Copy the folders of the packaged tool of our example, remove the jar files of the example and replace them by the jar files that are required by your tool.
6. Add to the directory `conf` the resources required at runtime by your system. These files will at runtime be located at the current working directory.
7. Modify the `descriptor.xml` according to these changes. Also change the meta information in the first part of the `descriptor.xml` file. You need to specify the class

bridge and the library dependencies.

8. Zip the content of the package (do not include the package folder itself in the zip file).

3 Using additional resources

Most semantic search tools require some additional resources to be available during execution in order to run properly. Examples are configuration files, lexicon files such as Wordnet, and many other kind of resources. If such resources are required by a tool, they have to be copied to the directory conf, see also Appendix A.1. In the deployment phase of the tool, the SEALS platform will copy the contents of this folder (recursively, if there are sub folders) into the working directory where the evaluation process will be executed later on.

For instance, consider that our demo tool requires the file `mysql.properties` to be available in the directory `configuration/` relative to the working directory where the tool is executed. For that reason we have placed it in the package of the tool as shown in A.1. If your tool requires some additional libraries/applications/servers that have to be installed prior to running the evaluation, you have to inform us about this. We will take care that required software is installed on the machines where your system will finally be evaluated.

4 Test your packaged tool

4.1 Validating structure and content

The jar-file `seals-sst-validator.jar` can be used to check the structure and content (e.g., correctness of references) of the package. For that purpose you need to have your tool zipped as described in the last step of §2.

In our example we called the resulting zip-file “semanticsearch-demo-linux/win.zip”. Given that in the current working directory we have “seals-sst-validator.jar” and “semanticsearch-demo-linux/win.zip”, we can check the zipped package with the following command.

```
A:\temp>java -cp seals-sst-validator.jar
eu.sealsproject.platform.res.tool.utils.validation.PackageValidator
-v all -r semanticsearch-demo-linux/win.zip
```

In case of a successful test, the following output is displayed.

```
log4j:WARN No appenders could be found for logger
(eu.sealsproject.platform.res.tool.bundle.factory.impl.DirectoryNormalizer).
log4j:WARN Please initialize the log4j system properly.
Package '/Users/kmelbedweihs/sst-tutorial-package/semanticsearch-demo-linux.zip' is valid
Package structure validation report:
- Tool package configuration: [Passed]
- Binary directory.....: [Passed]
- Configuration directory...: [Passed]
- Library directory.....: [Passed]

Package descriptor validation report:
- PackageConfiguration.....: [Passed]
- DeployCapabilityConfiguration....: [Passed]
+ Shell script configuration:
  - Script file 'deploy.sh' found.
- UndeployCapabilityConfiguration...: [Passed]
+ Shell script configuration:
  - Script file 'undeploy.sh' found.
- StartStopDependency.....: [Passed]
```

```

- StartCapabilityConfiguration.....: [Passed]
+ Shell script configuration:
  - Script file 'start.sh' found.
- StopCapabilityConfiguration.....: [Passed]
+ Shell script configuration:
  - Script file 'stop.sh' found.
- InvokeCapabilityConfiguration.....: [Passed]
+ Java application configuration:
  - Jar file 'semanticsearch-demo-bridge.jar' found.
  - Class 'uk.ac.shef.dcs.semanticsearchdemo.seals.ToolBridge' found.
  - Dependencies specified:
    + Dependency 'lib/semanticsearch-demo.jar' found and valid.
    + Dependency 'lib/json.jar' found and valid.
    + Dependency 'lib/commons-io-1.4.jar' found and valid.
- ShellScriptFileUniqueness.....: [Passed]

```

In this example the validation has been passed successfully. In case of problems, they are reported.

A A full example: semanticsearch-demo

The tool that we analyze as an example is called SemanticSearchDemo. Its functionality is encoded in a jar file semanticsearch-demo.jar. It requires the additional libraries json.jar and commons-io-1.4.jar to be available on the classpath. Further, it uses the configuration file "mysql.properties". In some of the following sections of this Appendix we show how to package SemanticSearchDemo and how to write the so called ToolBridge, which will be the interface through which the SEALS platform accesses its functionality. All the required files (jars, wrapped tool file) can be downloaded in a windows and linux-variant.

A.1 Structure of the tool package

This example is based on the Linux variant. Please download semanticsearch-demo-linux.zip and unzip it on your machine. The resulting directory structure will look like this:

```
bin/
  lib/
    semanticsearch-demo.jar
    json.jar
    commons-io-1.4.jar
    semanticsearch-demo-bridge.jar
    deploy.bat (or *.sh on linux)
    start.bat
    stop.bat
    undeploy.bat
  conf/
    mysql.properties
  lib/
    (empty)
  descriptor.xml
```

Basically, the tool wrapper goes under the bin directory together with the four script files (deployment, removal, start and stop). The four files provided with our example (which currently do not perform any action) should also be used for any other tool. The conf directory contains any files for required configurations to be provided by the platform at runtime and finally the lib directory under the bin directory contains the tool's distribution files and any additional third-party libraries required for successful execution of the tool. Some of the other folders, such as "lib", are not required by the tool. However, it is obligatory that these folders exist.

A.2 Content of descriptor

It is very easy to modify the descriptor file according to your needs. We explain it step by step.

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:package
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns="http://www.seals-project.eu/resources/res/tools/bundle/v1"
  id="SemanticSearchDemo"
  version="1.0">
  <ns:description>SemanticSearchDemo is a semantic search tool developed
    for test purposes.</ns:description>
  <ns:endorsement>
    <ns:copyright>Copyright information</ns:copyright>
    <ns:license>Specification of license</ns:license>
  </ns:endorsement>
  ...
```

In the first part, you have to define the id of your tool (chose its name or an abbreviation in case the name is too verbose (its version and a short description). You can also specify copyright and license information. While the first part contains some meta-information, the final part is about the wrapping of the tool itself. It describes where required libraries can be found inside the package.

```
<ns:wrapper>
  <ns:management>
    ...
    ...
    ...
  </ns:management>
  <ns:bridge>
    <ns:class>uk.ac.shef.dcs.semanticsearchdemo.seals.ToolBridge</ns:class>
    <ns:jar>semanticsearch-demo-bridge.jar</ns:jar>
    <ns:dependencies>
      <ns:lib>lib/semanticsearch-demo.jar</ns:lib>
      <ns:lib>lib/json.jar</ns:lib>
      <ns:lib>lib/commons-io-1.4.jar</ns:lib>
    </ns:dependencies>
  </ns:bridge>
</ns:wrapper>
```

As you can see, both the libraries and the main jar of the tool have to be specified as dependencies. In addition, a class has to be implemented that acts as bridge between the functionality implemented in SemanticSearchDemo and the interface required by the SEALS platform. The naming of this class and its package does not need to follow any pattern. In our example, it is the class “uk.ac.shef.dcs.semanticsearchdemo.seals.ToolBridge” that can be found in “semanticsearch-demo-bridge.jar”. Note also that the name of this jar can be chosen arbitrarily as long as it is correctly referenced in the file descriptor.xml. In the following section we explain how to develop the wrapper or bridge for your tool.

A.3 Development of the tool bridge

You need to implement one of the two interfaces: “ISemanticSearchToolBridge” or “UtilSemanticSearchToolBridge” depending on which evaluation phase you want to participate in. The interfaces allow us to run your tool under the SEALS platform. The functionality of the APIs can be split into three different areas: methods required in both phases, methods required by the user-in-the-loop phase and methods required for the automated phase.

A.3.1 Common methods

These API calls will be required by both phases of the evaluation and can be found in the interface “ISemanticSearchToolBridge”.

- **public boolean loadOntology(*URI ontology, String ontologyName, String ontologyNamespace*) throws ToolBridgeException, ToolException;**
This method loads a specific ontology and dataset into the tool. We provide you with the name and namespace of the ontology together with the location of the file containing the ontology. It should return true if the ontology was successfully loaded and false otherwise.
- **public boolean isResultSetReady() throws ToolBridgeException, ToolException;**
This method is used to determine if the results of a query are available by the tool or not yet. It is used (in combination with a timer) to determine how long a tool takes to answer a query.
- **public URL getResults() throws ToolBridgeException, ToolException;**
This method obtains the result list currently held by the tool. It is left to the tool provider to decide how many results to return to use for evaluation. It returns the locations of the file containing the results which should conform to the SPARQL Query Results XML Format (W3C Recommendation 15 January 2008).
- **public void showGUI(boolean show) throws ToolBridgeException, ToolException;**

This method instructs the tool to display the GUI for the user to start the evaluation process.

- **public boolean canExecute();**
This method checks whether a tool is ready to be executed. For some tools, they can be executed directly and therefore this method will directly return true. In case prerequisites are required, they can be checked before returning true or false according to the checks done.
- **public ToolType getType();**
This method is required for successful execution in SEALS platform. Please implement only the following line in this method:
return ToolType.SemanticSearchTool;

A.3.2 Automated phase

These API methods are only required by the automated phase of the evaluation and can be found in the interface “ISemanticSearchToolBridge”.

- **public boolean executeQuery(String query) throws ToolBridgeException, ToolException;**

This method executes the query. The content of the query is determined by the tool provider and will be tailored to an individual search tool (i.e., a string representation of the internal query representation). The method returns true or false denoting success or otherwise of the attempt to execute the query.

A.3.3 User-in-the-loop phase

These API methods are only required by the user-in-the-loop phase of the evaluation and can be found in the interface “UITSemanticSearchToolBridge”.

- **public void setToolInstallationPath(String path);**
This method sets the absolute path of the tool’s installation directory on the specific machine where it got installed and ready to be running. Knowledge of this path may not be necessary for full execution of the tool but it is known that some tools need this information to function correctly.
- **public String getInternalQuery() throws ToolBridgeException, ToolException;**
This method is called once the user input has been completed and the search tool has transformed the input into its own representation. It returns a String representation of this tool-specific internal query.
- **public boolean isUserInputComplete();**
This method determines whether or not the user has finished inputting the query into the tool using its interface. It would return false while the user is still typing or clicking (or thinking) and would return true once the user has hit the ‘submit’ button (or equivalent).
- **public String getUserQuery();**
This method is called once the user input has been completed. The returned string will contain the string representation of the user’s query input. For instance, if the tool uses a Natural Language interface, this method would simply return the text entered by the user.