A Query Engine for Web Service Compositions

Issa Memari

February 22, 2018

1 Introduction

The aim of this project is to create an engine for executing web service compositions where adding a new web service requires minimal effort. This is achieved by representing a web service in an XML file containing the description of the inputs and outputs of the web service along with a description of the components of its URL call, in addition to an XSLT file describing how to transform the XML response of the web service into tuples.

The engine views the call result of a web service as a table (set of tuples) and is capable of executing web service composition queries over these tables. In this report, we describe the development of this engine and along with a discussion about the shortcomings of this approach.

2 Choice of Web Service API

We choose Google Maps API to test our implementation of the query engine for three main reasons: (i) it is highly available, (ii) it is very easy to use and (iii) it provides many functions that can be composed into something meaningful.

We define web services for the following four functions:

1. Geocoding, which is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739). The signature of the function is geocode^{io}(?full_address, ?location). Where the variable ?location takes values which are the latitude and longitude of a location separated by a comma.

2. Reverse geocoding, which is the process of converting geographic coordinates into a human-readable address. The signature of the function is reverseGeocode^{io}(?location, ?full_address).

3. Place search, which takes as input a location, radius and a type and outputs Google maps place identifiers of all the points of interest of the given type that fall within distance less than radius from the given location. The signature of the function is nearbySearch^{iiio}(?location, ?type, ?radius, ?place_id).

4. Place details, which takes as input a Google maps place identifier and returns the following details about the given place: name, type, phone number, full address, rating, website and whether the place is open. The signature of the function is placeDetails^{iooooooo}(?place_id, ?name, ?type, ?phone_number, ?full_address, ?rating, ?website, ?open).

3 Implementation Details

3.1 Web Service Definitions

A web service is introduced to the engine by creating two files: (i) An XML file describing the inputs and outputs of the function call and the components of the call URL, and (ii) An XSLT file describing how to transform the call results XML file into another XML file that contains tuples.

In order to get access to the Google Maps API, one needs to create an application and obtain API keys. We created two the applications necessary to obtain the required keys and defined the files for the 4 described web services. The files are stored in the `ws-definitions` folder and are called `geocode.xml`, `geocode.xsl`, `reverseGeocode.xml`, `reverseGeocode.xsl`,`placeDetails.xml`, `placeDetails.xsl`, `nearbySearch.xml` and `nearbySearch.xsl`.

3.2 The `Query` Class

To parse and execute given queries, a class `Query` was developed. The constructor of the class takes as input the query string, it parses the query string and stores the information about the function calls in `Atom` objects. The class `Atom` is another class that we developed and it captures the abstract concept of and atom, which is a name of a function along with an ordered list of its inputs, which can be either variables or literals.

3.3 The Parser

Suppose we wish to evaluate the following query: get the names, phone numbers and addresses of all open bakeries within 500 meters from 212 Rue de Tolbiac (La Maison des Élèves de Télécom ParisTech). The following is an example of a valid way to write this query:

result(?name, ?phone_number, ?full_address) ← geocode^{io}(”212 Rue de Tolbiac”, ?location) # nearbySearch^{iiio}(?location, ”bakery”, ”500”, ?place_id) # placeDetails^{ioooooooo}(?place_id, ?name, ?type, ?phone_number, ?full_address, ?rating, ?website, ”true”).

More formally, the parser validates the given query against the regular language defined by the following regular grammar:

```
Query -> Head "<-" Body
Head -> Atom
Atom -> VariableName ( FieldSequence )
FieldSequence -> (Variable | Literal) (, (Variable | Literal))*
Variable -> ? VariableName
VariableName -> [a-zA-Z_$][a-zA-Z_$0-9]*
Literal -> [\"][^\"]*[\"]
Body -> Atom (# Atom)*
```

3.4 Query Execution

Query execution is performed by the method `execute` defined in the `Query` class, and the results are output into the file `result.xml`. Before the execution of the query starts, the query is checked for admissibility. We say that a given query is admissible if, for every variable, the first occurrence of the variable in the call sequence of the body is in an o-position of a function call.

After having checked the admissibility of the query, the execution starts. The functions are called sequentially in the order in which they appeared in the query and intermediate results are stored in the file `result.xml`. The intermediate result starts as an empty file and is then populated by the call results of the first function. Then, consecutive functions are called once for each tuple in the current intermediate result. The input arguments for the function call are the corresponding values taken from the current tuple.

After the call results are downloaded, the result of the call is joined with the file containing the intermediate results, by asserting that only the matching tuples from the result of the call are stored as the new intermediate result. After that, selection is performed by filtering out the tuples that don't match the selection criteria. And finally, projection onto the set of fields in the head atom is done by simply writing only the corresponding values from each tuple into the final results file.

3.5 How to Run

The engine can be run by creating an object of type query and passing the query string to the constructor, then calling the method execute, which runs the query and outputs the results into the file `result.xml` in the current working directory.

3.6 A Working Example

The result of the query mentioned earlier in (2.2) at the time of execution is the following XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RESULT>
<RECORD>
<ITEM ANGIE-VAR="?name">LAURENT DUCHNE Paris 13eme</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 65 00 77</ITEM>
<ITEM ANGIE-VAR="?full_address">2 Rue Wurtz, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Grard Mulot</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 81 39 09</ITEM>
<ITEM ANGIE-VAR="?full_address">93 Rue de la Glacire, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Essentials - Anthony Bosson</ITEM>
<ITEM ANGIE-VAR="?phone_number">09 66 95 27 61</ITEM>
<ITEM ANGIE-VAR="?full_address">73 Boulevard Auguste Blanqui, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">L'Imprial</ITEM>
<ITEM ANGIE-VAR="?phone_number"></ITEM>
<ITEM ANGIE-VAR="?full_address">4 Avenue Reille, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Boulangerie Ptisserie M'seddi</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 89 31 15</ITEM>
<ITEM ANGIE-VAR="?full_address">202 Rue de Tolbiac, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Aux Dlices de Kenza</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 88 64 72</ITEM>
```

```
<ITEM ANGIE-VAR="?full_address">5 Rue de l'Amiral Mouchez, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Exploitation Etablissements Pereira SARL</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 65 06 64</ITEM>
<ITEM ANGIE-VAR="?full_address">25 Rue de la Butte aux Cailles, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Le Linois</ITEM>
<ITEM ANGIE-VAR="?phone_number"></ITEM>
<ITEM ANGIE-VAR="?full_address">95 Rue de la Glacire, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Royer Loic</ITEM>
<ITEM ANGIE-VAR="?phone_number">01 45 80 12 96</ITEM>
<ITEM ANGIE-VAR="?full_address">104 Rue Bobillot, 75013 Paris, France</ITEM>
</RECORD>
<RECORD>
<ITEM ANGIE-VAR="?name">Lorette boulangerie Pains Bio</ITEM>
<ITEM ANGIE-VAR="?phone_number">09 87 19 95 01</ITEM>
<ITEM ANGIE-VAR="?full_address">48 Rue Bobillot, 75013 Paris, France</ITEM>
</RECORD>
</RESULT>
```

4 Discussion of Shortcomings

Perhaps the most obvious shortcoming of this approach is the fact that it views the response of a web service as a single relational table, which is a less expressive model than tree-like models such XML or JSON. As an example where relational tables fail in describing the outputs of a web service, we look at the response of the place details web service: each element (place) in the result might have multiple types, for example, a bus station can be associated with the two types (bus_station and transit_station). It is quite difficult to capture this information in a single relational table as we do not know exactly how many types a given place can have and we would have to somehow store a variable-sized list of types associated with every place in the response.

This problem does not only affect the ability to store all the information in a web service response, but also the ability to execute queries, for example in the case where we want to allow a certain field to take on multiple values. A possible solution which may or may not be difficult to implement is to keep the responses in their tree-like formats and try to execute the queries over these responses using specialized query languages like XPath or XQuery.

One other shortcoming is the difficulty of optimizing query execution because we do not have much control over the response, for example, it is not possible to push projections as there is no way to limit the response to output certain fields. However, optimization is not a huge concern for this type of queries because web service responses are usually bounded by the API server to be small in order to reduce response time and to allow a higher throughput.