Developing a virtual assistant for answering music related questions

Claudio SCALZO (scalzo@eurecom.fr)
Luca LOMBARDO (lombardo@eurecom.fr)

May 1, 2018

Contents

Chapter 1

Introduction

1.1 Why a virtual assistant?

In these years we're assisting to a huge development of virtual assistants, mainly in two different shapes: vocal assistant and chat bots. Usually, each of the tasks that the user asks to the virtual assistant are tasks that can be done in other ways; we can think about a user asking for directions to *Siri* or a user who tells his *Google Assistant* to schedule an appointment in his calendar. Each of these are tasks that can be done without the help of an assistant! So, why they are becoming so popular? The reason is of course related to the ease of use. There are tasks which are, even if easy, boring or annoying to do, and we can take as example the scheduling of an appointment: we can unlock the phone, open an app, write the name of the appointment, select the start date and the end date, select the reminder scheduling, select the option to keep it private or make it public, and select other options depending on which calendar app we're using. Easy of course, but way more complicated than saying: *"Hey Google, set me an appointment with Luca for tomorrow afternoon at 5pm, and please remind me 2 hours before"*.

Having said that, the reason behind the development of our virtual assistant should be clear: simplify the user's life when he wants to access some informations about classical music.

1.2 Scope of the project

Our project puts down its root into *DOREMUS*[1], a classical music knowledge base, mainly composed by an ontology of 64 classes and 250 properties, and a set of 17 vocabularies. The informations contained inside *DOREMUS* can be accessed thanks to a SPARQL endpoint, that lets users write queries to retrieve the set of

data they want to achieve.

Our project starts from this difficulty: how many people know SPARQL? And inside the set of people who know it, how many problems can rise using the SPARQL language? Using the right properties, checking the exact domain and ranges for each property and other annoying problems just to obtain simple results. Our bot faces this issue, making simple to obtain the most common informations the users want.

1.3 Aim and expected results

The real aim of the virtual assistant so, is to make simpler the use of the *DOREMUS* knowledge base and the retrieval of its data. But which will be our, concrete, final results? First of all, the shape of the virtual assistant is originally a chat bot. The queries are textually written and the responses are visually seen on the display of each device running a client with the bot installed. This, of course, is not the only shape the bot can have. It can be a vocal assistant, and the query can be done by voice. In each of the cases, the bot will extract the informations from the *DOREMUS* knowledge base using SPARQL queries, answering to the user's requests and without letting them know a single detail about the SPARQL query that is using or other technical details. It will also help the user to make a more detailed and precise query (guiding him with some questions), and will correct the users sentences when he does some typos or when he's wrong in writing some precise words (like artist names, instruments or musical genres).

Chapter 2

Natural Language Understanding

2.1 What is NLU?

2.2 How it works? (Papers)

2.3 State of art (1st meeting slides)

Chapter 3

Dialogflow

3.1 What is Dialogflow

3.2 Why Dialogflow

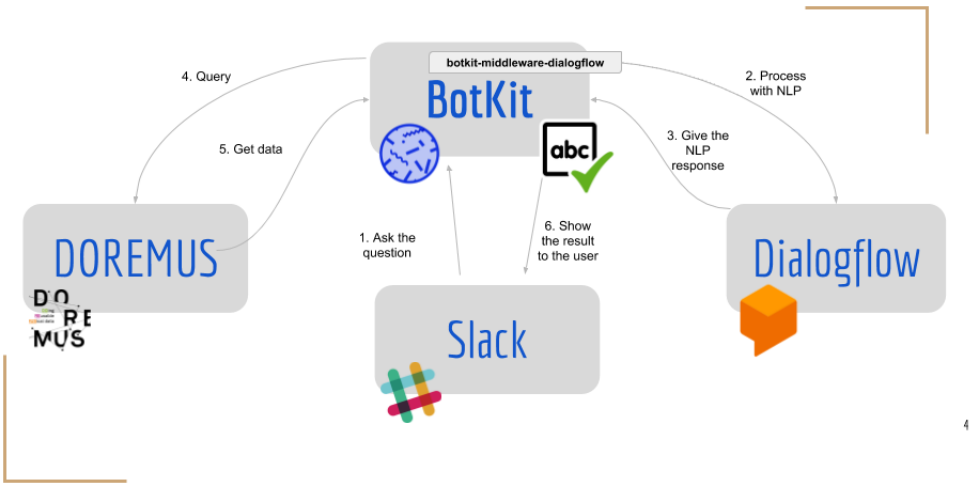
3.3 How it works?

Chapter 4

The bot

4.1 The architecture

The architecture of the bot is divided in four categories:



First of all we can find the client, that in the case of our development, testing and validation process, has been *Slack*. Of course, it can be any of the clients which support the installation of the bot on it (*Telegram*, *Facebook Messenger*, etc.). The use of Slack let us to exploit the beautiful *Slack Cards*, to make the answers of our bot (works, artists, performances) prettier and easier to understand in a glance. A set of examples will be provided in the last chapter.

The second part of the architecture is represented by the NLP. In this case, as

told in the previous chapter, we used *Dialogflow*, to exploit its advanced slot-filling techniques and its NLU power.

However, we didn't use *Dialogflow* on its own, exploiting the direct integration with *Slack*, but we used something that we placed in the middle of the two: *BotKit*. *BotKit* is a bot-making toolkit that aims to ease the building process of a bot, potentially exploiting different NLPs and/or different clients. In our case, *BotKit* has been deployed on a web server, and thanks to the *NodeJS* code we were able to come up with a series of features (like the spell checker) that would have been impossible to reach with a simple (direct) integration between *Slack* and *Dialogflow*. We'll talk more about that in the following paragraphs.

The last part of the architecture is of course represented by the data source: *DOREMUS*. We talked about that in the previous chapters, but from the architecture is important to notice how the knowledge base is queried: each query is dynamic, in the sense that according to the intent, the number of filters and the desired results wanted by the user, the query will have a different shape and a different content. The code in the web server is able to add different pieces of queries according to what *Dialogflow* is able to understand and to provide as output values of the API.

4.2 Entities

The sentences that the bot is required to recognize are of course full of informations related to the *DOREMUS* knowledge base. This means that the NLP has to be able to understand some "entities" (informations, words, piece of sentences) that are not in the standard language, but are related to the *DOREMUS* or, in general, musical world. Our bot is equipped with three different entities:

- `doremus-artist`
  Contains all the artists in the *DOREMUS* knowledge base. It's organized in a *key-value* pair where the *key* is the unique id of the artist inside the knowledge base, and the *value* is the set of full names and surnames of the artist, in all the available language of the knowledge base. The query with which the artist were retrieved is the following:

```
SELECT DISTINCT ?composer
  (GROUP_CONCAT (DISTINCT ?name; separator="|") AS ?names)
  (GROUP_CONCAT (DISTINCT ?surname; separator="|") AS ?surnames)
```

```
  (COUNT (?expression) AS ?count)
WHERE {
  ?expression a efrbroo:F22_Self-Contained_Expression .
  ?expCreation efrbroo:R17_created ?expression ;
    ecrm:P9_consists_of / ecrm:P14_carried_out_by ?composer .
  ?composer foaf:name ?name .
  ?composer foaf:surname ?surname
}
GROUP BY ?composer
ORDER BY DESC (?count)
```

The artists are ordered by descending count of works, in order to have the most famous artists at the top of the entity dictionary, and let *Dialogflow* find the most famous artists in case just a surname is given. The result of the query has been taken as a `.csv` file and then processed a little bit (delete the count, join the names and surnames columns, duplicate the composer id, eliminate the pipes and some special characters) in order to fit with the *Dialogflow*'s constraints.

- `doremus-instrument`
  Contains all the instruments of the `iaml/mop/` dictionary of the *DOREMUS* knowledge base. Also in this case the *key-value* pair dictionary has as keys the id of the instrument, and as values all the synonyms (names in different languages). The query thanks to which we retrieved this entity is:

```
SELECT DISTINCT ?instrument
  (GROUP_CONCAT (DISTINCT ?instrument; separator="|") AS ?instruments)
WHERE {
  ?instr skos:prefLabel ?instrument .
  ?instr skos:topConceptOf | skos:inScheme ?res .
  VALUES (?res) {
    (<http://data.doremus.org/vocabulary/iaml/mop/>)
  }
}
GROUP BY ?instr
```

- `doremus-genre`
  Contains all the genres of the `iaml/genre/` dictionary of the *DOREMUS* knowledge base. The query thanks to which we retrieved this entity is:

```
SELECT DISTINCT ?gen
(GROUP_CONCAT (DISTINCT ?genre; separator="|") AS ?genres)
```

```
  WHERE {
    ?gen skos:prefLabel ?genre .
    ?gen skos:topConceptOf | skos:inScheme ?res .
    VALUES (?res) {
      (<http://data.doremus.org/vocabulary/iaml/genre/>)
    }
  }
  GROUP BY ?gen
```

4.3 Intents

The intents are grouped in a simple and clear way, according to what the user wants to retrieve from the *DOREMUS* knowledge base:

- `works-by`
  Retrieves a set of works according to different filters (artists who composed the works, instruments used, music genre and/or year of composition).
- `find-artist`
  Finds a set of artists according to some filters (number of composed works, number of works of a given genre, etc.).
- `find-performance`
  Propose to the user a future performance (that can be filtered by city and/or date period), or shows to the user the details of a past performance.
- `discover-artist`
  Shows a card with a summary of an artist, with its birth/death place and date, a picture and a little bio. After the card visualization, a set of works of the artist (connection with the `works-by` intent) can be asked.

Now we're going to go deeper in the intent descriptions.

4.3.1 Retrieving a set of works

The `works-by` intent is the most complex one in the entire bot's intents set. It can retrieve a certain number of works from 1 to L, where L is the number specified by the user if it's smaller than the number of available works. Otherwise, if it's greater, all the avilable works are returned. Its default value (if not specified by the user) is 5.

The filters can be various:

- **Artist:** the artist name (full or surname).
  "Give me 3 works composed by Bach"

- **Instruments:** the instrument(s) (in `and`/`or` relation).
  "Give me 2 works for violin, clarinet and piano"
  "Tell us 4 works for violin or piano"

- **Genre:** the music genre.
  "List me 10 works of genre concerto"

- **Composition period:** the period in which the work has been written.
  "Tell me one work composed during 1811"
  "Give us 3 works written between 1782 and 1821"

The filters can be specified in every way: this means that the user can specify all the available filters, some of them and even none. If the number of filters in the first query is smaller than two, the bot asks the user if he wants to apply other filters. The user can answer positively or negatively, and then decide which kind of filter (and the value) to apply. It's important to notice that the kind of filter and the value can be specified together or not; let's see an example to make it more clear.

First of all, we are in the context in which the bot asks the users if he wants to apply some filters:

> *Please give me 3 works by Beethoven!* - **User**
> *You told me few filters. Do you want to add something?* - **Bot**
> *Yes!* - **User**
> *Ok, tell me what* - **Bot**

In this case, two scenarios can happen:

> *The composition year* - **User**
> *Of course! Tell me the time period.* - **Bot**
> *Between 1787 and 1812* - **User**

or directly...

> *Only works composed between 1787 and 1812* - **User**

The **dynamic** query used for the works by artist intent is the following:

```
SELECT DISTINCT ?title, ?artist, year(?comp) AS ?year, ?genre, ?comment, ?key
WHERE {

  ------------------------------------------------------------ STATIC SECTION
  ?expression a efrbroo:F22_Self-Contained_Expression ;
    rdfs:label ?title ;
    rdfs:comment ?comment ;
    mus:U13_has_casting ?casting ;
    mus:U12_has_genre ?gen .
  ?expCreation efrbroo:R17_created ?expression ;
    ecrm:P4_has_time-span ?ts ;
    ecrm:P9_consists_of / ecrm:P14_carried_out_by ?composer .
  ?composer foaf:name ?artist .
  ?gen skos:prefLabel ?genre .
  ?ts time:hasEnd / time:inXSDDate ?comp .
  OPTIONAL {
    ?expression mus:U11_has_key ?k .
    ?k skos:prefLabel ?key
  } .

  ------------------------------------------------------------ DYNAMIC SECTION

  ------------- Instrument
  ?casting mus:U23_has_casting_detail ?castingDetail .
  ?castingDetail mus:U2_foresees_use_of_medium_of_performance
                 / skos:exactMatch* ?instrument .
  VALUES(?instrument) {
    (<http://data.doremus.org/vocabulary/iaml/mop/' + instrument + '>)
  }

  ------------- Genre
  VALUES(?gen) {
    (<http://data.doremus.org/vocabulary/iaml/genre/' + genre + '>)
  }

  ------------- Artist
  VALUES(?composer) {
    (<http://data.doremus.org/artist/' + artist + '>)
  }

  ------------- Composition Year
  FILTER (?comp >= "' + yearstart + '"^^xsd:gYear AND
          ?comp <= "' + yearend + '"^^xsd:gYear)
}
ORDER BY rand()
LIMIT ' + num '
```

It's important to notice the particularity of the instrument information because, as previously said, it can be a list of instrument connected by an `and/or` logic operator. In the query implemented inside the `works-by` intent, this doesn't mean only to create dynamically the query (like for the other informations) but also to differentiate the code between a single-instrument filter, an "and" multi-instrument filter and an "or" multi-instrument filter. The differences are shown here:

```
---- AND CASE
if (strictly === "and") {
  for (var i = 0; i < instr.length; i++) {
    query += '?casting mus:U23_has_casting_detail ?castingDetail' + i + ' .
            ?castingDetail' + i + '
              mus:U2_foresees_use_of_medium_of_performance
              / skos:exactMatch* ?instr' + i + ' .
            VALUES(?instr' + i + ') {
              (<http://data.doremus.org/vocabulary/iaml/mop/' + instr[i] + '>)
            }'
  }
}

---- OR CASE
else {
  query += '?casting mus:U23_has_casting_detail ?castingDetail .
            ?castingDetail
              mus:U2_foresees_use_of_medium_of_performance
              / skos:exactMatch* ?instr .
            VALUES(?instr) {'

  for (var i = 0; i < instr.length; i++) {
    query += '(<http://data.doremus.org/vocabulary/iaml/mop/' +instr[i] + '>)'
  }
}
```

In the `and` case we add (with a `for` operator that loops through the `instr` array) a `castingDetail` for each instrument, forcing its value to that specific instrument. In the `or` case, instead, the `castingDetail` is just one and can assume a set of values filled with the `for` operator that loops through the `instr` array.

4.3.2 Finding some artists

4.3.3 Finding some future and past performances

The bot is also capable of giving the user some future and past performances and events. The place and the time period can be, of course, specified in the query.

The query is the following:

```
SELECT ?title, ?subtitle, ?actorsName, ?placeName, ?date
WHERE {
  ?performance a mus:M26_Foreseen_Performance ;
    ecrm:P102_has_title ?title ;
    ecrm:P69_has_association_with / mus:U6_foresees_actor ?actors ;
    mus:U67_has_subtitle ?subtitle ;
    mus:U7_foresees_place_at ?place ;
    mus:U8_foresees_time_span ?ts .
  ?place rdfs:label ?placeName .
  ?actors rdfs:label ?actorsName .
  ?ts time:hasBeginning / time:inXSDDate ?time ;
    rdfs:label ?date .
  FILTER ( ?time >= "' + startdate + '"^^xsd:date AND
           ?time <= "' + enddate + '"^^xsd:date ) .
  FILTER ( contains(lcase(str(?placeName)), "' + city + '") )
}
ORDER BY rand()
LIMIT ' + number '
```

And here there is an example:

> *Tell me one event in Paris!* - **User**
> *In which period?* - **Bot**
> *Next month* - **User**

4.3.4 Know more about an artist

The last intent of the bot is `discover-artist`. This is the richest intent in terms of informations provided: it returns the artist summary in a *Slack card*, with a picture, a short bio, the birth/date places and dates. The query is the following:

```
SELECT DISTINCT ?composer, ?name, ?bio, xsd:date(?d_date) as ?death_date,
                ?death_place, xsd:date(?b_date) as ?birth_date, ?birth_place, ?
  image
WHERE {
  VALUES(?composer) {(<http://data.doremus.org/artist/' + artist + '>)} .
  ?composer rdfs:comment ?bio .
  ?composer foaf:depiction ?image .
  ?composer schema:deathDate ?d_date .
  ?composer foaf:name ?name .
  ?composer dbpprop:deathPlace ?d_place .
  OPTIONAL { ?d_place rdfs:label ?death_place } .
  ?composer schema:birthDate ?b_date .
```

```
    ?composer dbpprop:birthPlace ?b_place .
    OPTIONAL { ?b_place rdfs:label ?birth_place } .
    FILTER (lang(?bio) = "en")
  }
```

And here there is an example:

> *Tell me something about Mozart* - **User**

or...

> *What do you know about Beethoven?* - **User**

4.4 The flow

4.5 The spell checker

Bibliography

[1] http://www.doremus.org/ — DOREMUS: DOing REusable MUSical data.