D-K 905 Dynamic Content Management

Project Report

The goal of this project is to develop an execution engine for call compositions within a framework where adding a new web service is almost effortless. The engine shall be able to perform the following tasks:

1. Parse input string in multiple queries.
2. Validate queries by checking that every query has an input, either provided by the user or coming from a previous query.
3. Execute queries in desired order. More specifically, from the second query on the engine shall be able to retrieve results from the previous query and execute a call at the current web service for each one of them, since they are inputs of the current query.

All the three tasks have been implemented directly in the query engine, contained in the **QueryEngine** class. Utility classes have been created, namely the **Query** class and the **CallResult** class.

The following guidelines have been followed, each one with its advantages and shortcomings. In order to give clearer explanations about decisions taken during the implementation, the following example of query will be used:

```
mb_getArtistInfoByName("Frank Sinatra", ?artistId, ?beginDate, ?endDate)
#mb_getAlbumByArtistId(?artistId, ?albumId, ?albumName, ?albumYear)
#mb_getSongByAlbumId(?albumId, ?songId, ?songName, ?duration)
```

Parsing

Following formatting rules have been applied:

- Queries must be separated by a '#' character, as suggested in the assignment paper.
- Input is delimited by " " characters, and must be only one. This limitation results from the fact that queries are allowed to have parameters that are at the same time input and output, and hence don't have an output name to be matched with following query parameters. We consider the following query as an example:

  ```
  mb_getArtistInfoByName("Frank Sinatra", ?artistId, ?beginDate, ?endDate)
  #mb_getAlbumByArtistName(?artistName, ?albumId, ?albumName, ?albumYear)
  ```

  Since the input "Frank Sinatra" is unique, the engine is able to match it to the first parameter of the second level query and it has been implemented to do so, printing nonetheless a warning. But if we had the following query,

  ```
  mb_getArtistInfoByName("Frank Sinatra", ?artistId, "1964", ?endDate)
  #mb_getAlbumByArtistName(?artistName, ?albumId, ?albumName, ?albumYear)
  ```

  the engine wouldn't be able to know which results it should use as inputs of ?artistName.

  A viable alternative would be not allowing queries to have inout parameters, which would solve the problem and allow the framework to accept multiple input queries. However, since the example presented in the assignment paper contained an inout parameter, this path hasn't been followed.

- Outputs, and more in general parameters that are not part of user input, are preceded by the '?' character, and their name should match the name in the web service description. Although increasing formality, this choice has been taken in order to allow the user to use also a fewer

number of parameters than the ones present in the web service, without having to always use all of them. For example, the query

#mb_getAlbumByArtistName("Frank Sinatra", ?albumId, ?albumName, ?albumYear)

could be rewritten as

#mb_getAlbumByArtistName("Frank Sinatra", ?albumName)

and would still be a working query keeping in memory only the ?albumName parameter. Obviously, call and transformation results files remain the same.

Following format guidelines, the ***parse*** method inside the QueryEngine class takes as input the query composition string and returns a list of Query objects, initialized with their parameters.

In our example, the result of the parse method is a list of three queries: the first one will have an input parameter ("Frank Sinatra") in position 1, while others won't have input parameters – here, for input we mean user input.

Validation

The ***validate*** method implements query validation inside the QueryEngine class. It verifies that all queries from the second on have a parameter with the same name as a parameter in the previous query: in other words, it verifies that a query composition is possible by assigning output of the previous query to input of the following one.

The engine will successfully validate queries in the example: parameter "?artistId" appears in query 1 and query 2 and parameter "?albumId" appears in query 2 and query 3, thus creating a consistent cascade of queries.

Execution

In order to properly carry out the chain of queries, the ***executeQuery*** method, after parsing and validating, retrieves the input for queries, performs them and stores results, if present, in the **CallResult** class, which is then assigned to the query for future retrieval.

In order to avoid debugging issues and to facilitate code readability, both in the Query class and in the CallResult class Maps and Lists were used instead of 1d and 2d arrays. This choice may cause scalability issues at a certain point.

When executing a chain of queries, the first one will usually consist of only one call to the web service and will thus have only one CallResult object – in our example, the CallResult corresponding to input "Frank Sinatra". After the first query, Query object will own a variable number of CallResult objects: calling n the number of results of the first query, the second query will then have n CallResult objects, each one with its input and results. Third query will have to be executed on all the results gathered from the n CallResult of the second query, except obviously empty results.

Coming back to the example, the second query

#mb_getAlbumByArtistId(?artistId, ?albumId, ?albumName, ?albumYear)

will result in a number of calls equal to the number of different artists id contained in the result of the previous query. And the third query

#mb_getSongByAlbumId(?albumId, ?songId, ?songName, ?duration)

will have a number of calls corresponding to the amount of all album ids of all artists contained in the results of the second query.

To retrieve input values that define the calls, we use the same function as in the validation task to match parameters from one query to another.

The executeQuery method returns a list of Query objects containing results, in order to allow users to perform any other eventual modification to the objects or to the results.

Full run for example query (skipping intermediate output):

```
List<Query> queries = QueryEngine.executeQuery(
        "mb_getArtistInfoByName(\"Frank Sinatra\", ?artistId, ?beginDate, ?endDate)"
        +"#mb_getAlbumByArtistId(?artistId, ?albumId, ?albumName, ?albumYear)"
        +"#mb_getSongByAlbumId(?albumId, ?songId, ?songName, ?duration)");

Query last = queries.get(queries.size()-1);
System.out.println(" --- FINAL RESULTS: Songs by artits named Frank Sinatra --- ");
for ( String res : last.getAllResults("?songName") ) {
    System.out.println(res);
}
```

```
 --- QUERY EXECUTION TERMINATED ---
 --┼ FINAL RESULTS: Songs by artits named Frank Sinatra ---
I Love You
Just One of Those Things
Sunday
Wrap Your Troubles in Dreams
Taking a Chance on Love
Jeepers Creepers
Get Happy
All of Me
Lean Baby
I’m Going to Sit Right Down and Write Myself a Letter
How Could You Do a Thing Like That to Me
Why Should I Cry Over You
White Christmas
Jingle Bells
Silent Night, Holy Night
O Little Town of Bethlehem
It Came Upon the Midnight Clear
Santa Claus Is Comin' to Town
Should I
You Do Something to Me
When You’re Smiling
It’s Only a Paper Moon
My Blue Heaven
The Continental
It All Depends on You
Lover
You Go to My Head
These Foolish Things
A Ghost of a Chance
I Don’t Know Why (I Just Do)
Someone to Watch Over Me
Why Shouldn’t I?
Try a Little Tenderness
Paradise
Should I
```